HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

CLOUD GUARD

Bachelorarbeit

Intelligente Güterwagen

13. Juni 2018

Florian Bitterlin, Giuliano De Gani, Dominik Thamm

Betreuer
Prof. Beat Stettler

Industriepartner
CloudGuard Software AG, Zürich

Experte
Marco Facetti

Gegenleser
Prof. Dr. Farhad Mehta

Studiengang Informatik

Hochschule für Technik Rapperswil

Abstract

Ziel der vorliegenden Bachelorarbeit war es, einen Prototyp für eine IoT (Internet of Things) Plattform zu realisieren, welche es ermöglicht, IoT-fähige Geräte zu verwalten. Zudem sollten über das öffentliche Mobilfunknetz Sensordaten von diesen Geräten übermittelt und von der Plattform visualisiert und analysiert werden können. Zu diesem Zweck wurden zunächst bestehende IoT-Dienstleister evaluiert und in einem weiteren Schritt die Kernfunktionalitäten abstrahiert, um einen vollständigen Gerätezyklus abzubilden. Dieser reicht von der Provisionierung bis hin zur Dekommissionierung von Geräten.

Die gesamte IoT-Plattform wurde mit Hilfe von Java basierten Technologien umgesetzt. Sie setzt sich aus mehreren Komponenten zusammen. Die RESTful HTTP API, realisiert mit dem Spring Boot Framework, dient zur Anbindung der Benutzeroberfläche. Eine Verbindung zu AWS IoT, ein IoT-Service von Amazon, ermöglicht den Datenaustausch mit Geräten. Spring Cloud Data Flow erlaubt das Orchestrieren, Verarbeiten und Analysieren von Datenströmen. Das Web UI, umgesetzt mit React, erleichtert die Bedienung.

Die entwickelte IoT-Plattform zeichnet sich durch eine einfache Handhabung im Gerätemanagement und hohe Flexibilität im Deployment (on premise oder cloud) aus. Sie bietet dank der geschickten Abstraktion der IoT-Funktionalität, die Möglichkeit, den IoT-Service Anbieter auszutauschen ohne dabei den Code anpassen zu müssen.

Inhaltsverzeichnis

Management Summary

Problemstellung

Im Schienengüterverkehr tätige Unternehmen haben heutzutage meist wenig Möglichkeiten, ihr Rollmaterial zu überwachen. Dies stellt die Unternehmen insbesondere vor Probleme, wenn ihre Güterwagen international eingesetzt werden. So ist oft der Standort eines Güterwagens, sowie dessen Rückkehrzeitpunkt nicht genau bekannt. Um eine bessere Planung und Überwachung des Rollmaterials zu ermöglichen, sollen die Güterwagen mit Hilfe von Sensoren aus der Ferne überwacht werden können. Dabei geht es nicht nur um die Standortermittlung, sondern zum Beispiel auch um die Ermittlung des Materialverschleisses und die Alarmierung bei Grenzwertüberschreitungen (z.B. Temperatur, Erschütterungen usw.).

Vorgehen

Anhand von Ausschreibungen der SBB Cargo und der ÖBB wurden die Anforderungen an ein solches Projekt analysiert. Es wurden Technologien für eine Plattform evaluiert, welche das Management von Sensorgeräten während ihres gesamten Lebenszyklus erlauben sowie das Sammeln und die Analyse der Gerätedaten ermöglichen.

Die Entwicklung eigener Hardware war nicht Teil der Arbeit. Die Sensoren wurden daher mit einfach programmierbaren Entwicklungsgeräten der Firma Pycom simuliert. Für die Kommunikation mit den Geräten wird das in der Industrie übliche MQTT-Protokoll eingesetzt. Als Messagebroker kommt eine bereits bestehende IoT-Lösung zum Einsatz. Die dazugehörende Schnittstelle ist soweit abstrahiert, dass keine starke Abhängigkeit zu einem bestimmten Anbieter besteht. Das Kernstück der Anwendung basiert auf dem Java Spring Boot Framework. Es bietet einerseits eine allgemeine REST Web API für die Anbindung der Benutzeroberfläche, andererseits bindet es eine Schnittstelle zur Steuerung der IoT-Lösung ein. Über die mit React implementierte Benutzeroberfläche ist es möglich, Geräte zu verwalten und konfigurieren, Sensordaten einzusehen und zu exportieren. Durch den Einsatz von Spring Cloud Dataflow ist die Datenverarbeitung flexibel und die Analysemöglichkeiten sind weiter ausbaubar. Die Kommunikation zu den Geräten wurde für den Prototyp mit AWS IoT umgesetzt.

Ergebnisse

Das Ergebnis ist ein Prototyp, welcher die grundlegenden Funktionen der angedachten Plattform umsetzt. So kann der ganze Lebenszyklus eines Geräts abgebildet werden. Es ist möglich neue Geräte zu erstellen, gegenüber dem System zu autorisieren und mit einer Konfiguration zu versehen. Sensordaten einzelner Geräte sind im GUI als Liste oder als Graphen einsehbar. Es wird auch eine Filter- und Sortierfunktion nach Zeit angeboten. Verschiedene Gerätetypen können dank Konfigurationstemplates, welche auch über das GUI verwaltet werden, individuell initialisiert werden. Für die Ausmusterung kann den Geräten der Zugang zum System vom Benutzer wieder entzogen werden.

Ausblick

Basierend auf dem Prototyp kann nun ein Produkt für den produktiven Einsatz entwickelt werden. Grosses Potential besteht bei Komfortfunktionen im Benutzerinterface, z.B. bei der Verwaltung von Konfigurationen oder zusätzlichen Filtereinstellungen. Weitere Möglichkeiten sind eine durch den Benutzer konfigurierbare Datenverarbeitung, sowie die Anbindung von Drittsystemen des Kunden. Für grosse Datenmengen sollten noch Verbesserungen bei der Darstellung umgesetzt werden. Je nach Anwendungsfall muss die Gerätehardware entsprechend angepasst werden. Dank des flexiblen Designs ist die Plattform auf keine spezifische Geräteart beschränkt.

Teil I.

Projektplan

1. Projektübersicht

1.1. Zweck und Ziel

Im Rahmen der Arbeit sollen eine mögliche Hardware- und Softwarelösung mit Prototyp für eine IoT-Plattform evaluiert und implementiert werden.

Grundlage für die Projektplanung bildet die Aufgabenstellung. Siehe Anhang.

1.2. Lieferumfang

Der Lieferumfang der Arbeit umfasst

- einen lauffähigen Prototyp, welcher das evaluierte Verfahren umsetzt.
- den Quellcode des Prototypen.
- die Hardware des Prototyps.
- die Build-Scripts.
- ein Benutzerhandbuch und eine Installationsanleitung.
- den Bachelor-/Studienarbeitsbericht (inkl. Doku gemäss HSR).
- das Abstract.
- ein Poster in elektronischer Form.

1.3. Annahmen und Einschränkungen

Das Projekt beschränkt sich auf die in der Aufgabenstellung gegebenen Anforderungen und den von der HSR vorgegebenen Zeitraum und Aufwand.

2. Projektorganisation

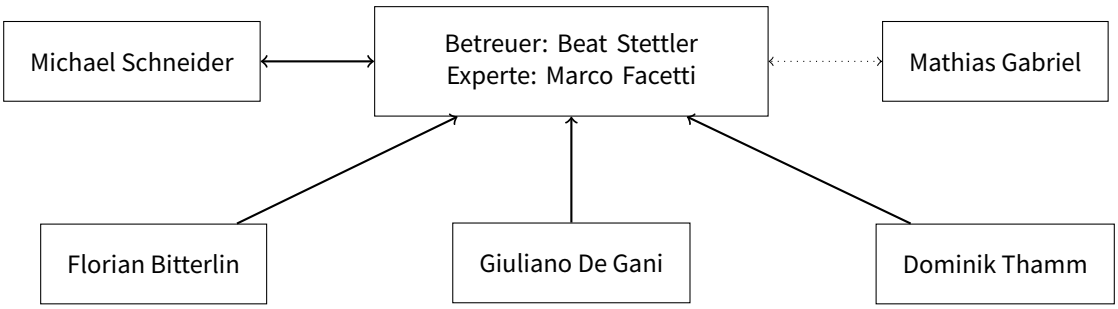


Abbildung 2.1.: Organigramm

Tabelle 2.1.: Rollen

Name	Rolle	E-Mail
Beat Stettler	Betreuer	beat.stettler@hsr.ch
Michael Schneider	Industriepartner (Cloudguard AG)	michael.schneider@cloudguard.ch
Marco Facetti	Experte	marco.facetti@trivadis.com
Mathias Gabriel	Assistent	mathias.gabriel@hsr.ch
Florian Bitterlin	Team Mitglied	florian.bitterlin@hsr.ch
Dominik Thamm	Team Mitglied / QS	dominik.thamm@hsr.ch
Giuliano De Gani	Team Mitglied / Ansprechpartner	giuliano.degani@hsr.ch

3. Zeitliche Planung

In diesem Kapitel werden die einzelnen Phasen und Meilensteine des Projekts beschrieben. Das Projekt dauert 17 Wochen. Die untenstehende Grafik gibt eine Übersicht.

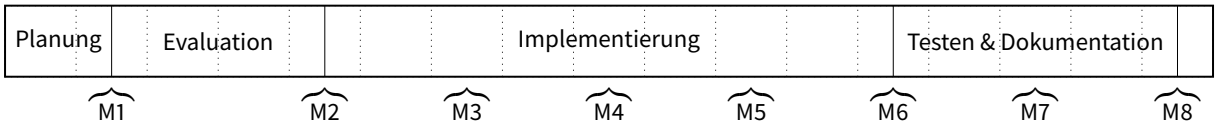


Abbildung 3.1.: Zeitplan

3.1. Phasen

Das Projekt besteht aus 4 Phasen. Diese sind in den folgenden Kapiteln genauer beschrieben.

3.1.1. Phase Projektplanung

In dieser Phase findet das Kick-off Meeting statt und es werden die Dokumente aufgesetzt. Der Zeitplan für das Projekt wird aufgestellt. Die Phase dauert 2 Woche.

3.1.2. Phase Konzeption und Evaluation

Diese Phase dauert 3 Wochen und beinhaltet die wesentlichen Tätigkeiten:

- Analyse der Aufgabenstellung
- Erstellen der Anforderungen und Softwarespezifikationsdokumente (Software Anforderungs Spezifikation, Architekturübersicht, Domainanalyse)
- Evaluation möglicher Architekturlösungen sowie IoT Geräte und Plattformen

3.1.3. Phase Implementation und Realisation

In dieser Phase wird der Software- und Hardware-Prototyp erstellt. Insgesamt umfasst die Phase 8 Wochen.

3.1.4. Phase Testen, Auswertung, Dokumentation

In diesen 2 Wochen wird:

- das Gesamtsystem getestet und die Leistung beurteilt.
- die Projekt- und Softwaredokumentation finalisiert.
- die Installationsanleitung geschrieben und die Präsentation vorbereitet.

3.2. Sprints

Die Phasen sind in Sprints aufgeteilt. Ein Sprint dauert 2 Wochen. Phasen mit ungeraden Wochen enden in einem einwöchigen Sprint.

Während der Sprintplanung, welche jeweils zu Beginn jedes Sprints durchgeführt wird, werden die Arbeitspakete für den Sprint festgelegt. Sprints beginnen jeweils am Mittwoch um 14:00 Uhr.

3.3. Meilensteine

Jeder Sprint endet mit einem Meilenstein. Diese sind unten beschrieben und aufgeführt.

3.3.1. M1: Projektplan

28.02.18 Der Projektplan ist aufgesetzt und die grobe Projektplanung abgeschlossen.

3.3.2. M2: Analyse IoT und Konzeption abgeschlossen

21.03.18 Die Analyse der Anforderungen und bestehenden Lösungen zu Sensoren, IoT Geräte und IoT Plattformen ist abgeschlossen. Die Lieferobjekte „Software Architektur“, Software Anforderungsspezifikation (SAS), Domainanalyse sind erzeugt und abgenommen.

3.3.3. M3: Infrastruktur, Entwicklerumgebung und IoT Geräte und Plattformen bereit

04.04.18 Sensoren, IoT Geräte und IoT Plattformen sind beschafft und können für die Entwicklung genutzt werden. Die benötigten Werkzeuge und Entwicklerumgebungen sind eingerichtet. Alle organisatorischen Hindernisse für die Entwicklung sind überwunden.

3.3.4. M4: Architekturprototyp

18.04.18 Ein lauffähiger Prototyp einer IoT Anwendung ist erstellt. Alle beteiligten Teilsysteme, vom Sensor, über das Gateway und die IoT Provider Komponente bis zur Webanwendung sind miteinander verbunden. Eine bidirektionale Kommunikation von und zu den Geräten ist vom Client möglich. Das heisst:

- Betriebsdaten können vom Sensor über das System im Client UI visualisiert werden.
- Der Client ist in der Lage, ein Command über das System zum Gerät abzusetzen.

3.3.5. M5: Abstraktion des IoT Providers

02.05.18 Die Einbindung des IoT Providers ist soweit abstrahiert, dass er ohne Eingriffe im Code Austauschbar ist. Ein Gerät kann über die Webanwendung kommissioniert, dekommissioniert und konfiguriert werden.

3.3.6. M6: Feature Freeze

23.05.18 Es werden keine weiteren Features oder Anforderungen umgesetzt. Die Entwicklungsarbeit beschränkt sich ab sofort auf das Bugfixing.

3.3.7. M7: System Integrationstests und Feldversuche

30.05.18 Das Gesamtsystem wurde hinsichtlich der Spezifikation getestet und bewertet. Feldversuche wurden durchgeführt und dokumentiert.

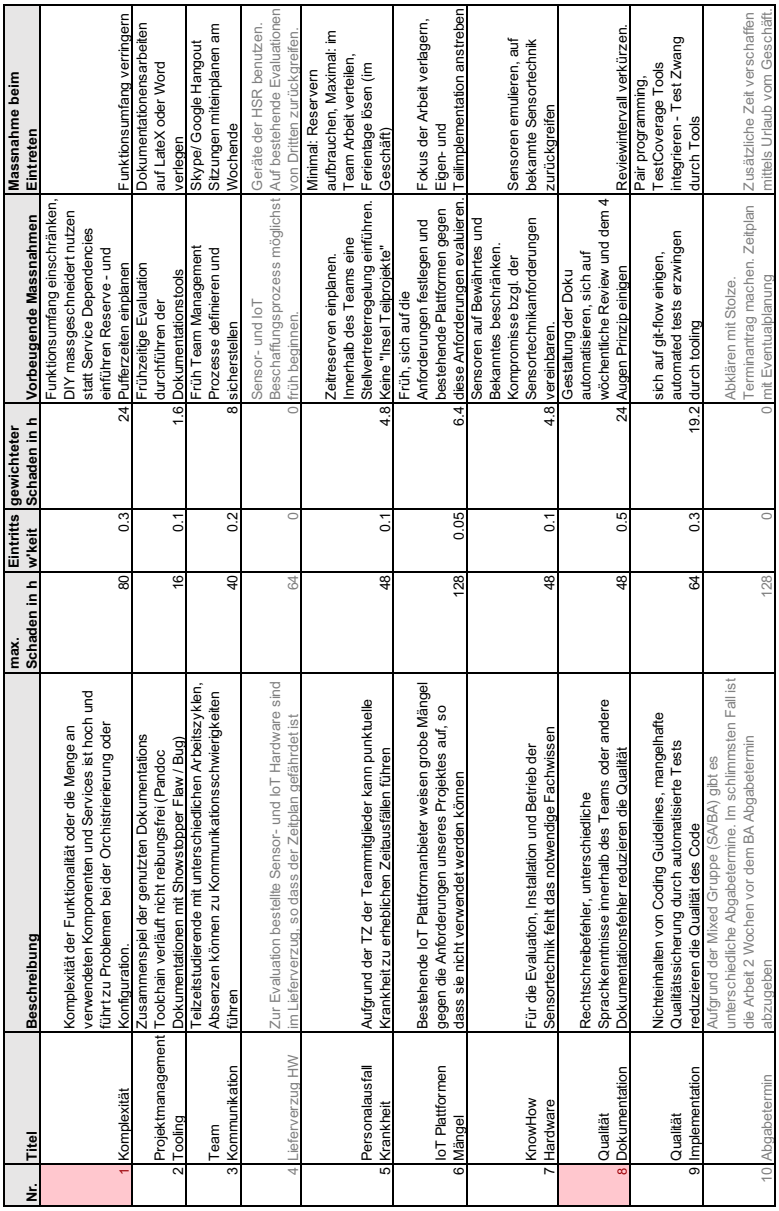
3.3.8. M8: Projektdokumente finalisiert, Arbeit abgegeben

13.06.18 Alle Dokumente sind korrigiert und formatiert. Die Arbeit wurde erfolgreich abgegeben. Die Präsentationsunterlagen sind vorbereitet.

3.4. Meetings

Meetings mit dem Betreuer finden wöchentlich jeweils am Mittwoch um 13 Uhr statt. Im Team wird mindestens Mittwochs und Donnerstags ein Meeting abgehalten um das weitere Vorgehen zu besprechen.

4. Risikomanagement



Nr.	Titel	Beschreibung	max. Schaden in h	Eintritts w'keit	gewichteter Schaden in h	Vorbeugende Massnahme	Massnahme beim Eintreten
1	Komplexität	Komplexität der Funktionalität oder die Menge an verwendeten Komponenten und Services ist hoch und führt zu Problemen bei der Orchistrierung oder Konfiguration.	80	0.3	24	Funktionsumfang einschränken, DIY massgeschneidert nutzen statt Service Dependencies einführen Reserve - und Pufferzeiten einplanen	Funktionsumfang verringern
2	Projektmanagement Tooling	Zusammenspiel der genutzten Dokumentations Toolchain verläuft nicht reibungsfrei (Pandoc Dokumentationen mit Showstopper Flaw / Bug)	16	0.1	1.6	Frühzeitige Evaluation durchführen der Dokumentationstools	Dokumentationsarbeiten auf LateX oder Word verlegen
3	Team Kommunikation	Teilzeitstudierende mit unterschiedlichen Arbeitszyklen, Absenzen können zu Kommunikationsschwierigkeiten führen	40	0.2	8	Früh Team Management Prozesse definieren und sicherstellen	Skype/ Google Hangout Sitzungen miteinplanen am Wochende
4	Lieferverzug HW	Zur Evaluation bestellte Sensor- und IoT Hardware sind im Lieferverzug, so dass der Zeitplan gefährdet ist	64	0	0	Sensor- und IoT Beschaffungsprozess möglichst früh beginnen.	Geräte der HSR benutzen. Auf bestehende Evaluationen von Dritten zurückgreifen.
5	Personalausfall Krankheit	Aufgrund der TZ der Teammitglieder kann punktuelle Krankheit zu erheblichen Zeitausfällen führen	48	0.1	4.8	Zeitreserven einplanen. Innerhalb des Teams eine Stellvertreterregelung einführen. Keine "Insel Teilprojekte"	Minimal: Reservern aufbrauchen, Maximal: im Team Arbeit verteilen, Ferientage lösen (im Geschäft)
6	IoT Plattformen Mängel	Bestehende IoT Plattformanbieter weisen grobe Mängel gegen die Anforderungen unseres Projektes auf, so dass sie nicht verwendet werden können	128	0.05	6.4	Früh, sich auf die Anforderungen festlegen und bestehende Plattformen gegen diese Anforderungen evaluieren.	Fokus der Arbeit verlagern, Eigen- und Teilimplementation anstreben
7	KnowHow Hardware	Für die Evaluation, Installation und Betrieb der Sensortechnik fehlt das notwendige Fachwissen	48	0.1	4.8	Sensoren auf Bewährtes und Bekanntes beschränken. Kompromisse bzgl. der Sensortechnikanforderungen vereinbaren.	Sensoren emulieren, auf bekannte Sensortechnik zurückgreifen
8	Qualität Dokumentation	Rechtschreibefehler, unterschiedliche Sprachkenntnisse innerhalb des Teams oder andere Dokumentationsfehler reduzieren die Qualität	48	0.5	24	Gestaltung der Doku automatisieren, sich auf wöchentliche Review und dem 4 Augen Prinzip einigen	Reviewintervall verkürzen.
9	Qualität Implementation	Nichteinhalten von Coding Guidelines, mangelhafte Qualitätssicherung durch automatisierte Tests reduzieren die Qualität des Code	64	0.3	19.2	sich auf git-flow einigen, automated tests erzwingen durch tooling	Pair programming, TestCoverage Tools integrieren - Test Zwang durch Tools
10	Abgabetermin	Aufgrund der Mixed Gruppe (SA/BA) gibt es unterschiedliche Abgabetermine. Im schlimmsten Fall ist die Arbeit 2 Wochen vor dem BA Abgabetermin abzugeben	128	0	0	Abklären mit Stolze. Terminantrag machen. Zeitplan mit Eventualplanung	Zusätzliche Zeit verschaffen mittels Urlaub vom Geschäft

Abbildung 4.1.: Risikomatrix

5. Infrastruktur

Alle Teammitglieder verwenden ihre privaten Computer sowie die von der HSR zur Verfügung gestellten Workstations im Arbeitszimmer an der Schule für die Arbeit am Projekt. Für die verwendeten Server-Tools wird die private Server-Infrastruktur von Florian Bitterlin zur Verfügung gestellt. Zusätzlich werden zwei virtuelle Maschinen für weitere Aufgaben benutzt.

5.1. Übersicht der Tools

Die verwendeten Tools sind in zwei Kategorien geteilt aufgeführt. Die Server-Tools werden im Multiuserbetrieb verwendet während die Entwicklungsumgebung lokal auf den Arbeitsmaschinen der Projektteilnehmer liegen.

5.1.1. Server-Tools

Folgende Tabelle listet die eingesetzten Tools auf. Zugangsdaten können bei den entsprechenden Administratoren angefragt werden.

Tabelle 5.1.: Server-Tools

Bezeichnung	Version	Beschreibung
GitLab CE / Git	10.*	Versionsverwaltungstool mit Weboberfläche für die Benutzerverwaltung und Codereviews. **Link:** https://git.bitterlin.net **Administrator:** Florian Bitterlin **Offizielle Webseite:** https://about.gitlab.com/ https://git-scm.com/

Bezeichnung	Version	Beschreibung
GitLab Runner	10.*	Build-Agent zur Verwendung mit GitLab. **Administrator:** Florian Bitterlin **Offizielle Webseite:** https://docs.gitlab.com/runner/
JFrog Artifactory	5.10.1	OpenSource-Variante des JFrog Artifactory Repository Managers. **Link:** https://repo.riot.bitterlin.net **Administrator:** Florian Bitterlin **Offizielle Webseite:** https://jfrog.com/open-source/#artifactory

5.1.2. Entwicklungsumgebung

Die Entwicklungsumgebung jedes Teammitglieds beinhaltet folgende lokal installierten Werkzeuge:

- **git**: Zur Versionsverwaltung wird Git verwendet. Es steht jedem Teammitglied frei dazu eine grafische Oberfläche zu verwenden.
- **Texteditor**: Zum Bearbeiten der Dokumentation mit integriertem Markdown-Linter und Spellchecking.
- **Pandoc**: Um die in der Dokumentation verfassten Markdown-Dateien in Latex umzuwandeln.
- **TeX-Umgebung**: Wird benötigt um den Build-Flow der Dokumentation abschliessen zu können.

Ein Teil der Arbeit sieht vor, zu evaluieren, inwiefern ein Projekt nach den Vorstellungen der SBB und ÖBB, mit den heutigen Mitteln umgesetzt werden kann. Es kann daher noch keine Aussage getroffen werden, welche weiteren Entwicklungs-Tools später zum Einsatz kommen werden.

5.2. Dokumentation

Dokumente werden in `Markdown`(Pandoc-Flavor) geschrieben. Damit sind sie versionierbar und mit jedem Texteditor lesbar. Mithilfe von `Pandoc` die `Markdown`-Files in ein `TeX`-Format umgewandelt. Aus dem resultierenden `TeX` wird mit Hilfe von `pdflatex` und `Biber`/`Biblatex` eine mit Verzeichnissen(Literatur, Inhalt, etc.) und Titelseite angereicherte und gestylte PDF-Datei generiert.

Mit Hilfe des Build-Agents von GitLab wurde das Generieren der Dokumente automatisiert. So werden stets alle Dokumente bei einem Commit ins Dokumentations-Repository[1] neu gebaut und stehen danach als Artefakte auf GitLab zum Download bereit.

5.3. Virtuelle Maschinen

5.3.1. HSR VM

Die VM der HSR wurde über den Server-Kiosk bestellt und daher nach dem gängigen Image für Studenten-Projekte erstellt. Sie läuft auf dem ESX-Cluster der HSR und hat ihr End of Life am 30.09.2018.

- **Betriebssystem**: Ubuntu Server 16.04 LTS
- **CPU**: 1 vCPU
- **RAM**: 2 GB

Die VM ist erreichbar über: sinv-56070.edu.hsr.ch

Backups

Sämtliche git-Repositories werden per `git pull` im Ordner `/home/root/backups` gesichert. Der `pull` erfolgt täglich um 02:00 Uhr und wird durch einen Cronjob ausgeführt.

5.3.2. innofield AG VM

Durch die Geschäftsbeziehungen von Dominik Thamm wurde uns von der innofield AG eine virtuelle Maschine in ihrem ALP1 Rechenzentrum[2] zur Verfügung gestellt. Mit ihrer mehr als ausreichenden Rechenleistung soll sie uns als Produktions-Server dienen und alle selbstentwickelten Services der Arbeit hosten.

- **Betriebssystem**: Ubuntu Server 16.04 LTS
- **CPU**: 19 GHz
- **RAM**: 6 GB

[1] https://git.bitterlin.net/ba/BA-SA-IG-Documentation
[2] https://innofield.com/blog/new-hosting-region-in-the-swiss-alps/

5.4. Kommunikation

Als Kommunikationsmittel sollen Whatsapp, E-Mail und Telefon verwendet werden. Die Reaktion der Teammitglieder soll innert 24 Stunden, ausgenommen ist der Sonntag, erfolgen.

6. Qualitätsmanagement

Der Qualitätsmanagementprozess ist ein dynamischer Prozess. Er kann fortlaufend angepasst werden.

Das Qualitätsmanagement umfasst dabei die Teilaspekte

- Identifikation von Qualitätsstandards basierend auf Anforderungen des Kunden oder der Projektbeteiligten.
- Planung der Qualitätsmassnahmen.
- Kontrolle, das heisst eine kontinuierliche Überwachung und Steuerung, der Qualitätsmassnahmen.

6.1. Identifikation

Für sämtliche Lieferobjekte gelten die allgemeinen Richtlinien und Qualitätsmerkmale für Projektarbeiten an der HSR.

Zusätzlich werden für folgende Lieferobjekte weitere Qualitätsmerkmale definiert. Das Qualitätsmanagement verfolgt das Ziel, diese Qualitätsmerkmale zu erfüllen und zu erhalten.

6.1.1. Evaluationsberichte

Evaluationsberichte umfassen minimal die folgenden Punkte:

- Beschaffungsprozess
- Technische Spezifikation / Fähigkeitenkatalog
- Vor- und Nachteile bzw. Stärken und Schwächen
- Anwendungsmöglichkeiten und Anwendungsgebiet im Rahmen eines IoT Projekts

6.1.2. Software-Anforderungsspezifikationen

- Ziele oder Anforderungen sind kategorisiert in MUSS / SOLL / KANN Ziele.
- Ziele sind messbar und lösungsneutral beschrieben.
- Anforderungen enthalten Abnahmekriterien.

6.1.3. Softwareprodukt

Sämtliche Software Erzeugnisse sind dokumentiert und getestet.

6.1.4. Projektdokumentation

Die Projektdokumentation umfasst alle Dokumente, welche dem Projektmanagementprozess zugeordnet werden können. Das beinhaltet unter Anderem Auswertungen bezüglich der Arbeitsaufwände, Zeiterfassungsdokumente, Arbeitspaketauflistungen, Statusberichte und Sitzungsprotokolle.

Diese Dokumente sind so zu gestalten, dass sie nachvollziehbar und transparent sind.

6.1.5. Prozesse

Prozesse helfen einer Projektgruppe, ihre Aufgaben zu strukturieren und zu planen sowie für die Kommunikation nach aussen.

Die Projektgruppe einigt sich darauf, agil zu arbeiten. Zu erledigende Aufgaben werden in Arbeitspakete gegliedert und mit einem vereinfachten Scrumban Prozess verwaltet. Die Projektgruppe verpflichtet sich, im wöchentlichen Rahmen den aktuellen Projektstatus dem Betreuer zu rapportieren.

6.2. Planung

Für die Erreichung und Erhaltung der definierten Qualitätsmerkmale werden folgende Massnahmen getroffen:

Nr.	Massnahme	Ziel	Zeitraum
1	Review der Evaluationsberichte und Spezifikationen	Sicherstellen, dass die Merkmale für Evaluationsberichte und Spezifikationen erfüllt sind	Nach Abschluss des Issues
2	git-flow und Code Review	Einhaltung der Coding Guidelines, Erzwingen der Code Dokumentation, Sicherstellen der Lesbar- und Wartbarkeit	Beim Pull Request

Nr.	Massnahme	Ziel	Zeitraum
3	Automatisierte Tests, Continous Integration	Erhöhen der Code Qualität durch schnelleres Aufdecken von Fehlern. Erhöhung der Geschwindigkeit, in der Änderungen gemacht werden können durch angstbefreites Refactoring	Fortlaufend
4	Wöchtentliche Sitzung mit Betreuer	Überwinden von administrativen Hindernissen und logistischen Bedürfnissen, Machbarkeit und Gelingen des Projektes sicherstellen durch Eingrenzen des Projektumfangs, Priorisierung und Ausrichtung	Einmal pro Woche
5	Automatisierte Sicherung sämtlicher Dokumente und Lieferobjekte	Sicherstellen, dass keine Arbeit verloren geht	täglich
6	Dokumentation mit Versionierungssystem git verwaltet	Sicherstellen der Transparenz und Nachvollziehbarkeit sämtlicher Dokumente	Fortlaufend

6.3. Kontrolle

Der Qualitätsverantwortliche überprüft mindestens einmal pro Iteration die Einhaltung der getroffenen Massnahmen und deren Effektivität und passt diese bei Bedarf an. Die Überprüfung und Beurteilung erfolgt mündlich und in Absprache mit dem Projektteam.

Teil II.

Bericht

7. Ausgangslage

7.1. Produktperspektive

Die IoT Plattform ist eine eigenständige Anwendung für das Verarbeiten und Verwalten von Betriebsdaten, welche von zahlreichen Sensoren oder Sensorverbunden über das öffentliche Mobilfunknetz an die Plattform übermittelt werden. Betriebsdaten beschreiben den Zustand von logistischen Leistungserbringern, wie zum Beispiel Lieferfahrzeuge, Güterwagen oder Maschinen. Mögliche Betriebsdaten sind Positionsdaten oder Betriebsstunden, Treibstoffverbrauch, Öltemperatur usw. Diese Betriebsdaten können in Echtzeit visualisiert werden oder zur Analyse weiterverarbeitet werden. Diverse Benutzergruppen können via Webanwendung auf die Plattform zugreifen und auf die Betriebsdaten zugreifen um damit den Status zu überprüfen.

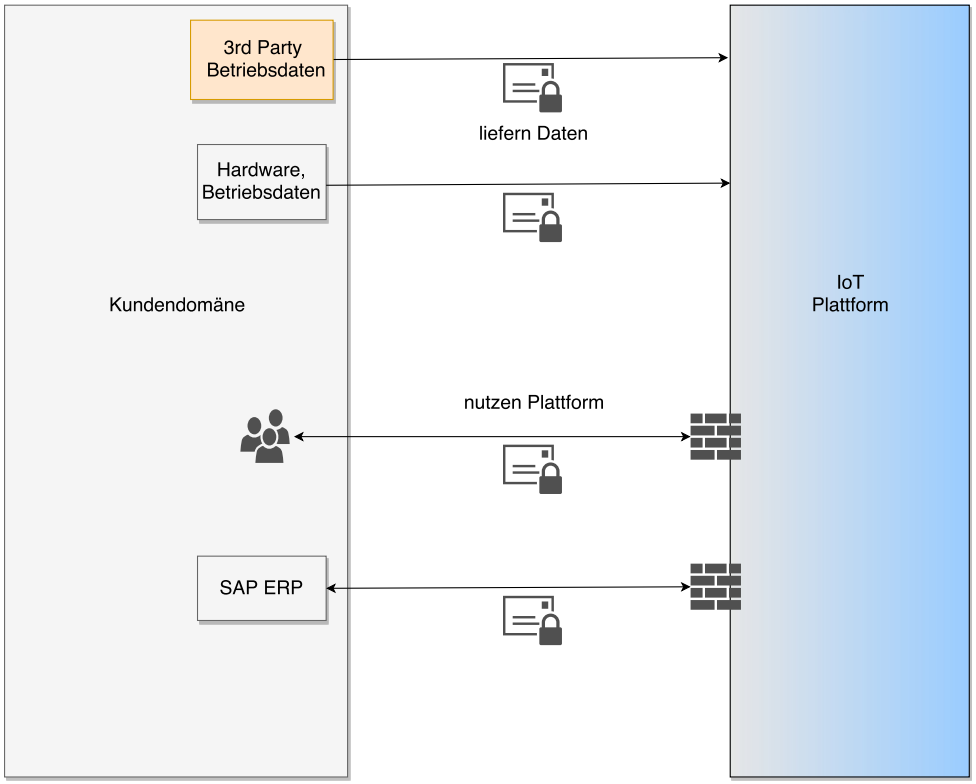


Abbildung 7.1.: Systemkontext

7.2. Produktfunktion

Die Funktionalität wird in diesem Dokument durch die technischen Anforderungen (Use Cases) detailliert beschrieben.

7.3. Benutzermerkmale

Zur Zielgruppe der IoT Plattform gehören primär Leistungserbringer in der Logistikbranche, zum Beispiel Bahn- und Transportunternehmen sowie Industriezweige mit verteilten Maschinen, welche es zu überwachen gilt.

7.4. Annahmen

- Für den Einsatzraum existiert eine Internetfähige Mobilfunknetz Infrastruktur, welche betriebsbereit ist und genutzt werden kann.
- Für den Betrieb der IoT Geräte ist eine lokale Stromversorgung verfügbar oder die Geräte können mit einer autonomen Batterie für einen Zeitraum von mindestens einem Jahr betrieben werden.
- Die Iot Geräte werden nicht over-the-air (OTA) aktualisiert.

8. Spezifikation

8.1. Funktionale Anforderungen

8.1.1. Use Case Diagramm

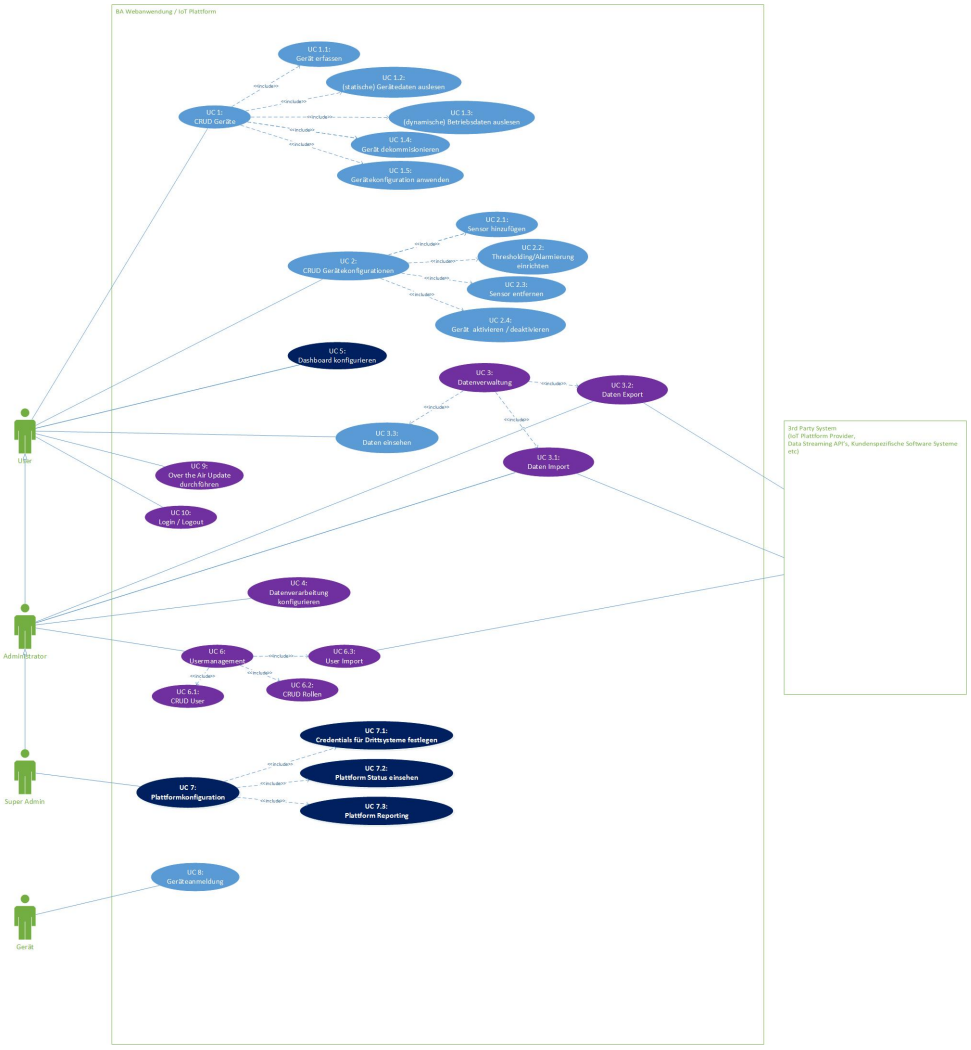


Abbildung 8.1.: Usecase Diagramm

Das Use Case Diagramm definiert die folgenden Aktoren:

Tabelle 8.1.: Aktoren im Use Case Diagramm

Aktor	Beschreibung
User	Alle Anwender der Plattform sind User. Dieser ist in der Lage, den gesamten Life Cycle von Geräten zu verwalten, sowie Daten zu exportieren und Funktionalitäten von Drittanbietern zu nutzen, wie zum Beispiel Datenanalysefunktionen.
Administrator	Ist zusätzlich in der Lage, User und deren Rollen zu verwalten sowie das Importieren von Daten von Drittanbietern, wie z.B. Userprofile, bestehende Gerätelisten oder Ähnliches.
Super Administrator	Ist zusätzlich in der Lage Plattform Diagnose Möglichkeiten zu nutzen und für die Inbetriebnahme relevante Einstellungen zu tätigen. Dies beinhaltet zum Beispiel das Einsehen des Plattformstatus oder Vergeben von Berechtigungen für das Anbinden an Drittsysteme.
Gerät	Diese Gruppe von Aktoren beschreibt die IoT Devices, welche sich am System anmelden können.

8.1.2. Beschreibungen

In den nachfolgenden Unterkapiteln werden alle Use Cases im brief-Format beschrieben.

UC 1 CRUD Geräte

Der Use Case „CRUD Geräte“ umfasst die Operationen für das Verwalten der Geräte. Der Begriff Gerät beschreibt sowohl Gateways als auch Sensoren. Diese Vereinfachung wird gemacht, weil der Funktionsumfang der Geräte über die Gerätekonfiguration abgebildet ist.

UC 1.1 Gerät erfassen

Ein Aktor kann ein Gerät für die Verwendung in Gerätekonfigurationen zur Verfügung stellen. Dieser Schritt ist notwendig, um der Plattform das Gerät bekannt zu machen. Das Gerät kann sich im Anschluss per Geräteanmeldung (UC 8) bei der IoT Plattform erfolgreich anmelden.

UC 1.2 (statische) Gerätedaten auslesen

Der Aktor kann die gerätespezifischen Metadaten auslesen, wie z.B. verwendetes Betriebsystem, Softwareversionen und Anschlussmöglichkeiten des Gerätes.

UC 1.3 (dynamische) Betriebsdaten auslesen

Auslesen der Sensordaten. Der Aktor kann den aktuellen Status auslesen. Das heisst, das Gerät stellt alle verfügbaren Daten der Sensoren dem Aktor zur Verfügung.

UC 1.4 Gerät dekommisionieren

Der Aktor kann ein Gerät vom System abmelden. Das Gerät ist dann nicht mehr von der Plattform erfasst und es wird kein Gerätestatus mehr erfasst.

UC 1.5 Gerätekonfiguration anwenden

Die Fähigkeiten eines Gerätes werden über dessen Konfiguration bestimmt. Der Aktor kann ein Gerät mit einer Konfiguration laden, so dass zum Beispiel neu angeschlossene Sensoren verfügbar gemacht und ausgelesen werden können oder Timeout / Threshold Werte angepasst werden können.

UC 2 CRUD Gerätekonfiguration

Der Use Case „CRUD Gerätekonfiguration“ umfasst alle Operationen für das Verwalten der Gerätekonfigurationen. Eine Gerätekonfiguration kann erstellt, geändert, ausgelesen und gelöscht werden. Mit Hilfe dieser Konfigurationen werden Geräte bzw. deren Fähigkeiten und Ausstattungen konfiguriert.

UC 2.1 Sensor hinzufügen

Der Aktor kann Sensoren in die Konfiguration hinzufügen, damit diese vom Gerät ausgelesen werden können.

UC 2.2 Alarmierung einrichten

Der Aktor kann eine Alarmierung mittels Schwellwert für konfigurierte Sensoren einrichten. Dabei wird für gewählte Sensordaten ein Schwellwert gesetzt, bei welchem das Gerät eine automatische Benachrichtigung an die konfigurierten Plattformen absendet.

UC 2.3 Sensor entfernen

Der Aktor kann Sensoren aus der Konfiguration entfernen. Entfernte Sensoren sind dem Gerät nicht mehr bekannt.

UC 2.4 Gerät aktivieren und deaktivieren

Ein Aktor kann konfigurierte Sensoren oder Geräte aus- oder einschalten. Die Einstellungen bleiben erhalten, aber die Betriebsdaten des Gerätes werden vom System nicht mehr verarbeitet.

UC 3 Datenverwaltung

UC 3.1 Daten Import

Dieser Use Case beinhaltet Operationen wie das Importieren von bestehenden Gerätekonfigurationen, Geräteinformationen von Herstellern.

UC 3.2 Daten Export

Der Aktor kann seine aufgezeichneten Daten in bestehende Drittsysteme exportieren. Die Daten werden dazu zum Beispiel als *.csv Dokument formatiert und abgelegt. Die Formatmöglichkeiten sind mit dem Kunden oder Drittanbieter abzuklären und zu implementieren.

UC 3.3 Daten einsehen

Der Aktor kann aufgezeichnete oder aktuelle Daten einsehen.

UC 4 Datenverarbeitung konfigurieren

Der Aktor kann zusätzliche Formen der Aufbereitung der Daten wie das Sortieren, Filtern oder Weiterverarbeitung konfigurieren.

UC 5 Dashboard konfigurieren

Jeder Benutzer kann die im Dashboard enthaltenen Elemente bezüglich ihrer Sichtbarkeit und Position konfigurieren. Diese Einstellungen sind pro Anwender als Konfiguration hinterlegt (Mandantenfähigkeit).

UC 6 Usermanagement

Umfasst sämtliche administrative Operationen zur Verwaltung der User und deren Rollen.

UC 6.1 CRUD User

Der „CRUD User“ Use Case beschreibt das Erstellen, Auslesen, Ändern und Löschen von Benutzern.

UC 6.2 CRUD Rollen

Rollen werden im System dazu verwendet, um User mit Berechtigungen auszustatten. Der „CRUD Rollen“ Use Case beschreibt das Erstellen, Auslesen, Ändern und Löschen von Rollen.

UC 6.3 User Import

Der Aktor kann bestehende User oder Rollen über bestehende Drittsysteme (z.B. eine LDAP Anbindung) importieren.

UC 7 Plattformkonfiguration

Der Use Case „Plattformkonfiguration“ beinhaltet die Aufgaben, die in der Regel der Inbetriebnehmer der Plattform erledigt.

Er umfasst die Use Cases:

UC 7.1 Credentials für Drittsysteme festlegen

Der Aktor kann die Verbindungen zu Drittsystemen festlegen. Dazu gehören zum Beispiel das Einrichten der API Public Keys für Drittsysteme oder andere Schnittstellenkonfigurationen.

UC 7.2 Plattform Status einsehen

Der Aktor kann den Plattformstatus einlesen. Er hat Zugang zu sämtlichen diagnostischen Möglichkeiten der Plattform.

UC 7.3 Plattform Reporting

Der Aktor kann den Plattformstatus verarbeiten, visualisieren und exportieren.

UC 8 Geräteanmeldung

Der Aktor Gerät kann sich am System anmelden (Handshake). In diesem Vorgang erhält das Gerät sämtliche Softwareupdates und Zertifikate, welche für den Betrieb und die Kommunikation mit der Plattform notwendig sind. Diese Anmeldung ist nur erfolgreich, wenn vorgängig das Gerät vom User erfasst wurde, d.h. das Gerät im System bereits eingetragen und aktiviert wurde.

UC 9 Over the Air Update durchführen

Der Aktor kann für eine Auswahl von Geräten ein Softwareupdate OTA durchführen, d.h. ohne physikalischen Zugriff auf das Gerät.

UC 10 Login / Logout

Der Aktor kann sich an der Plattform an- und abmelden.

8.1.3. Priorisierung der funktionalen Anforderungen

Allen Use Cases sind in die Prioritätsstufen 1,2,3 zugeordnet, wobei 1 die höchste und 3 die niedrigste Priorität beschreibt. Diese Priorisierung legt den Grundstein für die Zeit- und Meilensteinplanung.

Tabelle 8.2.: Priorisierung der funktionalen Anforderungen

Anforderung	Priorisierung
UC 1.1 Gerät erfassen	1
UC 1.2 (statische) Gerätedaten auslesen	1
UC 1.3 (dynamische) Betriebsdaten auslesen	1
UC 1.4 Gerät dekommisionieren	1
UC 1.5 Gerätekonfiguration anwenden	1
UC 2.1 Sensor hinzufügen	1
UC 2.2 Alarmierung einrichten	1
UC 2.3 Sensor entfernen	1
UC 2.4 Gerät aktivieren und deaktivieren	1
UC 3.1 Daten Import	2
UC 3.2 Daten Export	2
UC 3.3 Daten einsehen	1
UC 4 Datenverarbeitung konfigurieren	2
UC 5 Dashboard konfigurieren	3
UC 6.1 CRUD User	2

Anforderung	Priorisierung
UC 6.2 CRUD Rollen	2
UC 6.3 User Import	3
UC 7.1 Credentials für Drittsysteme festlegen	3
UC 7.2 Plattform Status einsehen	3
UC 7.3 Plattform Reporting	3
UC 8 Geräteanmeldung	1
UC 9 Over the Air Update durchführen	2
UC 10 Login / Logout	2

8.1.4. Business Logic Layer

Stream processing

Die von den Sensoren kommenden Daten werden in der Plattform verarbeitet. Es werden drei verschiedene Verarbeitungen unterschieden.

Konvertierung

Die Konvertierung bringt die Daten in die gewünschte Form. Dabei können Daten gefiltert, sortiert und komprimiert werden sowie auch Mittelwerte ausgerechnet und mit Systemdaten angereichert werden. Dieser Schritt kann für verschiedene Daten unterschiedlich konfiguriert werden.

Analyse / Alarme

In der Analysephase werden die Daten auf definierte Regeln geprüft. Diese Regeln können sich auf konkrete Werte oder Mittel- und Prozentwerte beziehen. In der Konfiguration wird festgelegt, über welchen Zeitraum die Daten analysiert werden. Beispiele:

- Sensorwert X ist über 40.
- Der Wert Y ist in den letzten 5 Minuten um 10% gestiegen.
- 5 Geräte haben den Status Z

Wir unterscheiden zwei Arten von Alarmen:

Alarm auf dem Gerät: Werden vom Gateway ausgelöst, sobald ein bestimmter Wert eines Sensors überschritten wird. Die genauen Werte können in der Konfiguration pro Sensor festgelegt werden. Die Überprüfung dieser Werte ist Aufgabe des Gateways.

Alarm auf der Plattform: Trifft eine der oben beschriebenen Regeln zu, wird von der Plattform selbst ein Alarm generiert. Die Plattform übernimmt hier die Überprüfung der Werte, da sich diese Regeln auf mehrere Geräte beziehen können.

Speicherung

Bei diesem Schritt wird festgelegt, welche Daten wo und in welcher Art gespeichert werden. Es können z.B nur die konvertierten Daten gespeichert und die Rohdaten verworfen werden. Der Speicherort kann dynamisch festgelegt (z.B Datenbank, Filesystem, Object storage etc.) werden. Weiter soll die Möglichkeit bestehen, die Speicherung zeitlich und mengenmässig zu begrenzen. Zum Beispiel sollen immer die letzten zwei Stunden oder die letzten 10000 Einträge vollständig gespeichert werden und danach nur noch die komprimierten Daten.

Gerätegruppen

Geräte können für die erleichterte Verarbeitung der Daten in Gruppen zusammengefasst werden. Die Gerätegruppen können zur Laufzeit angepasst werden.

Für Datenverarbeitungen auf Gruppenebene werden nur Daten berücksichtigt, wenn das Gerät zu diesem Zeitpunkt eine Gruppenzugehörigkeit aufweist.

8.1.5. Benutzer Management

Die Plattform beinhaltet ein Benutzermanagement welches die Authentisierung und Autorisierung von Benutzern übernimmt. Die einzelnen Teile der Verwaltung werden in diesem Kapitel genauer beschrieben.

Gruppen

Gruppen dienen der Abgrenzung von Geräten und Benutzern (z.B. unter verschiedenen Abteilungen einer Firma). Alle Benutzer der Gruppe teilen sich die gleichen Geräte, Konfigurationen und Sensoren.

Benutzer

Benutzer sind die Aktoren des Systems. Sie authentifizieren sich mit Benutzername und Passwort. Jeder Benutzer gehört zu mindestens einer Gruppe. Einem Benutzer können verschiedene Rollen zugeteilt sein.

Rollen

Rollen haben ein oder mehrere Berechtigungen und können Benutzern zugeteilt werden. Ein Benutzer kann mehrere Rollen haben.

Berechtigungen

Berechtigungen regeln den Zugriff auf die einzelnen Teile des Systems. Es gibt zwei Arten von Berechtigungen: lesen und schreiben. Die Berechtigungen sind vom System vorgegeben und können von Benutzern nicht bearbeitet werden. Beispiele für Berechtigungen sind: - Geräte ansehen (lesen) - Sensoren auflisten (lesen) - Konfigurationen bearbeiten (schreiben) - Rollen bearbeiten (schreiben)

8.1.6. Data API

Die Data API bietet eine Pattforminterne Schnittstelle zur Speicherung von Sensordaten an. Diese API existiert, um die Persistierung von Sensordaten auch bei Dritt-Anbietern zu ermöglichen und um die darunterliegende Speichertechnologie austauschbar zu halten.

Datenverwaltung

Die Data API bietet die Möglichkeit Daten zu erfassen und persistierte Daten zu lesen, bearbeiten und zu löschen.

Strukturen für berechnete Daten, welche ebenfalls persistiert werden sollen, können über die Data API in der darunterliegenden Speichertechnologie erstellt, bearbeitet und gelöscht werden.

Ordnung und Behandlung von Rohdaten

Es gibt eine Unterscheidung zwischen Rohdaten und berechneten Daten. Rohdaten bezeichnen Sensordaten, welche direkt vom Geräte geliefert werden. Berechnete Daten sind Sensordaten, welche mittels Korrelationen oder weiteren Verarbeitungsschritten berechnet wurden. Als Rohdaten ausgewiesene

Daten sind über die Data API nicht bearbeitbar, um sie für einen möglichen Export unberührt zu belassen. Strukturen für berechnete Daten können wenn nötig so deklariert werden, dass darin gespeicherte Daten bearbeitbar bleiben.

Sensorrohdaten sollen nach Gerät und Sensor geordnet abgelegt werden können. Weitere Attribute der Geräte, zum Beispiel die Zugehörigkeit des Geräts zu einer Gruppe, haben während der Erfassung eines neuen Sensorwerts keinen Einfluss auf deren Speicherort.

8.1.7. Client API

Die Client API ist eine einheitliche Schnittstelle, über welche die Benutzerapplikationen mit der Plattform kommunizieren. Mögliche Clients können eine Webanwendung, mobile Apps oder eine Desktopanwendung sein.

Die API deckt alle Aktionen ab, welche ein Benutzer auf der Plattform ausführen können muss. Durch die Authentifizierung der Benutzer und ihre Zuweisung in Rollen werden unberechtigte Zugriffe und Aktionen verhindert.

Geräteverwaltung

Die API bietet die Möglichkeit, Geräte zu verwalten. Neben der Möglichkeit, den aktuellen Gerätestatus und statische Informationen auszulesen, können auch neue Geräte erfasst und alte dekommissioniert werden. Diese Aktionen werden in Teilschritte unterteilt, dem Registrieren, dem Aktivieren bzw. Deaktivieren und dem Löschen.

Geräte können zur besseren Übersicht in Gruppen eingeteilt werden. Die Client API bietet an, Gerätegruppen zu Erstellen, Editieren und zu Löschen und Geräte den Gerätegruppen zuzuweisen.

Gerätekonfiguration

Geräte können über die API konfiguriert werden. Dies beinhaltet:

- Sensoren von Geräten zu aktivieren oder zu deaktivieren.
- Konfigurieren von Sendeintervall oder Alarmierung bei Überschreitung von Grenzwerten.

Gerätekonfigurationen können auf einzelne Geräte oder auf Gerätegruppen angewendet werden.

Datenverwaltung

Daten Import

Geräte können über einen Bulk-Import von Seriennummern bei der Plattform registriert werden. Gerätekonfigurationen können in einem vordefinierten Format in die Plattform als Konfigurationstemplate eingefügt werden.

Daten Export

Die Plattform bietet die Möglichkeit, aufgezeichnete Sensordaten über einen zeitlichen Rahmen zu exportieren. Über die Client API können Exporte angestossen werden, so dass sie von der Plattform für den späteren Bezug bereitgestellt werden können.

Dateneinsicht

Durch die Client API können aktuelle oder historische Sensordaten für die grafische Aufbereitung auf der Clientplattform bezogen werden. Die API bietet dabei eine zeitliche Filterung sowie eine Filterung nach bestimmten Werteeigenschaften an.

Datenverarbeitung konfigurieren

Über die Client API kann die Verarbeitung der Sensordaten konfiguriert werden. Dies beinhaltet unter Anderem die Korrelation mit historischen Daten oder die Weiterverarbeitung auf Drittsystemen.

Dashboard konfigurieren

Ein Benutzer kann sein Dashboard individualisieren, sowie seine Einstiegsseite nach dem Login konfigurieren. Das Dashboard kann Geräteinformationen in kondensierter Form visualisieren und einen Überblick über ausgelöste Alarme verschaffen.

Usermanagement

User

Benutzer mit der entsprechenden Berechtigung können über die Client API anderen Benutzern den Zugriff gewähren oder entziehen sowie Daten anderer Benutzer anzupassen.

Rollen

Benutzer mit der entsprechenden Berechtigung können über die Client API neue Rollen definieren und diese an Benutzer zuweisen.

Import

Das Einbinden von Drittsystemen zur Benutzerverwaltung kann über die Client API konfiguriert werden.

Plattformkonfiguration

Credentials für Drittsysteme festlegen

Anmeldeinformationen oder andere Schnittstellenkonfigurationen für Drittsysteme, welche direkt von der Pattform benutzt werden, können über die Client API konfiguriert werden.

Plattform Status einsehen

Die Client API bietet die Möglichkeit, den Status (zum Beispiel die CPU-Auslastung, Arbeitsspeicherbelegung oder Sensordatendurchsatz) einzusehen.

Plattform Reporting

Das automatische Verhalten bei Alarmen, z.B. bei einer zu hohen Auslastung, oder anderes Statusveränderungen der Plattform kann über die Client API konfiguriert werden.

8.1.8. IoT Provider

Die IoT Provider Komponente kümmert sich um die komplette Verwaltung, Authentifikation und Kommunikation von und zwischen den Geräten. Dazu gehört auch das Statemanagement mittels Konfiguration.

Message Broker

Die Kommunikation zwischen Geräten basiert auf dem MQTT Protokoll welches ein publish/subscribe pattern implementiert. Der Message Broker dient als Anlaufstelle für alle Subscriber und Publisher. Clients können beliebige Topics abonnieren. Empfangene Nachrichten eines bestimmten Topics werden vom Broker an alle Abonnenten weitergeleitet, sobald eine Verbindung besteht. Der Broker speichert die Nachrichten solange, bis sie an alle ausgeliefert wurde. Falls ein Topic keine Abonnenten hat, wird die Nachricht verworfen.

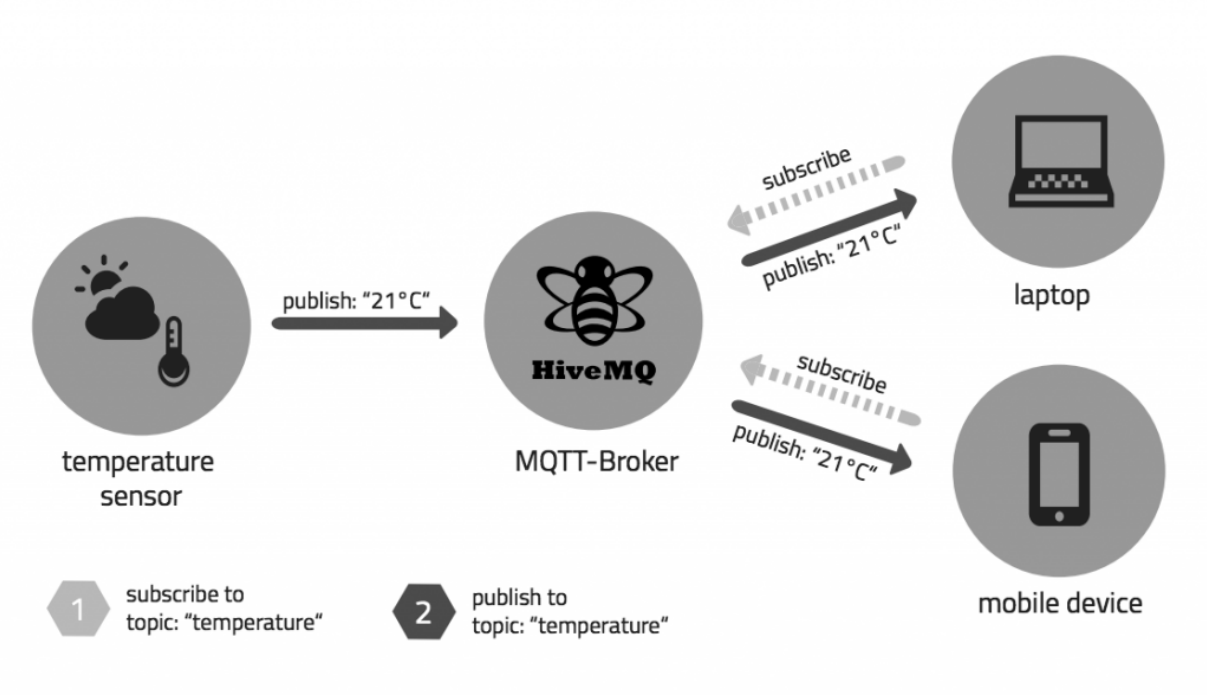


Abbildung 8.2.: Beispiel eines MQTT Brokers (hier HiveMQ)[7]

Device Registry

Alle Geräte, welche mit dem Provider kommunizieren wollen, werden von der Device Registry identifiziert und autorisiert. Die Registry kümmert sich um das Ausstellen und Verwalten der Gerätezertifikate mittels des hinterlegten CA Zertifikats. Wird ein neues Gerät hinzugefügt, wird ihm eine eindeutige ID, Key und Clientzertifikat ausgestellt. Diese Informationen werden zur Authentifizierung bei der Kommunikation verwendet. Bei Ausserbetriebnahme eines Gerätes werden die dazugehörigen Zertifikate gesperrt und somit die Kommunikation verhindert.

Eine simplere Variante wäre die Authentifikation mittels API_KEY ohne Zertifikat. Diese Umsetzung ist einfacher, jedoch weniger sicher.

Device Management

Das Device Management übernimmt die Verwaltung und Konfiguration der Geräte und deren Software. Für jedes Gerät wird der letzte gemeldete Zustand gespeichert, um allfällige Konfigurationsänderungen vornehmen zu können, auch wenn gerade keine Kommunikation möglich ist. Sobald die Kommunikation wieder möglich ist, wird die neue Konfiguration an das Gerät geschickt. Geräte besitzen zudem Metadaten, welche beim Erstellen gesetzt werden können. Metadaten sind Werte, die normalerweise

statisch sind und nicht (häufig) ändern, wie zum Beispiel Herstellerinformationen, Baujahr oder Seriennummer. Geräte können einer Kategorie/Typ zugewiesen und in Gruppen zusammengefasst werden. Gruppen können aus Geräten und/oder Untergruppen bestehen.

Schnittstelle

Der IoT Provider bietet eine Schnittstelle, über welche es möglich ist, die Funktionalitäten in andere Applikationen zu integrieren. Die geforderten Funktionen sind identisch mit der Provider API Abstraction welche im Kapitel Provider API beschrieben sind.

8.1.9. Gateway

Als Gateway wird jede Hardware benannt, welche direkt mit dem IoT Provider kommuniziert, um Sensordaten zu übermitteln.

Hardware

Die Hardware der Gateways muss

- mit dem IoT Provider kabellos und ortsunabhängig kommunizieren können

Die Hardware der Gateways kann

- Sensoren oder zusätzliche Hardware verbaut haben
- als Router für die Kommunikation der angeschlossenen Sensoren oder Gateways untereinander operieren
- Batterie- oder mit externer Stromversorgung betrieben werden

Weitere Vorgaben an die Hardware der Gateways sind nicht vorgegeben, weil der Fokus der Projektarbeit nicht auf der Hardwareentwicklung oder der Technologie der verbauten Sensoren liegt.

Die Bezeichnung „Gerät“ kann daher innerhalb der Dokumentation dieses Projekts als Synonym für Gateway verwendet werden.

Kommunikation

Gateways sind fähig über eine TCP/IP-Verbindung mit dem IoT Provider zu kommunizieren. Ob dies über ein vorhandenes Wifi-Netzwerk bei stationärem Einsatz oder über das UMTS-Netz bei mobilen Einsätzen geschieht, muss je nach Einsatzort definiert und die Gateways entsprechend ausgestattet werden.

Das Gateway beherrscht sowohl die Kommunikation über MQTT sowie HTTP. MQTT kommt bei der Kommunikation mit dem IoT Provider zum Einsatz, um Sensordaten zu Übermitteln und Konfigurationsänderungen anzustossen. Die Authentifizierung gegenüber dem IoT Provider erfolgt mittels Clientzertifikaten, welche die Gateways während der Geräteerfassung im System erhalten. HTTP wird genutzt um Erfassungsanfragen abzuhandeln und Firmwareupdates herunterzuladen. Jegliche Kommunikation ist stets per TLS verschlüsselt, unabhängig vom genutzten höheren Protokoll.

Software

Die auf dem Gerät ausgeführte Software muss in der Lage sein, angeschlossene Sensoren auf Basis der zugewiesenen Konfiguration anzusprechen und auszulesen. Zudem muss das Gateway in der Lage sein, Sensordaten, Statusinformationen und Änderungen an der Konfiguration mittels den oben spezifizierten Protokollen an den Server zu senden. Beim Eintreten von in der Konfiguration definierten Grenzerten pro Sensor, soll das Gerät die Plattform darüber informieren.

8.1.10. Provider API

Diese Schnittstelle stellt ein einheitliches Interface zum IoT Provider für die Plattform zur Verfügung. In diesem Kapitel sind die Anforderungen an eine solche Schnittstelle definiert.

Administration

Login Credentials der Plattform setzen

Damit die Plattform mit dem Provider kommunizieren kann müssen die Zugangsdaten richtig gesetzt werden.

Status des Providers abfragen

Für das Monitoring soll es möglich sein, Informationen zur Erreichbarkeit abzufragen.

Eigene CA eintragen

Um die Verwendung eigener Zertifikate zu ermöglichen, muss es möglich sein, die eigene CA beim Provider einzutragen.

Geräteverwaltung

Neues Gerät erstellen

Es muss möglich sein, über die Schnittstelle neue Geräte zu erstellen (registrieren). Jedem Gerät soll ein Typ zugeordnet werden können.

Metadaten der Geräte verwalten

Für die einzelnen Geräte muss es möglich sein, Metadaten wie Seriennummer, Model, MAC-Adresse etc. pro Gerät zu ändern.

Gerätetypen verwalten

Es soll möglich sein, neue Gerätetypen zu erstellen sowie bestehende zu verwalten.

Gerätegruppen verwalten

Geräte sollen in Gruppen eingeteilt werden können. Gruppen können aus Geräten und/oder weiteren Gruppen bestehen. Es sollen Gruppen erstellt, verändert und gelöscht werden können.

Konfigurationsvorlagen verwalten

Zur einfachen Konfiguration soll es möglich sein, Konfigurationsvorlagen für Geräte anzulegen und anzupassen. Diese können dann auf Geräte oder Gruppen angewendet werden.

Liste aller Geräte abfragen

Es muss möglich sein, eine Liste aller registrierten Geräte abzufragen und zu filtern/sortieren zum Beispiel nach Gerätetyp.

Status von Gerät abfragen

Die Schnittstelle soll den zuletzt gemeldeten Status des Geräts liefern. Dies beinhaltet Daten wie den Batteriestand und Betriebstunden.

Konfiguration für Gerät ändern

Einem Gerät muss eine zuvor erstellte Konfiguration zugewiesen werden können.

Nachricht an Gerät senden

Es muss möglich sein eine Nachricht an ein bestimmtes Gerät zu senden.

Nachricht an Topic senden

Nachrichten sollen per Schnittstelle auf beliebigen Topics publiziert werden können.

Geräte entfernen

Geräte müssen gelöscht werden können. Dabei soll auch gleich dessen Zertifikat und Zugriff auf das System gelöscht werden.

Datenverwaltung

Topics abonnieren

Es muss möglich sein, ein beliebiges Topic zu abonnieren. Somit erhält man jede auf diesem Topic publizierte Nachricht.

Abonnement entfernen

Das Abonnement auf ein Topic muss wieder entfernt werden können.

8.2. Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen sind spezifische, messbare Qualitätsmerkmale, die grossen Einfluss auf die Architektur- und Technologieentscheide haben.

Die nachfolgende Analyse der nicht funktionalen Anforderungen basiert auf dem „(F)URPS+ (Usability, Reliability, Performance, Supportability plus weitere)“ Modell.

8.2.1. Funktionalität (functionality)

Begriffe

- **Auditability (Protokollierung)**: Protokollierung wichtiger, interner Daten zur Systemausführung.
- **Error Management (Fehlerbehandlung)**: Erkennung, Weiterleitung und Speicherung von Systemfehlern.
- **Safety (Betriebssicherheit)**: Sicherheit aus Anwendersicht (bspw. Authentifizierung).

Tabelle 8.3.: NFRs - Funktionalität

Zuordnung	Anforderung
Auditability, Error Management	Zur Fehlerbehandlung und Nachvollziehbarkeit sind die System und Fehlermeldungen in einer Log Datei persistent abzulegen.
Auditability	Für die Nachvollziehbarkeit und Analysierbarkeit müssen alle eingehenden API Aufrufe geloggt werden.
Auditability	Logfiles müssen in einer für den Menschen lesbaren Form formatiert sein.
Safety	Die Anwendung muss aus Datenschutzgründen eine Authentifizierungsmöglichkeit bieten.

8.2.2. Benutzbarkeit (usability)

Begriffe

- **Operability (Bedienbarkeit)**: Darunter fällt beispielsweise die Geschwindigkeit, in welcher ein User die gewünschten Daten erfassen kann.

Tabelle 8.4.: NFRs - Benutzbarkeit

Zuordnung	Anforderung
Operability	Längere Arbeitsschritte des Systems werden dem Benutzer als solche erkenntlich gemacht. Die Benutzeroberfläche wird dabei nicht blockiert.

8.2.3. Zuverlässigkeit (reliability)

Begriffe

- **Accuracy (Genauigkeit)**: Umfasst die Richtigkeit und die Genauigkeit der durchgeführten Berechnungen.
- **Recoverability (Fehlertoleranz)**: Die Reaktion, mit welcher die Anwendung auf einen Fehler antwortet (Fehlererholungsprozess).
- **Security (Angriffssicherheit)**: Im Gegensatz zu Safety bezieht sich in diesem Fall die Sicherheit auf den Schutz des Objektes vor seiner Umgebung.

Tabelle 8.5.: NFRs - Zuverlässigkeit

Zuordnung	Anforderung
Accuracy	Sensordaten werden nach Genauigkeit der gelieferten Rohdaten, maximal 64 Bit, abgespeichert.
Recoverability	Fehleingaben von Benutzern müssen minimal serverseitig validiert werden und mittels verständlichen Fehlermeldungen kommuniziert werden.
Recoverability	Für kritische Operationen wie zum Beispiel das Löschen von Business relevanten Daten, müssen immer Bestätigungen von Seiten des Benutzers eingefordert werden.
Security	Die Integrität der Sensordaten muss sichergestellt sein, indem die Kommunikation von und zu Geräten ausschliesslich über sichere Kanäle erfolgt.

8.2.4. Effizienz (performance)

Begriffe

- **Throughput (Durchsatz)**: Darunter fallen bspw. die Anzahl Transaktionen pro Sekunde, Datenraten, Verzögerungszeiten und Overhead.
- **Capacity (Kapazität)**: Bezieht sich auf auf die Anzahl (paralleler) Transaktionen, gleichzeitig ausführender Benutzer sowie die zu speichernden Daten (Datenmengen).

Tabelle 8.6.: NFRs - Effizienz

Zuordnung	Anforderung
Capacity (IoT-Provider)	Der Provider muss bis zu 10’000 Geräteanfragen pro Sekunden verarbeiten können.
Capacity (Benutzer-Plattform)	Die Plattform muss bis zu 1’000 Benutzeranfragen pro Sekunde bewältigen können.

8.2.5. Unterstützbarkeit (supportability)

Begriffe

- **Compatibility (Kompatibilität)**: Die Kompatibilität des Systems mit seinen früheren Versionen.
- **Configurability (Konfigurierbarkeit)**: Die Einfachheit der Konfiguration der Anwendung.
- **Installability (Installierbarkeit)**: Die Leichtigkeit, mit der das System installiert werden kann.
- **Localizability (Lokalisierbarkeit)**: Die Unterstützung von Sprachen und örtlichen Einstellungen.
- **Maintainability (Wartbarkeit)**: Die Leichtigkeit, mit der das System betrieben und gewartet werden kann.

Tabelle 8.7.: NFRs - Übertragbarkeit

Zuordnung	Anforderung
Localizability	Für eine Lokalisierung und Internationalisierung müssen die technischen Grundlagen bestehen, so dass die Übersetzung der Bildschirmtexte möglich ist. Die Codebasis muss für das Nachrüsten der Lokalisierung nicht verändert werden.
Maintainability	Die Systemkonfiguration ist über ein Konfigurationsfile leicht anpassbar und erfordert keine Änderung an der Codebasis.

9. Evaluation

Dieses Kapitel bietet eine kurze Übersicht der Evaluationsphase. Ein detaillierter Bericht zur Evaluation ist im Anhang beigefügt.

9.1. Technologiewahl

Im Folgenden werden die Beschlüsse bezüglich der Auswahl der zu verwendenden Technologien beschrieben.

9.1.1. User Interface

Das User Interface wird mit Hilfe von React und Redux umgesetzt. Diese Kombination bietet die Möglichkeit, das User Interface mit genügend Logik auszustatten, damit es als eigenständiger Client agieren kann. Die Kommunikation mit der Plattform soll ausschliesslich über die Client API geführt werden. Durch den Einsatz eines React UI Frameworks erhoffen wir uns die Entwicklungskosten für ein ansprechendes Grunddesign gering halten zu können.

Der Industriepartner setzt bei eigenen Projekten bereits auf React. Zusätzlich besitzt das Projektteam Erfahrungen im Umgang mit React und Redux. Wir sehen daher React als optimale Lösung, bei der uns unsere Erfahrung rasche Erfolge bescheren und wir trotzdem auf dem Technologie-Stack des Industriepartners bleiben können.

9.1.2. Gateway

Das Hauptaugenmerk der Arbeit liegt auf der Umsetzung der Plattform. Die Geräte dienen als Mittel zum Zweck um die Plattform unter realen Bedingungen testen zu können. Dafür bieten uns die Pycom-Geräte genug Einfachheit, um uns nicht mit zusätzlichen Problemen bei der Arbeit zu stören und genügend Möglichkeiten, um alle benötigten Features grundsätzlich abdecken zu können.

9.1.3. Core / Client API

Die Backend Anwendung wird als RESTful HTTP Web Service mit dem Java Spring Framework realisiert. Durch die Verwendung des Java Spring Frameworks erhoffen wir uns eine hohe Interoperabilität, ein einfaches Deployment und gut ausgebaute Integrationsmöglichkeiten zu Drittanbietern. Wir nehmen dabei höhere Entwicklungskosten aufgrund mangelnder Erfahrung mit Spring in Kauf.

9.1.4. IoT Provider

Als IoT Provider setzen wir auf AWS. Aus den evaluierten Produkten überzeugt Amazon mit einem reifen Produkt, gut ausgebauter SDK und umfangreicher Dokumentation. Im ersten Jahr ist der Service bis zu einer bestimmten (für uns ausreichenden) Grösse gratis. Wir legen uns damit vorerst auf einen fixen Anbieter fest. AWS ermöglicht es uns, durch einen simplen Einstieg schnell einen Prototyp zu erstellen. Die Schnittstelle zum Provider wird so abstrahiert, sodass der Austausch des Providers zu einem späteren Zeitpunkt mit wenig Aufwand möglich ist.

9.1.5. Inter-System-Kommunikation

Für die Kommunikation der IoT Geräte mit der Plattform und zum Client werden die weitverbreiteten und erprobten Kommunikationsprotokolle wie MQTT und HTTPS verwendet. Wir erwarten dadurch gut dokumentierte Schnittstellen und hohe Sicherheit der Kommunikationskanäle. Wir verzichten bewusst auf den Support von weiteren Protokollen, um den Umfang einzugrenzen und den Fokus auf die Plattformentwicklung zu setzen.

9.2. Umfang

Das Projekt wird grundsätzlich agil geführt. Deshalb wird fortlaufend gemäss der Priorisierung der Use Cases an der Umsetzung gearbeitet.

Aufgrund der abgesteckten Projektzeit werden aber Zielsetzungen mittels Meilensteinplanung definiert, welche richtungsweisend sind für die Einplanung und Umsetzung der Arbeitspakete.

10. Architektur

10.1. Übersicht

Dieses Kapitel bietet eine Übersicht der Systemarchitektur. Das Diagramm zeigt die Interaktionen und Beziehungen zwischen den einzelnen Komponenten vom Sensor bis zum Webinterface.

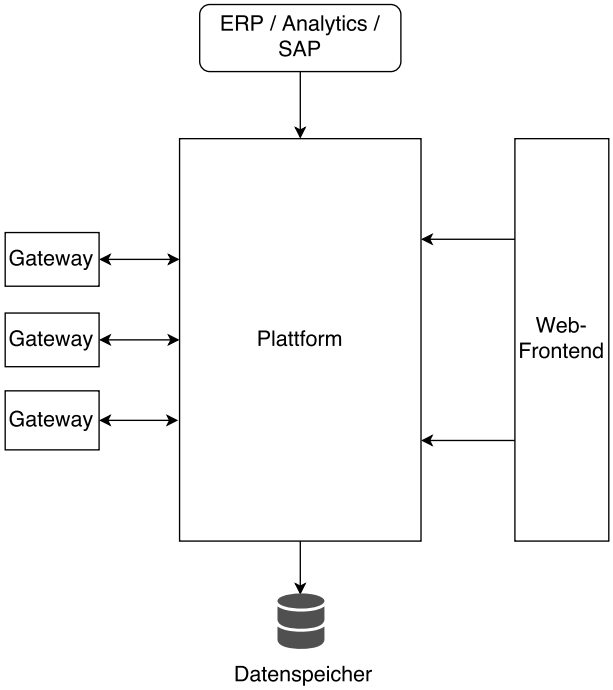


Abbildung 10.1.: Systemübersicht

Die einzelnen Komponenten werden in den folgenden Abschnitten genauer beschrieben.

10.1.1. Gateways

Gateways repräsentieren die eigentlichen IoT Geräte. Sie kommunizieren per MQTT mit der Plattform und sie sammeln und übermitteln die Daten der Sensoren. Ein durch die Plattform angestossenes OTA-Update ist möglich.

Die eingesetzten Geräte folgen, unabhängig von ihrem Typ, einem vorgegebenen Lebenszyklus.

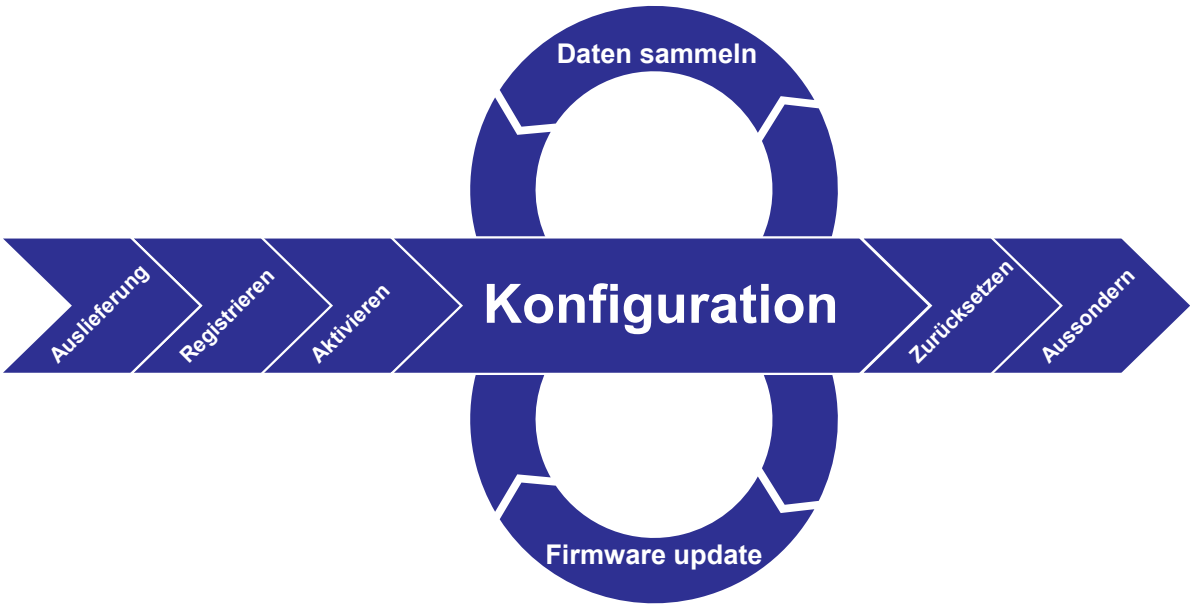


Abbildung 10.2.: Gateway Lebenszyklus

Die einzelnen Schritte des Lebenszyklus werden im Kapitel Sequenzdiagramme spezifischer erklärt. Die Schritte des Registrieren und Aktivieren werden dabei im Kapitel [Gerät-Provisionierung] behandelt, die Schritte Zurücksetzen und Aussondern im Kapitel Gerät-Dekommissionierung.

10.1.2. Plattform

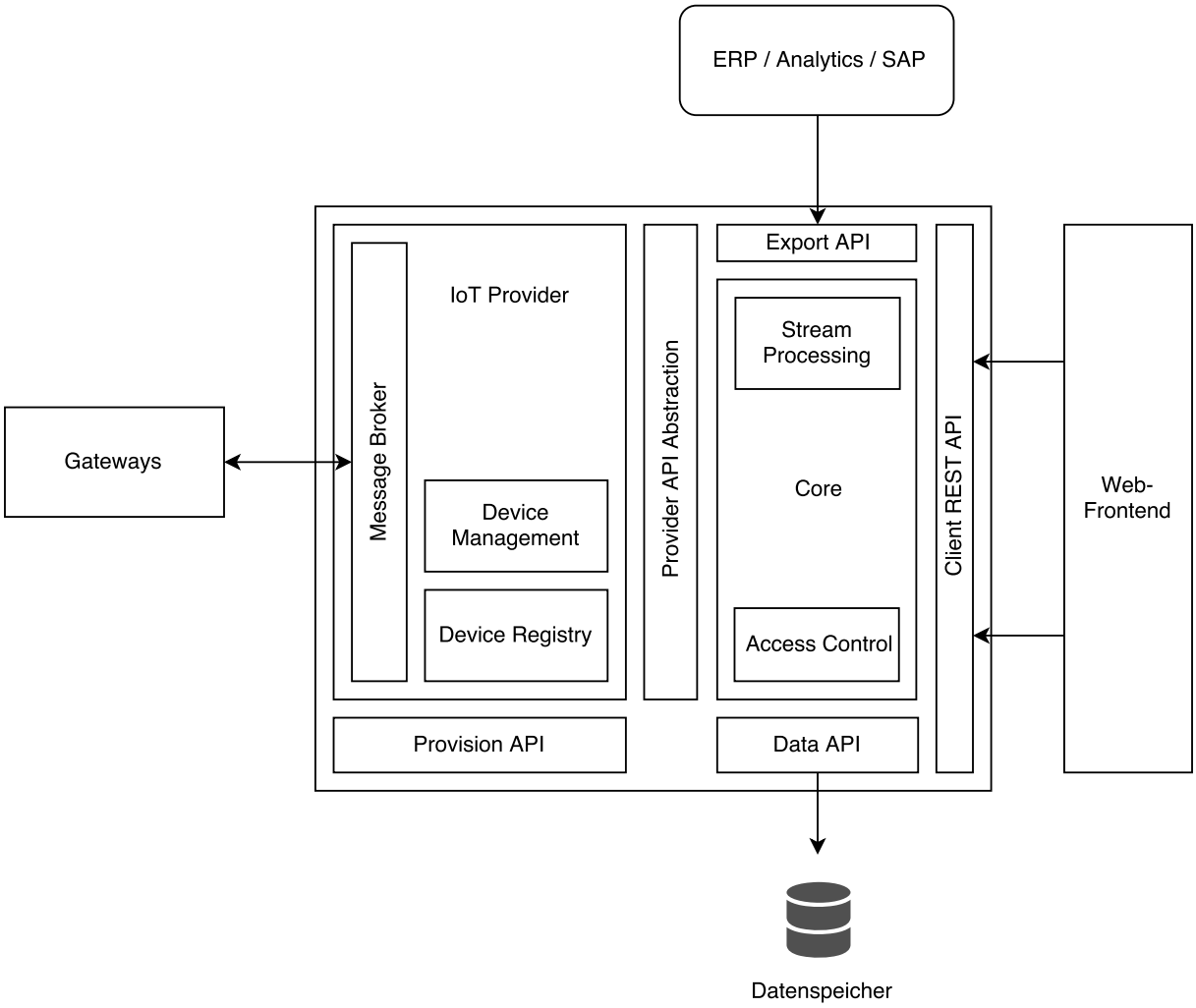


Abbildung 10.3.: Detailsansicht der Plattformkomponente

IoT Provider

Der IoT Provider kümmert sich um die Kommunikation mit den einzelnen Geräten und deren Verwaltung. Er unterstützt verschiedene Protokolle (HTTP, MQTT). Er stellt eine Schnittstelle zur Verfügung, um Geräte zu konfigurieren und zu verwalten.

Message Broker

Der Message Broker ist zuständig für den Austausch von Nachrichten zwischen den Geräten. Er empfängt Nachrichten basierend auf Topics und publiziert diese weiter an alle Abonnenten des Topics.

Device Registry

Die Device Registry identifiziert und authentifiziert Geräte, welche mit der Plattform kommunizieren.

Device Management

Das Device Management übernimmt die Verwaltung der Geräte in Bezug auf Konfiguration und Software. Sie ermöglicht OTA Updates und speichert den letzten bekannten Status bei Verbindungsverlust mit dem Gerät.

Core

Diese Komponente ist das eigentliche Herzstück der Applikation. Sie beinhaltet die Business Logic, steuert den Zugriff zum IoT Provider und übernimmt die Datenverarbeitung.

REST API

Diese Schnittstelle stellt die Daten für das Frontend (Webinterface) zur Verfügung. Sie kann auch von anderen Drittsystemen benutzt werden.

Provider API Abstraction

Stellt eine standardisierte Schnittstelle für den Zugriff auf den IoT Provider zur Verfügung. Diese Schicht kann ohne Codeänderungen der Plattform ausgetauscht werden um den Wechsel des IoT Providers zu ermöglichen. Je nach Provider dient sie nur als Mapper oder enthält selbst noch Logik.

Daten API

Das Data API beschreibt das Interface zum Datenspeicher für die Geräte- und Sensordaten. Die konkrete Speicherlösung bleibt somit flexibel und austauschbar. Mögliche konkrete Implementationen sind lokale Datenbanken oder ObjectStorage in der Cloud.

Provision API

Das Provision API beschreibt die Endpunkte für Geräte, welche sich zum ersten Mal verbinden. Es kümmert sich um die Autorisierung und Zertifizierung solcher Geräte zur Verbindungsaufnahme mit dem IoT Provider.

Export API

Ermöglicht es, Daten aus der Plattform zu exportieren, welche nicht oder nicht in diesem Format über die REST API verfügbar sind. Mögliche Formate sind csv, xml, json etc.

10.1.3. Webapplikation

Die Webapplikation bildet das UI für die Interaktion mit der Plattform. Sie wird als eine Single Page Applikation (SPA) realisiert. Sie holt die benötigten Daten von der Plattform mittels REST API.

10.1.4. Datenspeicher

Der Datenspeicher dient zur Speicherung aller aktuellen und archivierten Sensordaten der Plattform. Kann wenn gewünscht auf Drittanbieter ausgelagert werden.

10.1.5. Externe Systeme

Externe Systeme sind alle Applikationen und Dienste von Drittanbietern, welche mit der Plattform interagieren. Beispiele sind ERP Systeme, SAP Applikationen sowie auch Analyse-Tools für Daten.

10.2. Domainmodel

Die Grafik zeigt das grobe Domainmodel der Plattform. Das Benutzermanagement ist im untenstehenden Diagramm ausgenommen.

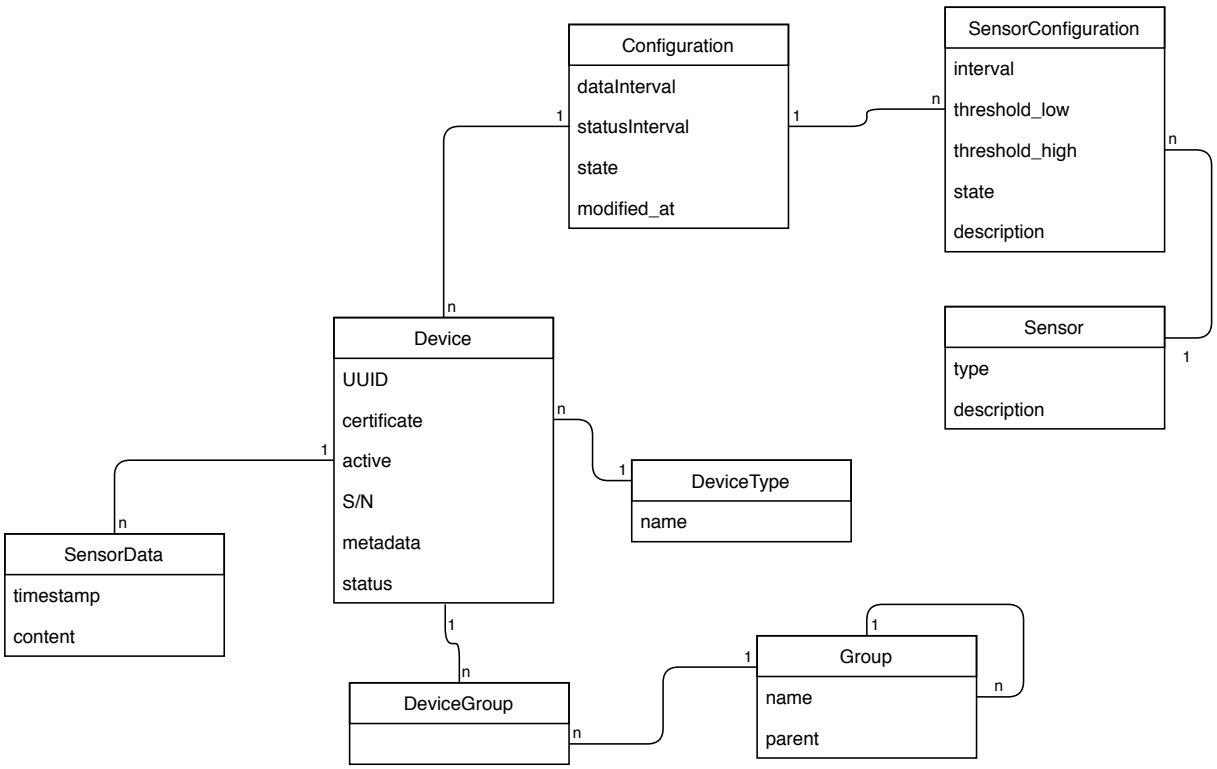


Abbildung 10.4.: Domainmodel

10.2.1. Sensor

Beschreibt die vorhandenen Sensoren, welche in eine Konfiguration aufgenommen werden können. Jeder Sensor besitzt einen Typ und Beschreibung.

10.2.2. Configuration

Stellt eine Konfiguration dar, welche einem Gerät zugewiesen werden kann. Sie besteht aus mehreren Sensoren und legt zusätzlich den Interval für Status- und Datenupdates fest.

10.2.3. SensorConfiguration

Weist die einzelnen Sensoren einer Konfiguration zu. Pro Sensor kann ein Status, ein Interval und eine Beschreibung gesetzt werden.

10.2.4. Device

Speichert die Daten der einzelnen Geräte wie deren Metadaten und Status. Auch die Zertifikate zur Authentifikation sind hier hinterlegt.

10.2.5. DeviceType

Ein Gerät ist je einem Gerätetyp zugewiesen, welcher beliebig benannt werden kann und zur schnellen Gruppierung dient.

10.2.6. Group

Definiert die möglichen Gruppen, in die ein Gerät eingeteilt werden kann. Eine Gruppe kann einer anderen Gruppe angehören und somit verschachtelt sein.

10.2.7. DeviceGroup

Verbindet die einzelnen Geräte mit den Gruppen. Ein Gerät kann in mehreren Gruppen sein.

10.2.8. SensorData

Enthält die vom den Geräten gesendeten Sensordaten. Diese sind mit einem Zeitstempel versehen und dem entsprechenden Gerät zugewiesen.

10.3. Sequenzdiagramme

10.3.1. Gerät erfassen

Folgende Sequenzdiagramme zeigen den Ablauf einer Geräteerfassung und die Interaktion der beteiligten Teilsysteme.

Voraussetzungen

Der Prozess sieht vor, dass Geräte initial mit einem kundenspezifischen Image versehen werden. Auf diesem werden folgende Parameter vorkonfiguriert:

- **Plattform**: Die URI der Plattform an welcher sich das Gerät bei der Inbetriebnahme registrieren soll.
- **Fabrik-Zertifikat**: Ein X.509-Zertifikat zur Authentifizierung, von einer zentralen CA (CloudGuard-CA) signiert.
- **Identifikations-Key (Key)**: Ein über alle hergestellten Geräte eindeutiger Schlüssel zur Identifikation eines Geräts. Er wird gebildet aus dem Geräte-Hersteller, dem Gerätetyp und der Seriennummer des Herstellers.

Der Kunde erhält mit der Auslieferung der Geräte eine Liste der Identifikations-Keys der gelieferten Geräte. Diese können als Bulk Operation in die Plattform importiert werden, um ihr die Geräte als anfrageberechtigt auszuweisen.

Gerät unbekannt

Die Plattform weist Anfragen von Geräten ab, deren Identifikations-Keys ihr nicht bekannt sind oder für den Betrieb noch nicht freigeschaltet wurden.

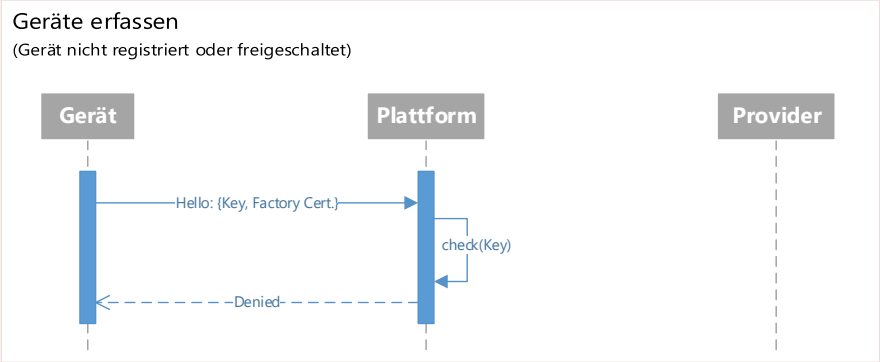


Abbildung 10.5.: Sequenzdiagramm Gerät erfassen: Gerät unbekannt

Gerät freischalten

Geräte welche aktiv verwendet werden sollen, müssen zuerst freigeschaltet werden. Die Plattform generiert dabei die Gerätezertifikate und sendet danach eine Anfrage an den IoT-Provider, um ein weiteres Gerät hinzuzufügen. Der Provider liefert darauf die Host-URI, über welche das Gerät mit dem Provider kommuniziert. Die nun erhaltenen Daten werden bis zur nächsten Anfrage des Geräts auf der Plattform zwischengelagert.

Sollte ein Gerät, welches erfasst werden soll, noch nicht mit seinem Identifikations-Key auf der Plattform registriert sein, kann ein Benutzer diesen Schritt auch von Hand vornehmen.

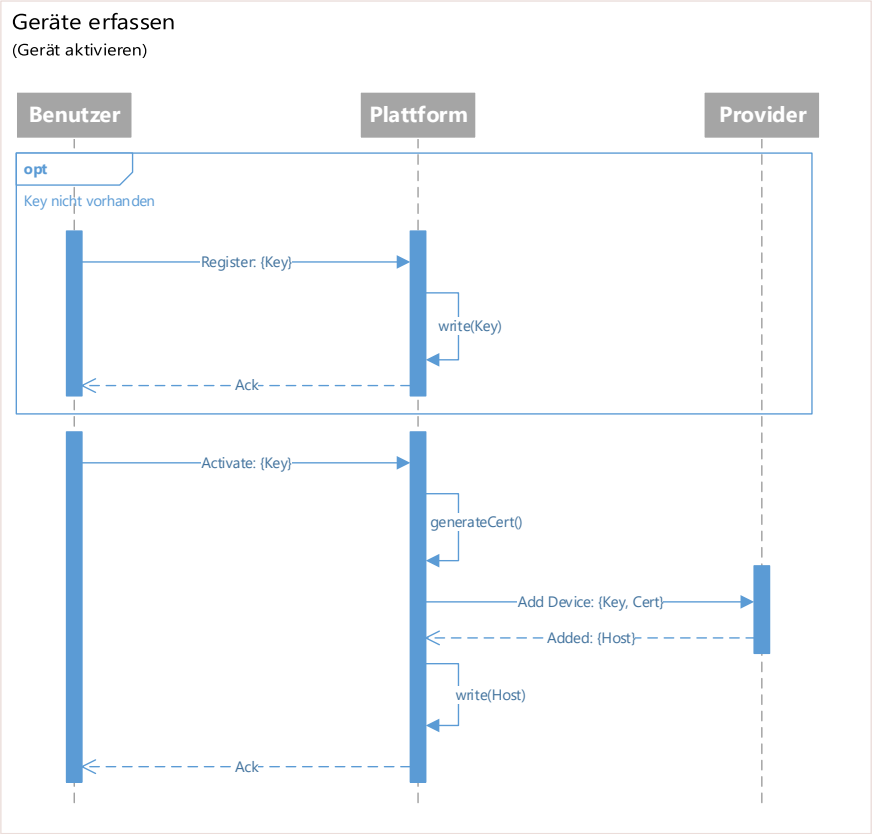


Abbildung 10.6.: Sequenzdiagramm Gerät erfassen: Gerät freischalten

Gerät aktivieren

Sobald ein Gerät registriert und freigeschaltet ist, kann es bei der Plattform die Parameter zur Kommunikation mit dem Provider anfragen. Bei der ersten Interaktion mit dem Provider übermittelt dieser eine minimale Grundkonfiguration an das Gerät. Es ist danach für die weitere Konfiguration vorbereitet.

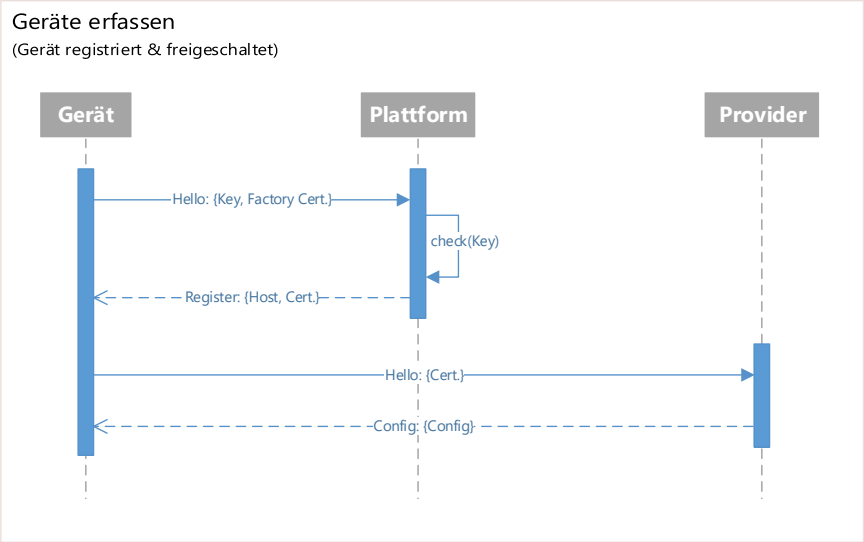


Abbildung 10.7.: Sequenzdiagramm Gerät erfassen: Gerät wird aktiviert

10.3.2. Geräte-Software update

Ein Software-Update, egal ob es sich um Firmware- oder ein Clientcode-Update handelt, wird durch eine Konfigurationsänderung auf dem Gerät durch den Provider eingeläutet. Das Gerät wird dabei in den Updatemodus versetzt.

Beim Update fragt das Gerät über HTTP GET, mit seiner derzeitigen Version als Parameter, ein Update-Manifest an. Das Manifest enthält eine Liste, welche Dateien gelöscht, sowie welche Dateien neu installiert werden sollen (inkl. URL und Hash für den Download). Nach dem Backup der zu ersetzenden Dateien werden die neuen Dateien heruntergeladen und gegen den dazugehörigen Hash geprüft. Insgesamt wird das Herunterladen einer Datei maximal fünfmal versucht, um Verbindungsschwierigkeiten auszugleichen.

Schlägt die Überprüfung des Hash bei einer Datei zu oft fehl, wird ein Rollback eingeleitet. Dabei werden alle neuen Dateien wieder gelöscht und die alten Versionen wieder hergestellt. Beim erfolgreichen Update werden die Backups am Ende gelöscht, um den Speicher freizugeben.

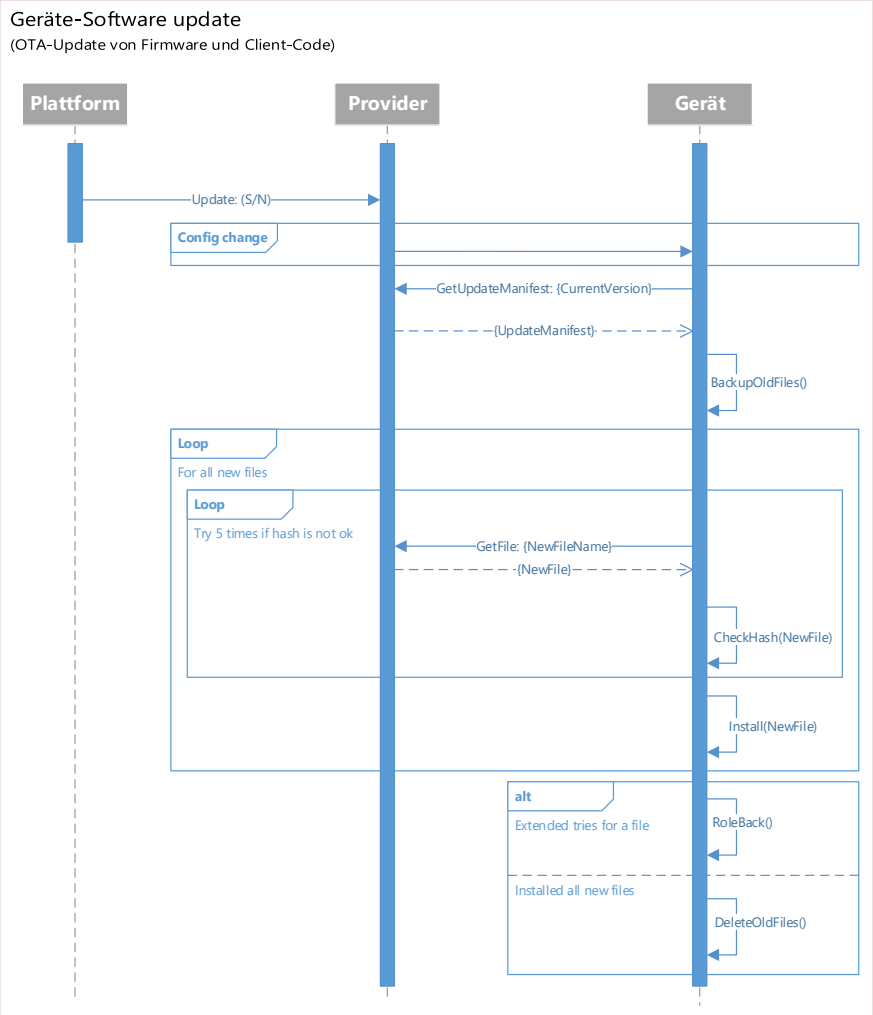


Abbildung 10.8.: Sequenzdiagramm Software-Update: Gerät erhält ein OTA-Update

10.3.3. Gerät-Dekommissionierung

Gerät deaktivieren und zurücksetzen

Wird ein Gerät dekommissioniert, wird es auf der Plattform sowie beim IoT-Service-Provider als inaktiv markiert. Das vorhandene Geräte-Zertifikat wird zurückgezogen (revoked). Dem Gerät wird ein Reset-Befehl gesendet. Das Gerät löscht daraufhin selbstständig sein Gerätezertifikat und setzt sich in den Auslieferungszustand zurück.

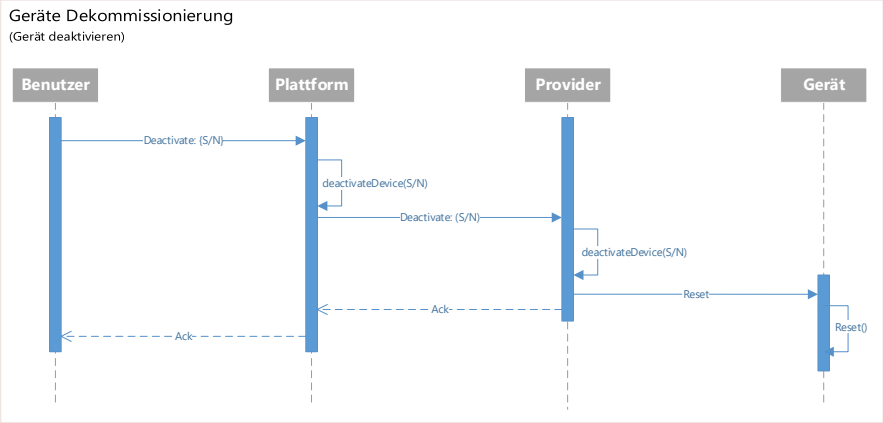


Abbildung 10.9.: Sequenzdiagramm Dekommissionierung: Gerät wird dekommissioniert

Verlorene Geräte

Sollte ein Gerät physisch und mittels Telekommunikationsmitteln unerreichbar bleiben, muss damit gerechnet werden, dass es den Reset-Befehl nicht erhalten hat und damit die letzte Konfiguration sowie das Gerätezertifikat auf dem Gerät verbleiben. Weil das Zertifikat jedoch zurückgezogen (revoked) wurde, wird jegliche Kommunikation bereits während des Verbindungsaufbaus unterbunden und ein verbleibendes Zertifikat stellt daher keine Sicherheitslücke dar. Bei zukünftigen Kommunikationsversuchen erhält das Gerät nachträglich einen Reset-Befehl als Antwort.

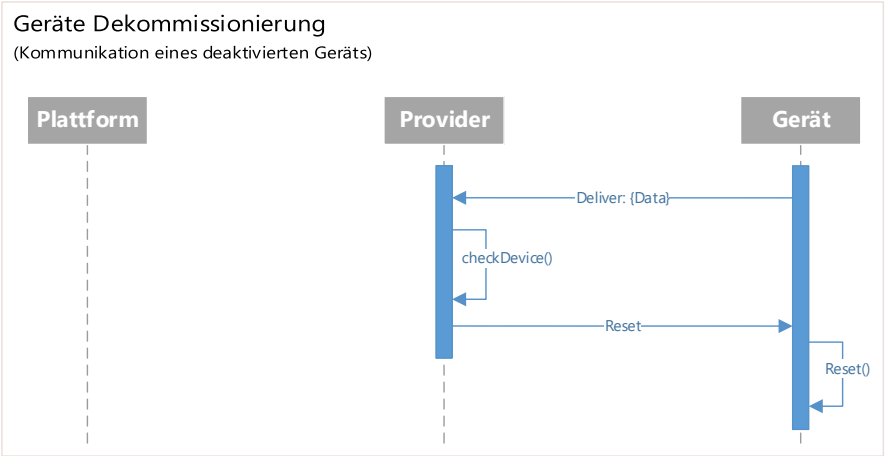


Abbildung 10.10.: Sequenzdiagramm Dekommissionierung: verlorene Geräte

10.3.4. Konfigurationsupdate

Bei einem Update der Konfiguration sendet die Plattform das Update an den Provider. Von dort aus wird es an die Geräte verteilt sobald diese erreichbar sind. Wenn das Update erfolgreich verteilt wurde, meldet der Provider dies zurück an die Plattform.

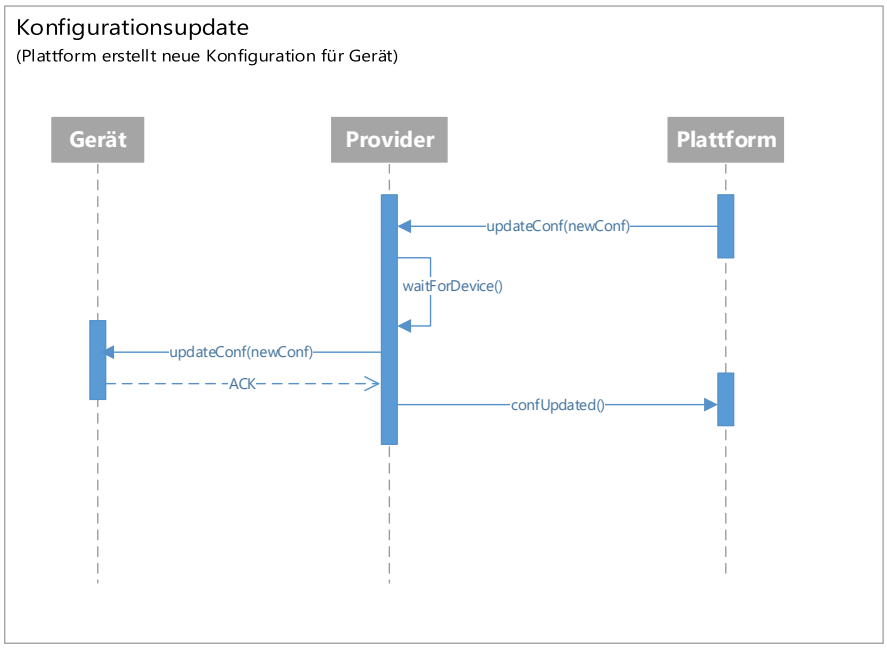


Abbildung 10.11.: Sequenzdiagramm Konfigurationsupdate

Konfiguration

Die Konfiguration ist aufgebaut als eine Aufstellung aus Key-Value Paaren. Eine Beispielskonfiguration:

```
{
    "sensors": [
        {
            "uid":"abc-123",
            "type":"temp",
            "interval":30,
            "threshold_low":15,
            "threshold_high":25,
            "state":"OFF",
            "description":"right corner"
        },
        {
```

```
            "uid":"abc-124",
            "type":"location",
            "interval":10,
            "state":"ON",
            "description":"GPS on roof"
        },
        {
            "uid":"abc-125",
            "type":"accel",
            "interval":1,
            "state":"ON",
            "description":"sensor on chip"
        },
    ],
    "statusInterval": 3600,
    "dataInterval": 600,
    "state":"ON",
    "lastUpdated": "2018-03-15 18:02"
}
```

10.3.5. Senden von Sensordaten

Die Geräte senden regelmässig die aktuellen Sensordaten an den Provider. Das Sendeinterval wird durch die Konfiguration bestimmt. Die Plattform kann beim IoT Provider als Observer registriert sein und erhält dadurch die Sensordaten.

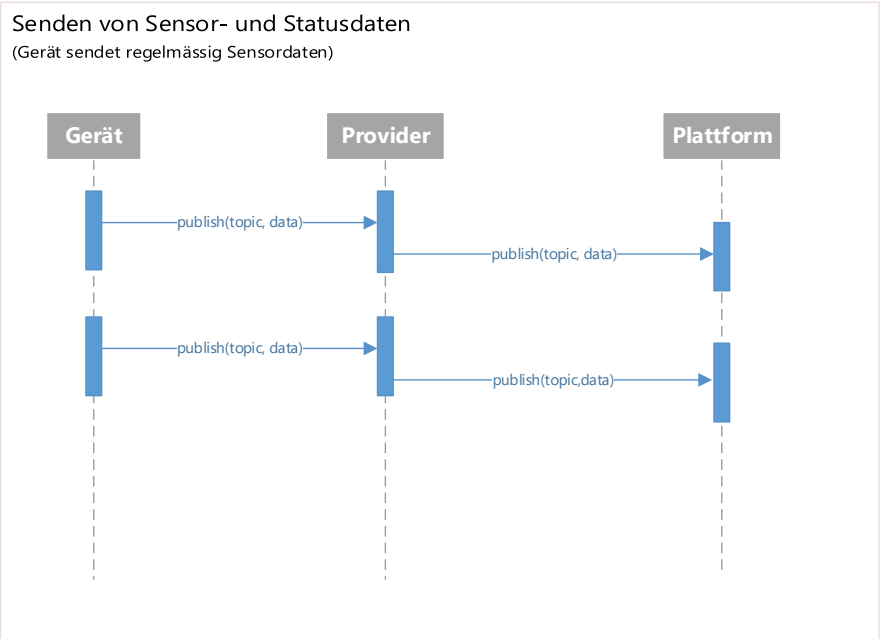


Abbildung 10.12.: Sequenzdiagram: Senden von Daten

Sensordaten

Ein Beispiel wie Sensordaten ausehen können:

```
{
    "timestamp": "2018-03-15 18:02",
    "type":"SHEDULED", //ERROR, ALERT, POWER_LOW
    "data": [
        {
            "sensor":"abc-123",
            "timestamp":"2018-03-15 18:01:30",
            "value": 22
        },
        {
            "sensor":"abc-123",
            "timestamp":"2018-03-15 18:02:00",
            "value": 21
        },
        {
            "sensor":"abc-123",
            "timestamp":"2018-03-15 18:02:30",
```

```
            "value": 20
        },
        {
            "sensor":"abc-124",
            "timestamp":"2018-03-15 18:01:10",
            "value": "47.359536,8.635645"
        },
        {
            "sensor":"abc-124",
            "timestamp":"2018-03-15 18:01:20",
            "value": "47.259536,8.735645"
        },
        {
            "sensor":"abc-124",
            "timestamp":"2018-03-15 18:01:30",
            "value": "  47.159536,8.835645"
        },
        {
            "sensor":"abc-124",
            "timestamp":"2018-03-15 18:01:40",
            "value": "47.059536,8.935645"
        },
    ]
}
```

Statusupdate

Ein Statusupdate könnte wie folgt aussehen:

```
{
    "timestamp": "2018-03-15 18:02",
    "battery": "50%",
    "uptime": "2547h"
}
```

11. Umsetzung

11.1. Gesamtsystem

11.1.1. Namensgebung

Das von uns entwickelte System erhielt Team-intern während der Entwicklungszeit den Codenamen „riot“. Der Name setzt sich aus dem bekannten englischen Ausdruck für das Internet der Dinge und dem englischen Verb „rolling“ zusammen. Der Name der Plattform bildet sich somit aus dem Akronym für den Ausdruck „rolling internet of things“, auf Deutsch „rollendes Internet der Dinge“. Da im Eisenbahnwesen die Gesamtheit der Schienenfahrzeuge als Rollmaterial bezeichnet wird, stellt das Verb rollen hier bewusst einen Bezug zu den Güterwagen her, für deren Überwachung unser System schlussendlich genutzt werden soll. Der Codename ist kurz, prägnant und bietet mit der Doppeldeutigkeit die nötige Spannung, um im Gedächtnis zu bleiben.

11.1.2. Übersicht

Riot besteht aus drei Teilsystemen. Die Client-Anwendung und die angeschlossenen Geräte werden als externe, austauschbare Komponenten betrachtet. Das folgende Systemdiagramm gibt einen Überblick über das Umgesetzte und die verwendeten Kommunikationsprotokolle.

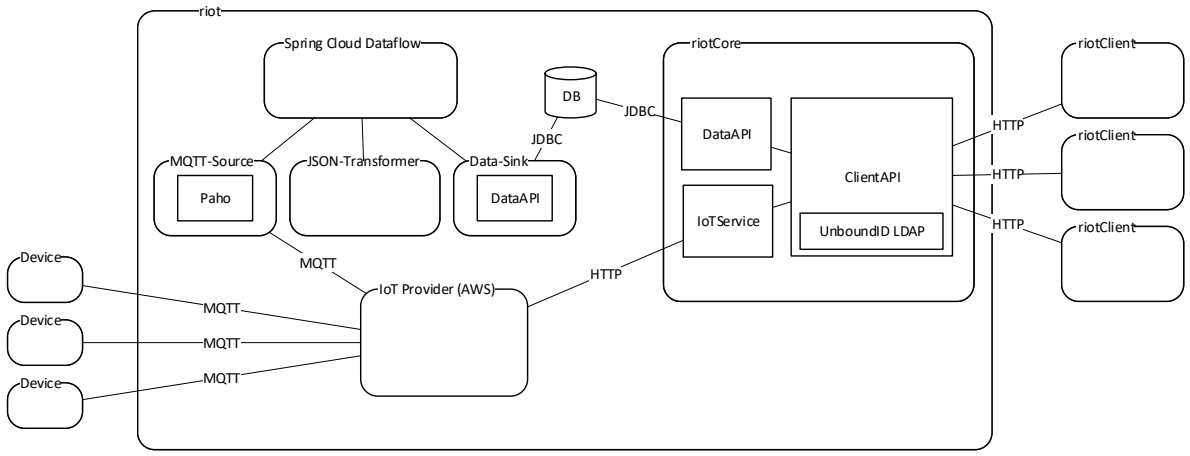


Abbildung 11.1.: Riot Systemdiagramm

11.2. Kernsystem

11.2.1. Überblick

Das Kernsystem umfasst das Client API und bindet die Schnittstellen zum IoT Provider an.

Der Client kann über ein RESTful http Webservice Daten konsumieren und verarbeiten.

Das Client API delegiert Anfragen des Clients an die Services (data access layer) weiter, validiert Eingangsdaten serverseitig und retourniert mittels HTTPStatus codes und data transfer objects.

11.2.2. Packages

Tabelle 11.1.: Packages im Client API Module

Package	Beschreibung
ch.hsr.riot.api.*	Enthält die API Controller , DTO's sowie Konfigurations- und Interceptor code für Authentifikation und Autorisierung.
ch.hsr.riot.configuration	Enthält den Konfigurationscode, der für das client API relevant sind
ch.hsr.riot.ldap.*	Enthält für die LDAP Anbindung relevanten Code, der vom client API benötigt wird.

11.2.3. Abhängigkeiten

Die Abhängigkeiten wurden mittels integrierter Analysemethode „Dependency Analysis with DSM“ der IntelliJ IDEA (Ultimate Edition) visualisiert.

Aufgeführt sind die Abhängigkeiten der Packages untereinander im gesamten Projekt und im Detail für das Client API:

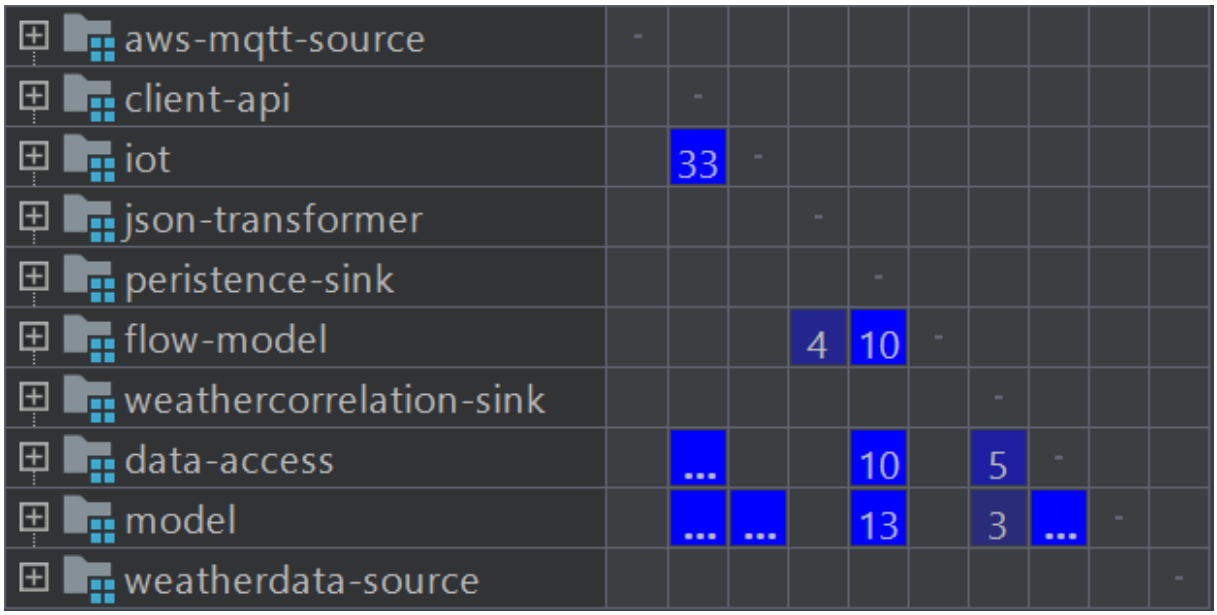


Abbildung 11.2.: Gesamtes Projekt

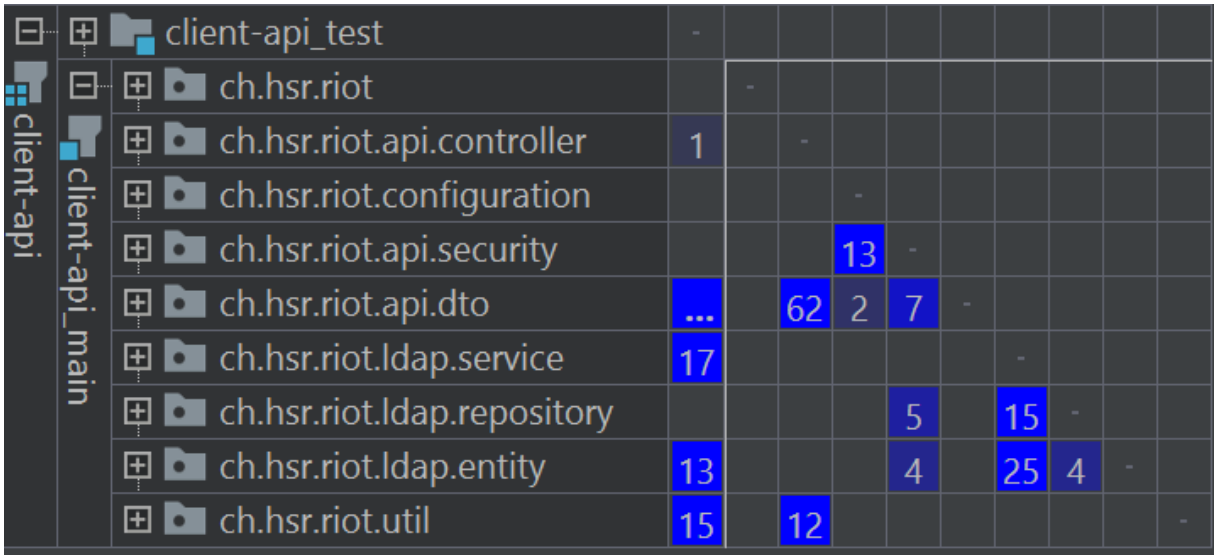


Abbildung 11.3.: Client API

Die Matrizen sind wie folgt zu lesen:

- Für jede Zeile sind die Abhängigkeiten zu anderen Packages in den Spalten, welche die Zeile jeweils in der Diagonalen kreuzt, ersichtlich.
- In den Zeilen sind die Abhängigkeiten von anderen Packages zum Package in der aktuellen Zeile ersichtlich. Man sieht also zum Beispiel, dass `ch.hsr.riot.api.controller` oft `ch.hsr.riot.api.dto` benötigt.
- externe Abhängigkeiten sind nicht aufgeführt

Es wird sichtbar dass:

- das gesamte Projekt und das Client API module mit dem Architektur Entwurfsmuster „Layers“ implementiert wurde, gegliedert in Packages
- keine zyklischen Abhängigkeiten existieren (das wären rot eingefärbte Kästchen)
- das Dependency Analyse Tool einige Abhängigkeiten nicht korrekt erkennt hat, weil diese in Form von kompilierten JARs eingebunden wurden, wie zum Beispiel für die data-flow Packages `weatherdata-source` oder `persistence-sink`.
- das Client API im Wesentlichen die Packages *.iot,* .data-access, und *.model benötigt.

11.2.4. API Controller

Die API Endpunkte wurden im Spring Boot Framework mit API Controllern realisiert.

Dazu wird im Spring Framework per Annotation, eine Form von aspektorientierter Programmierung, ein Controller mittels `@Controller` annotiert:

```
1 @Controller
2 @RequestMapping("/config")
3 @Api(description = "Configurations endpoint")
4 public class ConfigController { /*...*/ }
```

11.2.5. Sicherheit

Benutzer melden sich über den `/login` Endpunkt mit ihren Credentials an und autorisieren sich fortan mittels JWT (Bearer) Token, welches vom Server per response header zugestellt wird.

```
1 authorization: Bearer EXAMPLEEXAMPLEzUxMiJ9.
    eyJzdWIiOiJyaW90IiwiZXhwIjoxNTI4NjI5NzMzfQ.
    c3OCPXZ_0uW5oUXlVioz4TxyykVD5WaUJXpYFvc2TaKXNef4hQopboCEQ4ra-8EwL-
    OYA1hsDCIREFIyHcCzvg
```

Die API Endpunkte sind, mit Ausnahme des login Endpunkts, nur für autorisierte Benutzer zugänglich. Ein Benutzer mit einem gültigen Bearer Token gilt als autorisiert. Zwischen Client und API wird eine verschlüsselte Kommunikation sichergestellt, indem unverschlüsselte http Anfragen auf https redirected werden.

11.2.6. Dependency Management

Buildtime

Für das Auflösen und Laden der Abhängigkeiten zu Build Zeiten wird Gradle[1] verwendet. Für den Zweck dieser Arbeit ist der Funktionsumfang, die Konfigurierbarkeit und Flexibilität ausreichend. Gradle wurde Maven vorgezogen, um neue Erfahrungen sammeln zu können. Die Entscheidung für den Einsatz von Gradle basiert also auf persönlicher Präferenz und ist nicht auf funktionsbezogene Unterschiede zurückzuführen.

Runtime

Für das Management der Abhängigkeiten zu den Services und Components wird das Springinterne Dependency Management verwendet. Zum Einsatz kommt Constructor injection. Dazu wird der Constructor mit `@Autowired` annotiert. Dies hat gegenüber der Field injection einige Vorteile, zum Beispiel können die fields **`final`** deklariert werden und der Controller ist nach erfolgreichem Durchlaufen des Constructors vollständig initialisiert und „einsatzfähig“. Das Verwenden von DI bringt weitere Vorteile, wie verbesserte Testability (Möglichkeit für gezieltes Faking und Mocking um die Units besser zu isolieren) und eine bessere Erweiter- und Wartbarkeit, weil weniger Code angepasst werden muss, wenn beispielsweise eine neue Abhängigkeit eingeführt werden muss.

```
1  private final DataTransferObjectModelMapper modelMapper;
2  private final DeviceConfigService configService;
3  private final DeviceService deviceService;
4  private final ConfigurationValidator configurationValidator;
5  private final Logger logger;
6
7  @Autowired
8  public ConfigController(DeviceConfigService configService,
9                          DeviceService deviceService,
10                         DataTransferObjectModelMapper modelMapper,
11                         ConfigurationValidator configurationValidator,
12                         Logger logger
```

[1] https://gradle.org/

```
13 ) {
14     this.configService = configService;
15     this.deviceService = deviceService;
16     this.modelMapper = modelMapper;
17     this.configurationValidator = configurationValidator;
18     this.logger = logger;
19 }
```

Um Services oder Komponenten für das DI Management verfügbar zu machen, können innerhalb von mittels `@Configuration` annotierten Klassen `Beans` angelegt werden, welche als factory dienen, wenn eine Instanz einer bestimmten Klasse benötigt wird.

```
1  @Bean
2 Logger getLogger(InjectionPoint injectionPoint){
3     return LoggerFactory.getLogger(injectionPoint.getMethodParameter().
          getContainingClass());
4 }
```

11.2.7. API Documentation mit Springfox

Die API Documentation wird mit Hilfe von Springfox, einer Spring basierten Implementierung von Swagger[2], automatisch anhand der API Controller erzeugt. Die API wird also mittels code first Ansatz dokumentiert und damit automatisch aktuell gehalten. Für den Einsatz von Swagger sprechen das grosse Ökosystem (unter Anderem gibt es zahlreiche Clientbezogene Codegeneratoren für die Models) und die Erprobtheit.

Die aktuelle API Dokumentation ist einsehbar unter https://client.riot.bitterlin.net/api/swagger-ui.html, basierend auf der api-doc unter `api/v2/api-docs`.

Abhängigkeiten

In Gradle werden folgende Abhängigkeiten eingeführt:

```
1 repositories {
2   jcenter()
3 }
4
5 dependencies {
6     compile("io.springfox:springfox-swagger2:2.8.0")
```

[2] https://swagger.io/

```
7       compile("io.springfox:springfox-swagger-ui:2.8.0")
8       compile("io.springfox:springfox-data-rest:2.8.0")
9   }
```

springfox-swagger2:
Enthält die Kernfunktionalität von Springfox, welche es ermöglicht, eine API Dokumentation mit Swagger 2 zu erzeugen.

springfox-swagger-ui:
Enthält die Swagger UI Komponente, welche die Swagger Dokumentation unter einer konfigurierbaren URL anbietet. Defaultmässig ist die API Dokumentation unter `"/api/swagger-ui.html"` einsehbar.

springfox-data-rest:
Ist in der Lage, automatisch eine Swagger Dokumentation für Spring Data REST repositories zu erzeugen.

Konfiguration

Springfox wird mittels Annotations und code first configuration eingerichtet:

```
1  @Configuration
2  @EnableSwagger2
3  public class SwaggerConfig { ... }
```

Mittels der `@EnableSwagger2` Annotation wird für die Applikation Springfox mit Swagger2 bereitgestellt.

Als nächstes wird Springfox konfiguriert. Dies geschieht mittels eines `Docket` Bean, welches den primären Konfigurationsmechanismus darstellt:

```
1  @Bean
2  public Docket api() {
3      TypeResolver typeResolver = new TypeResolver();
4      return new Docket(DocumentationType.SWAGGER_2)
5              .select()
6              .apis(RequestHandlerSelectors.basePackage("ch.hsr.riot.api"
                   ))
7              .paths(PathSelectors.any())
8              .build()
9              .useDefaultResponseMessages(false)
10             .apiInfo(apiInfo())
11             .additionalModels(typeResolver.resolve(UserLoginDTO.class),
                   typeResolver.resolve(SuccessfulLoginDTO.class))
```

```
12              .securitySchemes(Arrays.asList(apiKey()))
13              .securityContexts(Arrays.asList(securityContext()));
14  }
```

Weitere Konfigurationsbeispiele sind unter https://springfox.github.io einsehbar.

Das Swagger UI lässt sich ebenfalls per Code konfigurieren:

```
1  @Bean
2  UiConfiguration uiConfig() {
3      return UiConfigurationBuilder.builder()
4              .deepLinking(true)
5              .defaultModelsExpandDepth(1)
6              .defaultModelExpandDepth(1)
7              .defaultModelRendering(ModelRendering.EXAMPLE)
8              .displayRequestDuration(true)
9              .docExpansion(DocExpansion.NONE)
10             .filter(false)
11             .maxDisplayedTags(null)
12             .operationsSorter(OperationsSorter.ALPHA)
13             .showExtensions(false)
14             .tagsSorter(TagsSorter.ALPHA)
15             .validatorUrl(null)
16             .build();
17 }
```

Im Anschluss werden die erkannten API Controller Klassen nach ihrem Typ (GET, POST etc.) und die verwendeten Data Transfer Objects im Swagger UI aufgezeigt.

11.2.8. Data Transfer Objects

Für die Kommunikation mit den Clients, werden data transfer objects eingesetzt. Diese kapseln die essentiellen Daten, welche für die verschiedenen API Endpoints relevant sind. Sie sorgen dafür, dass unnötige Daten nicht zum Client gelangen und sind gleichzeitig eine Art der DoKumentation, die dem Konsumenten des API zeigen, welche Objekte (in JSON repräsentiert) von den Endpunkten konsumiert bzw. geliefert werden.

Die Konvertierung der JSON Strings zurück in die DTO's wird vom Spring Boot Framework unter Einsatz eines JSON serializer automatisch durchgeführt.

Für die Transformation der models in die dto's und umgekehrt wird ein ModelMapper eingesetzt. Dies erhöht die Wartbarkeit und reduziert die Fehleranfälligkeit, weil die Transformationen nicht händisch gepflegt werden müssen, wenn sich models oder DTO's ändern.

Abhängigkeit

```
dependencies {
    ...
    compile('org.modelmapper:modelmapper:1.1.0')
}
```

Konfiguration

Der ModelMapper bietet die drei Mapping Strategien STANDARD, LOOSE und STRICT an. Für die Anwendung in riot ist die convention basierte STANDARD Strategie ausreichend und es ist keine weitere Konfiguration notwendig.

Verwendung

Transformation eines `CreateDeviceDTO deviceDto` (dto) zu einem `Device device` (model):

```
Device device = modelMapper.map(deviceDto, Device.class);
```

Transformation eines model in ein dto:

```
DeviceDTO deviceDto = modelMapper.map(device, DeviceDTO.class);
```

11.2.9. Testing

Das API (`ch.hsr.riot.api`) ist mittels automatischen unit tests mit folgender Abdeckung getestet:

Tabelle 11.2.: Testabdeckung ch.hsr.riot.api.*

Package	class	method	line
ch.hsr.riot.api.controller	100% (6/ 6)	83.8% (31/ 37)	83.1% (138/ 166)
ch.hsr.riot.api.dto	100% (12/ 12)	96.4% (53/ 55)	97.1% (67/ 69)
ch.hsr.riot.api.security	100% (5/ 5)	100% (14/ 14)	91.2% (62/ 68)

Die Testabdeckung liegt über 80%.

11.2.10. Bekannte Einschränkungen und Probleme

- Für das Client API gibt es keine *automatisierten* integrations- oder load tests.

11.2.11. Datenhaltung

Die Datenhaltung läuft ausschliesslich über den data-access layer. Darüber werden mit Hilfe des Repository Patterns und durch Verwendung der CrudRepository<Config, String> durch Spring (`org.springframework.data.repository.CrudRepository`) mittels Services auf die Daten zugegriffen. Die Repositories und Services sind unter dem Package ch.hsr.riot.persistence wie folgt organisiert:

Tabelle 11.3.: Repositoryübersicht

Repository	Beschreibung
DeviceRepository	Bietet CRUD für Devices an.
DeviceConfigRepository	Bietet CRUD für DeviceConfiguration an
SensorDataRepository	Bietet CRUD für SensorData an

Der Einsatz der von Spring bereitgestellten CrudRepository Implementationen reduziert den Boilerplate und erspart dem Entwickler die Anbindung an die konkrete persistence stores.

Tabelle 11.4.: data-access Services Übersicht

Services	Beschreibung
DeviceService	Delegiert die CRUD Aufrufe für Devices an die entsprechende CrudRepository Implementation weiter
DeviceConfigService	Delegiert die CRUD Aufrufe für DeviceConfig an die entsprechende CrudRepository Implementation weiter
SensorDataService	Delegiert die CRUD Aufrufe für SensorData an die entsprechende CrudRepository Implementation weiter

Die Verwendung der data-access Services abstrahiert den eigentlichen Datenbankzugriff auf die eingesetzte Datenhaltungstechnologie und bietet die Möglichkeit für weitere datenhaltungsrelevante Businesslogik, wie zum Beispiel das Transformieren von Datenbankexceptions.

11.2.12. Anbindung IoT Provider

Die Kommunikation mit dem IoT Provider geschieht über das Interface `IotService`:

```
1  Package ch.hsr.riot.iot;
2
3  import ...
4
5  public interface IoTService {
6
7      CreateDeviceResult createDevice(Device device) throws Exception;
8
9      DeleteDeviceResult deleteDevice(Device device) throws Exception;
10
11     UpdateDeviceResult updateDevice(Device device) throws Exception;
12
13     UpdateDeviceConfigResult updateDeviceConfig(Device device, Map<
           String,Object> config) throws Exception;
14
15     UpdateDeviceCertificateResult updateDeviceCertificate(Device device
           , String certificate) throws Exception;
16
17     RevokeDeviceCertificateResult revokeDeviceCertificate(Device device
           ) throws Exception;
18 }
```

Im Package ch.hsr.riot.iot existieren zwei unabhängige Implementationen der Schnittstelle:

Tabelle 11.5.: Übersicht der IoT Service Implementationen

Implementation	Beschreibung	Package
AWSIoTService	Implementation, welche mit Hilfe des AWS IoT SDK mit AWS kommuniziert	ch.hsr.riot.iot.aws
DummyIoTService	Leerimplementation, welche für UnitTests oder während der Entwicklung eingesetzt werden kann	ch.hsr.riot.iot.dummy

Die Implementationen können von den Konsumenten, in unserem Fall dem Client API, per Konfiguration ausgetauscht werden, ohne dass Code angepasst werden muss. Die API Keys, welche von der AWSIoTService Implementierung benötigt werden, um mit AWS erfolgreich zu kommunizieren, sind

ebenfalls konfigurierbar.

11.3. Dataflow

In diesem Kapitel geht es um das in die Plattform integrierte „stream processing“ von Daten. Damit ist die Verarbeitung von Sensordaten in Echtzeit gemeint. Während der Evaluationsphase war dieser Teil noch nicht eingeplant.

11.3.1. Spring Cloud Data Flow

Spring Cloud Data Flow ist ein Toolkit zum Bau von Echtzeit Datenverarbeitung. Die einzelnen Streams bestehen aus Spring Boot Applikationen, aufbauend auf dem Spring Cloud Stream und Spring Cloud Task Framework. Für den Betrieb benötigt Cloud Data Flow eine Message Queue und eine Datenbank. In unserem Fall kommt RabbitMQ als Message Queue und PostgreSQL als Datenbanktechnologie zum Einsatz. Über eine Webinterface oder API können die einzelenen Streams und Services verwaltet werden. Ein Maven Repository speichert die für die einzelnen Services benötigten JARs.

Streams

Streams sind das Herzstück von Dataflow. Jeder Stream ist aus Applikationen zusammengesetzt, deren Ein- und Ausgänge durch Nachrichtenkanäle miteinander verbunden sind. Streams können linear verlaufen oder auch Verzweigungen aufweisen.

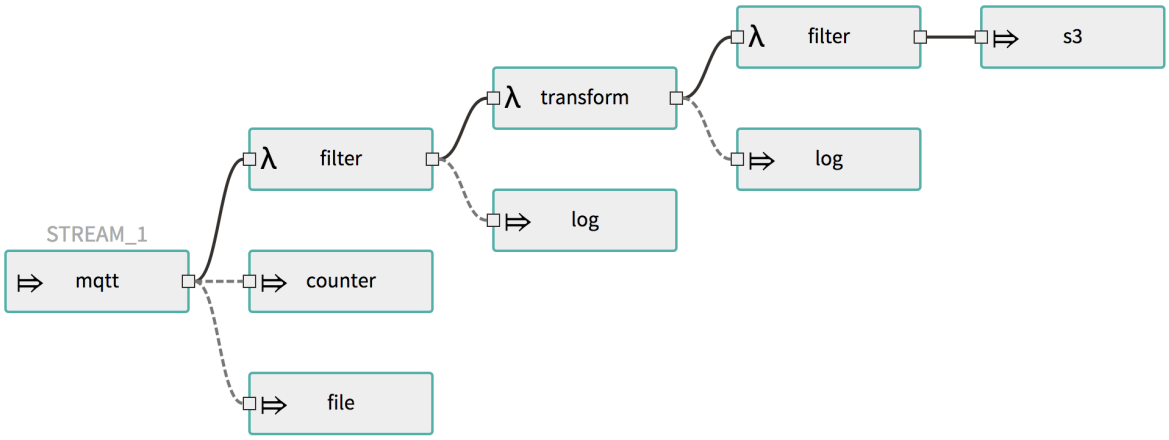


Abbildung 11.4.: Stream mit Verzweigungen

Die Nachrichtenkanäle steuern den Datenfluss vom Ausgang der einen Anwendung zum Eingang der nächsten. Im Hintergrund werden diese Kanäle als Pipelines in der Message Queue erstellt.

Source, Processor, Sink

Grundsätzlich gibt es drei Arten von Applikationen. Diese unterscheiden sich in der Anzahl der In- und Outputs. Für jede der drei Arten gibt es Beispielapplikationen, welche die bekanntesten Anwendungsfälle abdecken. Eigene Anwendungen können eine dieser Arten implementieren oder eigene Schnittstellen mit mehr als einem In- und Output definieren.

Source: keine Inputs, 1 Output

Eine Source ist somit ein Produzent von Daten. Daten werden entweder in regelässigen Abständen oder bei bestimmten Events, zum Beispiel beim Eintreffen einer MQTT Nachricht, generiert.

Processor: 1 Input, 1 Output

Prozessoren sind für die Verarbeitung von Daten zuständig, wie zum Beispiel das Konvertieren, Aggregieren oder Filtern von Werten. Eingehende Daten werden verarbeitet und dann an den Ausgang weitergeschickt.

Sink: 1 Input, keine Outputs

Sinks dienen als Endpunkte von Streams. Der gebräuchliste Anwendungfall ist das Speichern von Daten in einer Datei oder Datenbank. Für die bekanntesten Technologien (S3, MongoDB, JDBC, FTP etc.) werden bereits fertige Beispielapplikationen angeboten.

Tasks

Tasks erlauben es, kurzlebige Applikationen auf Abruf auszuführen. Das Resultat nach der Ausführung wird von Dataflow gespeichert. Es wurden für den Prototyp keine Tasks verwendet.

11.3.2. Integration in die Plattform

Spring Cloud Data Flow ist eine eigenständige Anwendung, welche unabhängig von den anderen Komponenten deployed werden kann. Für den Prototyp läuft Dataflow jedoch auf dem gleichen Server wie das Kernstück der Plattform.

11.3.3. Verarbeitung von Sensordaten

Um eine flexible und konfigurierbare Verarbeitung der Sensordaten zu ermöglichen, wurde entschieden, diese direkt durch Cloud Data Flow in die Plattform einzuspeisen. Das geschieht in drei Schritten:

1. Die Werte werden von einer MQTT Schnittstelle gelesen.
2. Eingetroffene Nachrichten werden in ein für die Plattform verständliches Format konvertiert.
3. Die Werte werden in die Datenbank gespeichert.

Diese drei Schritte werden im nächsten Abschnitt genauer erläutert.

MQTT-Source

Im Starterpaket des Frameworks wird bereits eine Source angeboten, welche MQTT Nachrichten empfangen kann. Jedoch ist es mit dieser nicht möglich, sich mit AWS zu verbinden, da die Authentifikation mit Zertifikaten von der Source nicht unterstützt wird. Deshalb musste die bestehende Source um diese Funktionalität erweitert werden. Dies wurde mittels einer neuen Konfiguration erreicht, welche den darunterliegeneden Paho Client mit den Zertifikaten konfiguriert. Es wurde so umgesetzt, dass auch direkt der MQTT-Sink mit Zertifikaten verwendet werden kann. Die Zertifikate können entweder in einem Verzeichnis auf dem Server abgelegt, oder auch direkt als String übergeben werden. Mit der erweiterten Version ist es nun möglich, Verbindungen (mittels Zertifikatauthentifikation) zu den AWS Servern aufzubauen. Die Applikation ist jedoch nicht auf AWS beschränkt sondern generell einsetzbar. Auch wurde die bestehende Funktionalität belassen, zum Server ohne Zertifikatauthentifikation verbinden zu können.

Json-Transformer

Dieser Prozessor wandelt die eingehenden Nachrichten im JSON Format in Java-Objekte um, welche dann zur Speicherung weitergeleitet werden. Grundsätzlich wird diese Funktionalität von einem ObjectMapper übernommen. Dieser kümmert sich um die korrekte Konvertierung der eingehenden Daten in das passende Java Object Model. Eine Schwierigkeit war die Konfiguration für die korrekte Konvertierung von Joda Datetime.

Persistence-Sink

In diesem Teil werden die angelieferten Werte in die Datenbank gespeichert. Dies geschieht durch den gleichen Service, welcher auch von der Plattform genutzt wurde. Der Sink ist daher unabhängig von der darunterliegenden Datenhaltung und hat lediglich eine Abhängigkeit zum dataservice. Beim Abspeichern der Werte wird auch die korrekte Verknüpfung zu den referenzierten Geräten sichergestellt.

11.3.4. Korrelation von Daten

Um einen Anwendungsfall zu demonstrieren, wurde versucht, eine aufwendigere data pipeline mittels Spring Cloud Data Flow einzurichten. Zu dem Zweck wurden die Packages `weatherdata-source` und `weathercorrelation-sink` implementiert. Die Absicht war es, aufzuzeigen, inwiefern bestehende Sensordaten (in unserem Fall Temperaturdaten) mit dem Wetter zum jeweiligen Zeitpunkt korrelieren. Bis zum Schluss der Projektarbeit konnte diese Pipeline nicht in Betrieb genommen werden. Grund dafür waren einerseits eine Unerfahrenheit im Umgang mit dem Framework und die unerwartete Komplexität beim Konfigurieren der Multi In- und Outputs der Sink bzw. Source.

11.4. IoT-Provider

Aufgrund der umfangreichen Funktionalität, welche Riot einmal annehmen soll, wurde schnell klar, dass während dieser Arbeit nicht alle Teilsysteme umgesetzt werden können. Bereits in der [# Evaluation] wurde daher AWS IoT ausgesucht um uns während der Umsetzung als IoT Provider zu dienen. Dadurch konnten wir wertvolle Zeit einsparen und hatten trotzdem Zugriff auf einen voll entwickelten IoT Provider nach unseren Bedürfnissen.

11.4.1. Certificate Management

Die Authentifikation von Geräten gegenüber dem IoT Provider findet mittels Zertifikaten statt. Dies ist heute ein Standard und wird von allen grossen Providern unterstützt. Über die Plattform können die Zertifikate von Geräten verwaltet werden.

Eigene CA

Statt das Ausstellen von Geräte-Zertifikaten dem IoT Provider zu überlassen, kann die eigene Certificate Authority (CA) benutzt werden, um Gerätezertifikate zu signieren. Dazu muss beim IoT Provider die eigene CA registriert werden, damit dieser die Zertifikate verifizieren kann. Dieser Schritt unterscheidet sich von Provider zu Provider. Die Vorgehensweise für AWS ist hier zu finden: https://docs.aws.amazon.com/iot/latest/developerguide/device-certs-your-own.html#register-CA-cert. Dieser Schritt muss noch direkt beim Provider durchgeführt werden und kann nicht über die Plattform erledigt werden. Grundsätzlich kann diese Funktionalität aber auch über die IoT Provider Schnittstelle implementiert und von der Plattform angeboten werden.

Zertifikate generieren

Die Generierung von Zertifikaten ist noch nicht in der Plattform implementiert. Um ein Zertifikat zu generieren sind die folgenden Schritte nötig:

- Key pair generieren

```
openssl genrsa -out deviceCert.key 2048
```

- CSR erstellen

```
openssl req -new -key deviceCert.key -out deviceCert.csr
```

- Aus CSR ein Zertifikat erstellen

```
openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key -
CAcreateserial -out deviceCert.pem -days 500 -sha256
```

Ein Anleitung von AWS (aus welcher auch die oben gezeigten Beispiele stammen) ist hier zu finden: https://docs.aws.amazon.com/iot/latest/developerguide/device-certs-your-own.html

Zertifikate registrieren

Nachdem die Zertifikate generiert wurden, müssen sie dem IoT Provider bekannt gemacht werden. Diese Funktionalität wird von der Plattform zur Verfügung gestellt. Pro Gerät kann ein Zertifikat heraufgeladen werden, welches dann beim Provider hinterlegt und aktiviert wird. Danach ist das Gerät berechtigt mit der Plattform zu kommunizieren. Mittels API der Plattform kann ein Zertifikat auch wiederrufen werden. Somit wird es für das Gerät nutzlos und eine Verbindung zum IoT Provider ist nicht mehr möglich.

Zertifikate aufs Gerät laden

Für eine erfolgreiche Kommunikation zwischen IoT Provider und Gerät werden folgende Dinge benötigt:

- Gerätezertifikat (zur Authentifikation gegenüber dem Provider)
- Private Key des Gerätezertifikates

- Root Zertifikat des IoT Providers (zur Verifikation des Servers durch das Gerät)

11.5. Gerät

Dieser Abschnitt befasst sich mit dem Geräteteil der Plattform. Als Gerät ist in diesem Fall das Pycom Entwicklergerät gemeint, das mit der IoT Plattform kommuniziert und Sensordaten liefert. Es fasst

Sensoren und Gateways zusammen, welche im produktiven Anwendungfall zum Einsatz kommen würden. Den Geräten wurde kein hoher Stellenwert in der Arbeit zugeteilt. Daher ist der Funktionsumfang beschränkt. Geräte können mit dem Server kommunizieren, Sensordaten auslesen und senden, sowie eine simple Konfiguration übernehmen.

11.5.1. Hardware

Als Hardware wurden die in der Evaluation beschriebenen Geräte von Pycom genutzt, jeweils bestehend aus einem Kommunikationsmodule (FiPy) und Sensorboard (PyTrack, PySens). Damit ist es möglich ein „echtes“ Gerät mit Sensoren zu simulieren und Sensordaten zu generieren.

FiPy

Das Kommunikationsmodul stellt verschiedene Schnittstellen bereit. In unseren Fall beinhaltet es WiFi, Bluetooth, LoRa, Sigfox und LTE-M. Davon wird im Prototyp jedoch aus Komplexitätsgründen nur das WIFI benutzt. Für den produktiven Einsatz müsste je nach Anwendungsfall die passende Technologie erprobt werden.

PySense und PyTrack

PySense und PyTrack sind Sensorboards, welche mit dem Kommunikationsmodul verbunden werden und so das Testgerät bilden. Die Boards enthalten jeweils unterschiedliche Sensoren, welche ausgelesen werden können.

PySense Board enthält 5 Sensoren:

- Umgebungslichtsensor
- Barometer
- Feuchtigkeitssensor
- 3 Axen 12-bit Beschleunigungssensor
- Temperatursensor

Zu Testzwecken wurden der Temperatur- und der Feuchtigkeitssensor ausgelesen.

Das PyTrack Board enthält 2 Sensoren:

- GPS
- 3 Axen 12-bit Beschleunigungssensor

Im Prototyp hat das Auslesen des GPS nicht korrekt funktioniert. Es wurde auch nach ausführlichen Tests keine Position vom Sensor geliefert. Der genaue Grund konnte nicht eruiert werden. Aus zeitlichen Gründen wurde somit auf die Verwendung dieses Boards im Prototyp verzichtet.

11.5.2. Software

Der Code der Geräte ist zu 100% in MicroPython, einer abgespeckten Version von Python, geschrieben. Ein Pycom Plugin für Visual Studio Code[3] ermöglicht das Aufspielen der Software auf dem Gerät über einen seriellen Port. Allgemein befinden sich das Plugin sowie die Firmware von Pycom noch in einer Betaphase. Deshalb kam es auch zu einigen Schwierigkeiten bei der Entwicklung aufgrund von Bugs (Probleme bei der Verbindung und Datenübertragung zum Gerät).

11.5.3. Konfiguration

Die Grundkonfiguration, welche für jedes Gerät gleich ist, wird mittels Konstanten festgelegt. Jedes Gerät hat zusätzlich eine Konfiguration, welche den eindeutigen Identifier (UUID) zuweist und die von Gerät zu Gerät variierenden Variablen,zum Beispiel für angeschlossene Hardware oder das zu verbindende WIFI, setzt.

```
1 # wifi configuration
2 WIFI_SSID = 'Pixel_1408'
3 WIFI_PASS = '12345678'
4
5 # client configuration
6 CLIENT_ID = 'f9ec0e44-cb58-41bf-af5d-3b8905ec3687'
7 BOARD_TYPE = 'sense' # 'track' or 'sense'
```

Diese Konfigurationen dienen als Default-Werte und können (derzeit) nicht durch die Plattform gesteuert werden. Grundsätzlich würden diese Einstellungen über die Provision API vorgenommen.

Beim Start wird die aktuelle von der Plattform gewünschte Konfiguration geladen. Das Gerät übernimmt alle Konfigurationen und bestätigt diese. Folgende Parameter haben eine Auswirkung auf das Verhalten:

- `device.isActive` definiert, ob Sensordaten zum Server gesendet werden.
- `device.interval` bestimmt das Intervall in Sekunden, in dem die Daten gesendet werden.

Alle anderen Parameter werden ignoriert.

11.5.4. IoT Provider Anbindung

Für den Prototyp wurde AWS IoT verwendet. Das Gerät verbindet sich beim Start automatisch mit AWS über die AWS IoT Device SDK mittels den hinterlegten Zertifikaten. Aktuell kann sich das Gerät nur mit diesem einen Provider verbinden. Für den produktiven Einsatz wäre eine flexiblere Variante

[3] https://code.visualstudio.com/

wünschenswert, was bedeutet, dass die Provision API alle nötigen Software-Bibliotheken und Parameter für den Verbindungsaufbau zur Verfügung stellen müsste. Dies ist mit dem aktuellen Stand des Prototyps nicht der Fall.

11.6. MQTT

MOTT ist das Protokoll, welches zur Kommunikation zwischen Plattform und Geräten verwendet wird. Es ist ein etabliertes Protokoll in der Welt der Internet of Things. Es ist ein Protokoll optimiert für langsame Datenraten, unstabile Verbindungen und wenig Rechenleistung. Weitere Informationen gibt es auf der offiziellen Webseite[4].

Publish/Subscribe

MQTT funktioniert nach dem Publish /Subscribe Prinzip. Es gibt verschiedene Nachrichtenkanäle, auf denen Nachrichten publiziert werden können. Es gibt einen Nachrichtenbroker, der als zentrale Anlaufstelle fungiert. Clients können die verschiedenen Kanäle abonnieren, um die darauf gesendeten Nachrichten zu erhalten. Zudem können sie auch selbst Nachrichten auf Kanäle publizieren.

Broker

Der Nachrichtenbroker ist zuständig für das Verwalten von Abonnenten und Kanälen (auch Topics genannt). Er leitet eingehende Nachrichten auf ein Topic an die Abonnenten weiter. Alle An- und Abmeldungen von Topics werden an den Broker gesendet.

Topics

Für Kommunikation zwischen Gerät und Plattform im Projekt, wurden verschiedene Topics definiert. In diesem Abschnitt werden die verschiedenen Topics erläutert und die Datenstruktur anhand eines Beispiels aufgezeigt.

11.6.1. Statusupdates

Alle Geräte senden regelmässig Statusupdates an den Server. Damit kann der User in der Plattform jederzeit den letzten Stand abfragen. Der Topic dafür ist `/status/{deviceId}`. Eine Statusnachricht

[4] http://mqtt.org/faq ## Funktionsweise

enthält Angaben über den Zustand des Gerätes, z.b. deren Verbindungsstatus, Batteriestand oder auch sonstige Betriebsdaten welche nicht von Sensoren stammen.

```
{
    "deviceUid":"device-abc",
    "timestamp": "2013-04-13T22:56:27.000+03:00",
    "type":"STATUS",
    "status": "connected",
    "battery": "50%",
    "uptime": "2547h"
}
```

11.6.2. Sensordaten

Sensordaten umfassen alle Daten, die von den einzelnen Sensoren kommen. Dies beinhaltet Temperatur, GPS, Feuchtigkeits- und weitere Werte. Diese Daten werden unter dem Topic `/data` versandt. Dieses Topic ist hierarchisch unterteilt in Geräte und dann in Sensortyp, wie zum Beispiel Temperatur oder Gps. Der finale Topic, auf dem das Gerät die Daten sendet, ist dann zum Beispiel `/data/{deviceId}/temp`.

Ein Datenpaket dieses Typs sieht folgendermassen aus:

```
{
    "deviceUid":"aa9c0799-b445-4941-9543-3a09822fd6dd",
    "timestamp": "2013-04-13T22:56:27.000+03:00",
    "type":"SCHEDULED",
    "data": [
        {
            "name":"abc-123",
            "timestamp":"2013-04-13T22:56:27.000+03:00",
            "value": 22
        },
        {
            "name":"abc-123",
            "timestamp":"2013-04-13T22:56:27.000+03:00",
            "value": 21
        }
    ]
}
```

Nachrichteninhalt:

- deviceUid: eine eindeutige Identifikation des Gerätes von welchem die Nachricht stammt, in Form einer UUID.
- timestamp: der Zeitpunkt, an dem das Paket versendet wurde.
- type: Die Art der Nachricht für die bessere Zuordnung (in diesem Fall ein geplanter Versand ohne speziellen Trigger)
- data: Die effektiven Werte der Sensoren in Form einer Liste.

Sensorwerte:

- name: Name des Sensors von welchen die Daten stammen.
- timestamp: Zeitpunkt, an dem der Wert registriert wurde.
- value: der effektive Wert.

11.6.3. Alarme

In der Konfiguration des Gerätes können verschiedene Bedingungen gesetzt werden, welche bei deren Eintritt einen Alarm auslösen. Diese Alarme werden auf einem separaten Topic und ohne Rücksicht auf sonstige Sendeintervalle gesendet. Alarme werden unter dem Topic `/alert` gesendet, welcher wieder weiter in Topics pro Gerät aufgeteilt ist. (`/alert/{deviceId}`)

Die Struktur der Nachricht unterscheidet sich nur in einem Feld von der Struktur der Sensordaten. Als zusätzliches Feld wird der Auslöser des Alarms mitgegeben. Bei Alarmnachrichten hat das Feld `type` den Wert ALERT.

```
{
    "deviceUid":"aa9c0799-b445-4941-9543-3a09822fd6dd",
    "timestamp": "2013-04-13T22:56:27.000+03:00",
    "type":"ALERT",
    "trigger": "Value over THRESHOLD 20",
    "data": [
        {
            "name":"abc-123",
            "timestamp":"2013-04-13T22:56:27.000+03:00",
            "value": 22
        }
    ]
}
```

11.6.4. Last will

Wenn sich das Gerät beim Nachrichtenbroker anmeldet, kann es einen `Last will`-Topic und eine `Last will`-Nachricht hinterlegen. Diese wird auf dem Topic publiziert, sobald das Gerät unerwartet die Kommunikation mit dem Broker abbricht.

/status, /status/{deviceID}

```
{
    "deviceUid":"aa9c0799-b445-4941-9543-3a09822fd6dd",
    "type":"LAST_WILL",
    "status": "lost_connection"
}
```

11.7. Konfiguration

Über die Plattform ist es möglich, Geräte zu konfigurieren. Dies ist mittels Key-Value Paaren gelöst. Die Funktionsweise wird in diesem Kapitel beschrieben.

11.7.1. Aufbau

Um etwas Ordnung in die offene Struktur von Key-Value Paaren zu bringen, wurde folgender Standard definiert. Konfigurationen, die das Gerät betreffen, starten mit „device.“, die Konfiguration einzelner Sensoren starten mit „sensors[x].“. Die Interpretation der Werte ist dem Gerät überlassen. Die Plattform unternimmt keine Überprüfung bezüglich dem Inhalt der Key-Value Paare. Einzig eine Grundkonfiguration, vordefinierte Schlüssel, werden forciert.

11.7.2. Konfigurationsablauf

Das Konfigurationsmanagement zwischen Plattform und Geräten wurde über die Shadow-States von AWS gelöst. Dies ist Best-Practise und wird bei allen IoT Anbietern in ähnlicher Weise angeboten. Im Hintergrund funktioniert die Konfiguration mittels MQTT-Nachrichten. Deshalb kann das gleiche System auch mittels einer eigenen Lösung implementiert werden, falls kein IoT Provider benutzt wird. Die Funktionsweise ist wie folgt: MQTT Nachrichten mit Statusänderungen werden auf (in diesem Fall von AWS) festgelegte Topics publiziert. Geräte rapportieren ihren aktuellen Status mittels dem „reported“ Keyword.

```
{
    "state": {
        "reported": {
            "device.interval": 10
        }
    }
}
```

Konfigurationsänderungen werden von der Plattform mittels dem „desired“ Keyword definiert.

```
json
{
    "state": {
        "desired": {
            "device.interval": 5
        }
    }
}
```

Der IoT Provider berechnet bei jeder Änderung die verbleibende Differenz, hier als Delta bezeichnet, und publiziert diese.

```
{
    "state": {
        "desired": {
            "device.interval": 5
        },
        "reported": {
            "device.interval": 10
        },
        "delta": {
            "device.interval": 5
        }
    }
}
```

Das Gerät wiederum bestätigt die Konfiguration.

```
{
    "state": {
        "reported": {
            "device.interval": 5
```

```
5             }
6         }
7     }
```

Danach ist der Prozess beendet.

Die grundlegende Funktionsweise und das Konzept, mittels „reported“, „desired“ und „delta“ die Zustände abzubilden, ist bei allen grossen IoT Providern gleich. Es gibt kleinere Unterschiede im Nachrichtenformat und den Topics. Alle beschränken sich aber, wie wir, auf Key-Value Paare.

11.8. Client

Der Riot-Webclient nutzt die von Riot zur Verfügung gestellte RESTful HTTP API. Er bietet für den Benutzer von Riot eine einfach bedienbare grafische Benutzeroberfläche an. Der Webclient wurde mit Hilfe von React als Single Page Application umgesetzt und kann separat von Riot deployed werden. Die folgenden Kapitel beschreiben die Umsetzung und den grundsätzlichen Aufbau des Client.

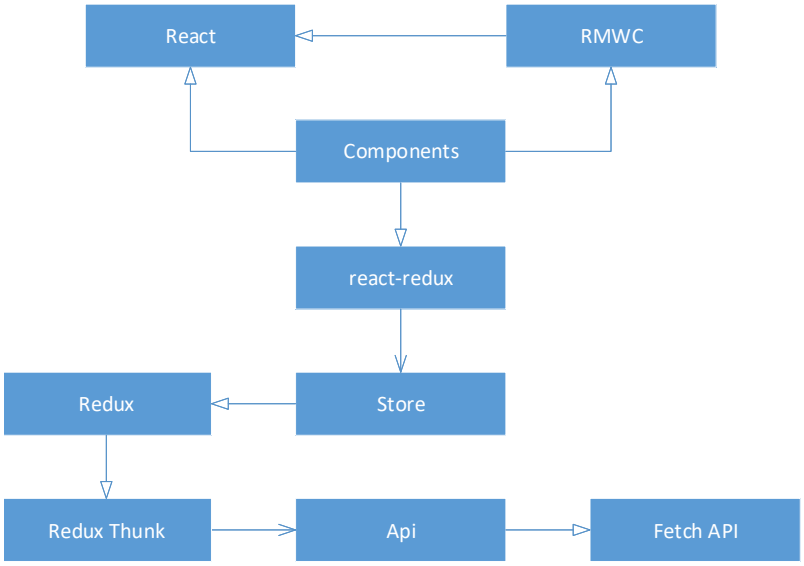


Abbildung 11.5.: Riot-Webclient Paketdiagramm

11.8.1. Ansprechen der Riot-API

Um die von Riot angebotene RESTful HTTP API anzusprechen, greift die Client-Implementation auf die Fetch API zurück. Diese weist eine gewisse Ähnlichkeit zu XMLHttpRequests auf, stellt aber ein

mächtigeres und flexibleres Featureset zur Verfügung. Fetch-Funktionen sind asynchron und geben statt eines direkten Resultats einen Promise als Rückgabewert zurück[4].

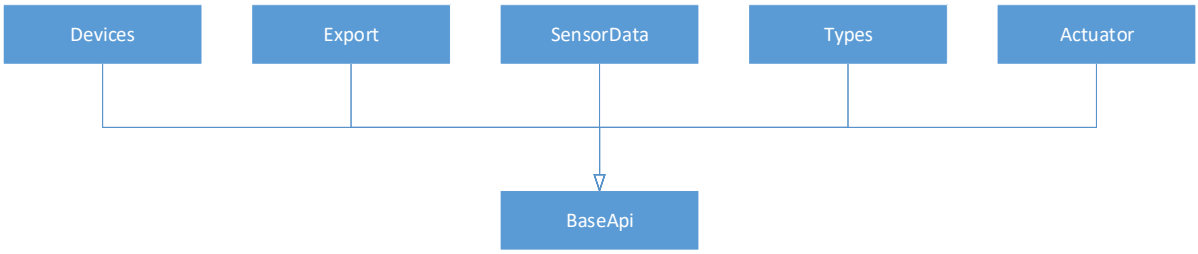


Abbildung 11.6.: Riot-Webclient API

Jegliche Nutzung der Fetch API wurde zentral in der BaseApi implementiert. Sie bietet alle grundsätzlichen Funktionen an, um HTTP-Requests an die Riot-API abzusetzen. Somit gehen alle vom Client, nach dem initialen Laden, abgesetzten Anfragen an Riot durch diese Basisfunktionen. Sie übernehmen das Setzen der HTTP-Header und überprüfen die Antwort auf einen erfolgreichen HTTP-Response-Code. Bei Anfragen, welche eine Authentifizierung benötigen, wird automatisch das Json Web Token gesetzt. Dieses wird nach einem erfolgreichen Login des Benutzers im SessionStorage des Browsers aufbewahrt und wird jeweils von der BaseApi von dort geholt.

Für alle von der Riot-API angebotenen Endpunkte existiert ein Gegenstück in der API-Implementation des Clients. Diese implementieren die von Riot definierten Schnittstellen. Sie übersetzen die vom Client gegebenen Parameter zur benötigten JSON-Payload und setzen den Endpunkt der API, welcher angefragt werden muss. Die BaseApi stellt Funktionen zur Verfügung, welche von den Schnittstellen-Implementationen bei Bedarf genutzt werden können, um die Antwort von Riot zu verarbeiten, z.B. um die JSON-Payload zu parsen.

11.8.2. State-Handling

Die Implementation als Single Page Application verlangt ein eigenes State Management im Client. Da dies auch mit Hilfe von React sehr schnell komplex und unübersichtlich werden kann, kommt Redux zum Einsatz. Dies ermöglicht das Halten eines zentralen States als Single Source of Truth, so dass der Client sich stets des kompletten Status bewusst ist.

Actions und Reducers

Redux unterscheidet zwischen Actions und Reducers, um Änderungen im globalen State zu bewirken. Actions tragen die Nutzlast, welche benötigt wird, um den State zu erneuern und werden mit einem

Typ ausgewiesen, der angibt, welchen Ursprung die Nutzlast hat[13]. Reducers spezifizieren, wie sich der State als Antwort auf die Actions verändert[15].

In der Riot-Client Implementation sind die Actions und Reducers zur besseren Übersicht voneinander getrennt. Es gibt jedoch zu jeder Action-Definition ein Reducer-Equivalent. Ein Action-Reducer-Paar definiert, bis auf wenige Ausnahmen, jeweils das State-Handling für eine Container-Komponente oder eine Funktionalität. Reducer sind als Pure-Funktionen implementiert worden. Es gibt daher keine Reducer, welche unabhängig voneinander das selbe Teilstück des States bewirtschaften. Der initiale State wird daher auch teilweise in dem Reducer definiert, welcher auch Änderungen daran vornehmen darf.

Um asynchrone Actions definieren zu können wird Redux-Thunk eingesetzt. Dies ermöglicht es, die asynchronen Funktionen der API-Implementation in einfache Redux-Actions zu integrieren und auf den Request-Aufruf sowie dessen Resultat individuell zu reagieren. Die Funktion `fetchAsync` abstrahiert dabei das Handling von Requests an die Riot-API und kann bis auf wenige Ausnahmen verwendet werden. Sie übernimmt das Handling, um den State entsprechend zu aktualisieren wenn der Request lädt, beantwortet wurde oder fehlerhaft ist. Sie nimmt als Parameter Callbacks entgegen, um Requests abzusetzen, um auf erfolgreiche und fehlerhafte Responses zu reagieren und entsprechende Meldungen, welche dazu angezeigt werden sollen.

Eine der Ausnahmen, welche nicht mit `fetchAsync` umgesetzt werden konnte, ist das Laden von Sensordaten. Sensordaten werden durch einen Timeout alle 10 Sekunden aktualisiert. Die Funktionsweise dieser AutoLoad-Funktion stellt einen Spezialfall dar und kann vom Benutzer zusätzlich pausiert und aktiviert werden. Diese erweiterte Komplexität macht es nötig, diese Funktionalität abseits von `fetchAsync` zu implementieren.

Integration von Redux

Für die Integration von Redux in die React-Komponenten kommt `react-redux` zum Einsatz. Der Redux-State wird dabei durch eine Provider-Komponente im React-Komponenten-Baum an möglichst hoher Stelle integriert. Dies ermöglicht es, Teile des Redux-State über die `connect`-Funktion direkt als properties in React-Komponenten einzubinden. Ebenso ist es möglich, Redux-Actions den Komponenten als ausführbare Funktionen zur Verfügung zu stellen.

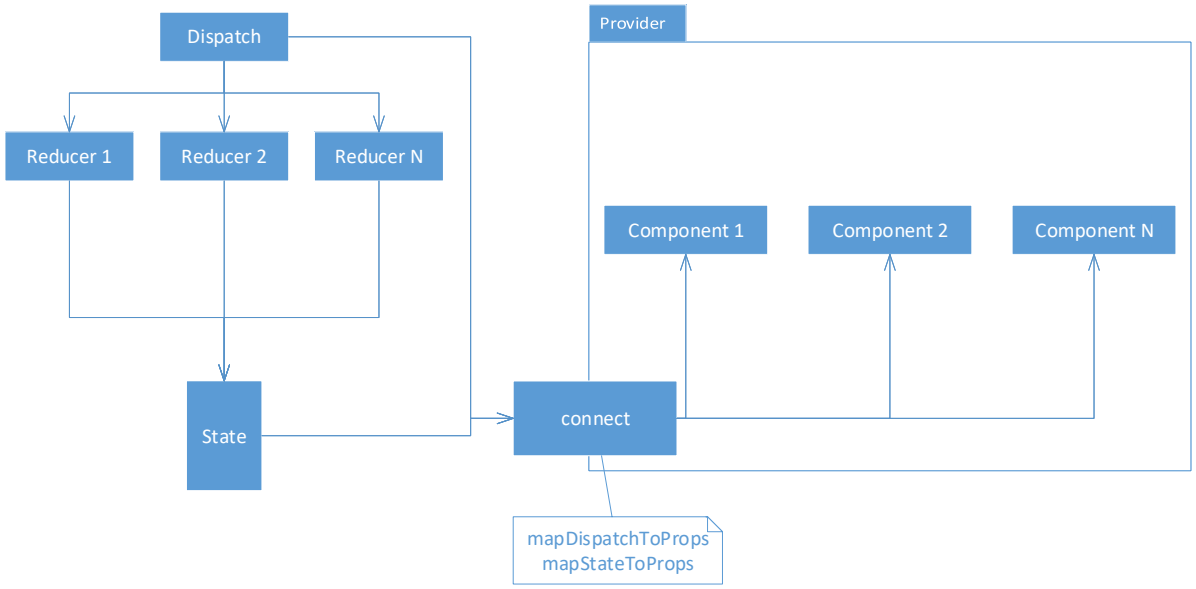


Abbildung 11.7.: Funktionsweise von react-redux

Anstatt das jede Komponente, wie ansonsten in React üblich, seinen eigenen State managed, wird dieser nun vom zentralen Redux-State entnommen. Dies reduziert einerseits den für das State-Handling benötigten Code innerhalb der React-Komponenten auf ein Minimum, andererseits werden auch potentielle Fehlerquellen verhindert, bzw. können Fehler im State-Handling durch die zentrale Implementierung schneller gefunden werden.

Bei der Umsetzung wird das Konzept der Separation von Container- und Presentational-Komponenten benutzt[3][14]. Es wird daher unterschieden zwischen wenigen smarten, mit dem State verbundenen Komponenten, und dummen, wiederverwendebaren Komponenten, welche nur zur Darstellung benutzt werden. Dadurch ist eine sehr gute Trennung zwischen der Darstellung und dem State-Handling, der Clientlogik, möglich.

11.8.3. Umsetzung der Benutzeroberfläche

Durch die Umsetzung mit React ist es möglich, die Benutzeroberfläche in Komponenten aufzuteilen. Dies erleichtert das Einbinden von UI-Frameworks und anderen Bibliotheken. In den folgenden Kapiteln wird auf die wichtigsten Strukturen und verwendeten Bibliotheken eingegangen.

Navigation

Im Client wird die React-Router-Bibliothek benutzt. Dies ermöglicht ein dynamisches Routing innerhalb der Applikation aufgrund von Änderungen am State[12]. Dem Benutzer wird dadurch das Verhalten einer gewöhnlichen Webseite, inkl. dynamischer Urls in der Adressleiste des Browsers simuliert, ohne das ein Roundtrip zum Server nötig ist. In Wirklichkeit erfolgt im Client nur der Austausch von Komponenten.

Der Riot-Client verwendet eine zentrale Navigationsstruktur, welche als JavaScript-Objekt angelegt ist. Durch die zentrale Definition der Navigationsstruktur ist es möglich, die Menüs und das Routing des Clients automatisch zusammenzustellen. Für den Entwickler ist sofort ersichtlich, welche Routen gültig sind. Zur Integration neuer Komponenten muss lediglich die Struktur erweitert werden. Die Struktur ist aufgeteilt in drei Teile.

Alle Routen im Main-Teil werden im Hauptmenü als Menüpunkte dargestellt. Routen im Additional-Teil werden im Hauptmenü als zusätzliche Menüpunkte am unteren Rand dargestellt. Routen im Blind-Teil werden in keinem Menü des Clients dargestellt. Sie werden lediglich, wie natürlich auch die Routen der anderen beiden Teile, beim React-Router registriert, damit dieser das Routing übernehmen kann.

Listing 11.1: Navigationsstruktur des Clients

```
1  const Structure = {
2    main: [
3      { id: "HomeItem", name: "Devices", to: "/", icon: "device_hub",
           component: Devices },
4      { name: "Device Types", to: "/types", icon: "category", component:
           Types },
5      { name: "Export", to: "/export", icon: "get_app", component:
           ExportData },
6    ],
7    blind: [{
8        // Link set explicit on Fab in DeviceView
9        to: "/devices/new", redirectTo: "/", component: NewDevice
10     },
11     {
12       // Link set explicit in DeviceCard
13       to: "/devices/:id", redirectTo: "/", component: DetailDevice
14     },
15     {
16       // Link set explicit in DeviceCard
17       to: "/devices/:id/config", redirectTo: "/devices/:id", component:
             ConfigDevice
18     },
19     {
```

```
20        // Link set explicit in DeviceCard
21        to: "/devices/:id/cert", redirectTo: "/devices/:id", component:
              CertDevice
22      },
23      {
24        // Link set explicit in DeviceCard
25        to: "/devices/:id/data", redirectTo: "/devices/:id", component:
              DeviceData
26      },
27      {
28        // Link set explicit on Fab in TypesView
29        to: "/types/new", redirectTo: "/types", component: NewType
30      },
31      {
32        // Link set explicit in TypeCard
33        to: "/types/:id", redirectTo: "/types", component: DetailType
34      },
35      {
36        // Link set explicit in TypeCard
37        to: "/types/:id/edit", redirectTo: "/types", component: EditType
38      },
39    ],
40    additional: [{
41      name: "About", to: "/about", icon: "info", component: About
42    }]
43  };
```

Material Web Components

Die Benutzeroberfläche des Clients ist nach den Vorgaben von Googles Material Design umgesetzt. Mit den Material Web Components stellt Google selber ein Framework zur Verfügung, welches die Material Design Vorgaben als HTML-Komponenten implementiert. Mit Hilfe der React-Wrapper-Implementierung von James Friedman[5] können diese ohne grossen Aufwand in React verwendet werden.

Chart.js

Um Graphen der Sensordaten darzustellen, wird Chart.js verwendet. Diese Opensource-Bibliothek erlaubt eine einfache Anpassung an das eigene Design. Durch die Verwendung von HTML5 Canvas im Hintergrund besitzt sie eine sehr gute Performance zum Darstellen der Daten.

Redux Forms

Die verschiedenen Formulare wurden mit Hilfe von Redux Forms implementiert. Dies übernimmt nach der Initialisierung jegliches State-Handling der Formulare und dessen Eingabefelder. Durch die starke Integration des State-Managments der Formulare in den Redux-State reduziert sich der Entwicklungsaufwand für Formulare in einer React-Applikation signifikant.

11.8.4. Bundling mit Webpack

Webpack wird als Bundler für den Client verwendet. Dies ermöglicht es, alle Abhängigkeiten der Applikation in einem Paket zusammenzufassen. Webpack baut dazu einen Abhängigkeits-Graphen auf und bündelt auf dessen Grundlage alle benötigten Abhängigkeiten. Im resultierenden Paket sind dadurch nur noch benötigte Codestücke und Assets wie Bilder und Schriftarten enthalten.

Die in Less geschriebenen CSS-Styles werden dabei in reines CSS umgewandelt. Das Javascript wird mit Polyfills nachgerüstet. Dadurch wird der Javascript-Code mit Workaround-Implementation von neueren Funktionen nachgerüstet. Damit wird sichergestellt, dass der Client auch in älteren Browsern lauffähig ist.

11.9. Deployment

Das Kernsystem von Riot ist dank seiner Umsetzung in Java und der Verpackung als Fat JAR relativ einfach, sowohl auf Windows- als auch auf Linuxsystemen, installier- und betreibbar. Einzige Bedingung ist natürlich, dass das Java Runtime Environment (JRE) vorhanden ist. Eine gewisse Herausforderung sind dagegen eher die Abhängigkeiten zu zusätzlichen Services.

Das Aufsetzen der hier beschriebenen Umgebungen, inkl. ihrer Abhängigkeiten, wird in der Installationsanleitung im Anhang genauer beleuchtet. Die folgenden Kapitel dienen als Ergänzung dazu und sollen auch Entscheide, welche im Bereich des Deployments getan werden mussten, begründen.

11.9.1. Entwicklungsumgebung

Um den Installationsaufwand für eine vollständige Entwicklungsumgebung möglichst gering zu halten, wurde ein spezielles Developer Spring-Profil angelegt. Ist dieses aktiv, lässt sich das Kernsystem ohne zusätzliche Installation von externen Services starten und ähnlich seinem produktivem Umfeld verwenden. Dadurch wird beim Systemstart eine embedded H2-Datenbank und ein embedded UnboundID-LDAP-Server gestartet. Die H2-Datenbank wird als Datei persistiert und im Server-Modus gestartet.

Somit ist sie für den Entwickler mit den entsprechenden Tools jederzeit einsehbar und die eingetragenen Daten überstehen auch einen Neustart des Kernsystems. Der verwendete UnboundID-LDAP-Server wird jeweils beim Start mit einer hinterlegten LDIF-Datei initialisiert. Es stehen also jeweils die darin definierten Gruppen und Benutzer zur Verfügung. Vom IoT-Service wird die Dummy-Implementation verwendet.

11.9.2. Systemtests

Für Unit- und Systemtests wurde ebenfalls ein eigenes Spring-Profil angelegt um das Kernsystem beim Start entsprechend zu konfigurieren. Dadurch sind die Tests sowohl auf der Entwicklermaschine, wie auch auf dem Build-Server automatisiert ohne externe Abhängigkeiten durchführbar. Im Gegensatz zum Entwickler-Profil gilt hierbei die Anforderung, dass jeder einzelne Testfall die selben Vorbedingungen antreffen muss. Aus diesem Grund wird die H2-Datenbank komplett im Arbeitsspeicher geführt, um sicher zu gehen, dass nach einem Testfall keine alten Daten zurückbleiben und womöglich den nachfolgenden Testfall behindern. UnboundID kommt ebenfalls zum Einsatz und wird über eine eigene LDIF-Datei mit Testbenutzern bestückt.

11.9.3. Testumgebung

Es wurden zwei Umgebungen für Continuous Integration und Continuous Deployment aufgesetzt. Dazu dienten die zwei virtuellen Maschinen welche im Kapitel Infrastruktur des Projektplans beschrieben wurden.

Auf der VM der HSR wurde dabei nur das Kernsystem entsprechend der Installationsanleitung aufgesetzt. Dies geschah einerseits aus der Überzeugung, dass die Maschine von der Leistung her als zu schwach für Dataflow empfunden wurde und andererseits daher, dass durch die restriktive Absicherung die MQTT-Ports gesperrt waren.

Auf der VM der Innofield AG wurde dagegen das Gesamtsystem aufgesetzt. Sie bot genug Leistung und war auch voll konfigurierbar bezüglich der offenen Ports in der Firewall. Neben dem Kernsystem wurde auch die Dataflow-Umgebung, in der lokalen Server-Variante, aufgesetzt. Damit war es möglich die einzelnen Dataflow-Anwendungen manuell zu testen und entsprechend, bei richtiger Konfiguration, auch die gesamte Anwendung von den Geräten bis zur Clientanwendung.

11.9.4. Cloud Foundry

Dass das Deployment der einzelnen Teilsysteme als eigene Services auf der gleichen Maschine funktionierte, konnte mit dem Deployment auf der Testumgebung bewiesen werden. Da dies jedoch eine eher

unwahrscheinliche Deployment-Variante für die Praxis bleiben würde, wurde versucht das System verteilt auf einem PaaS-Anbieter aufzusetzen. Die Wahl fiel dabei auf die Swisscom Application Cloud[5], eine von der Foundation zertifizierte Cloud Foundry Umgebung[2]. Einerseits war bereits Erfahrung mit Cloud Foundry vorhanden. Andererseits bietet der PaaS/SaaS-Ansatz eine andere Perspektive auf die Deployment-Möglichkeiten als zum Beispiel ein IaaS-Ansatz, welcher eher ein Deployment auf verschiedenen, verteilten Maschinen simulieren würde.

Das grösste Problem beim Aufsetzen von Riot auf der Cloud Foundry Umgebung bereitete sicherlich die von der Swisscom zur Verfügung gestellte Datenbank. Beziehungsweise es wurde bei der Umsetzung zu wenig auf mögliche Unterschiede in verschiedenen Datenbanklösungen geachtet. Bis zu diesem Moment wurde lediglich mit PostgreSQL- und H2-Datenbank gearbeitet. Die Swisscom bietet dagegen als relationelle Datenbanklösung nur MariaDBs aus einem Galera-Cluster an[9]. Dies hat zur Folge, dass entgegen der ersten Vermutung nicht die standartmässigen MariaDB-Treiber lauffähig sind, da der Galera-Cluster auf die InnoDB Datenbank-Engine setzt[10]. Hierbei muss daher auf die MySQLInnoDB-Treiber zurückgegriffen werden.

Ein weiteres Problem, welches sich durch die Verwendung von MariaDB zeigte, waren die verwendeten Datentypen in der Datenbank. Während sowohl PostgreSQL wie auch H2 das Abspeichern von UUIDs nativ unterstützen, führte dies bei der eingesetzte MariaDB-Variante zu Fehlern, die das Kreieren des Schemas auf der Datenbank verhinderten. Dieses Problem konnte gelöst werden, in dem das Datenbank-Schema so abgeändert wurde, dass die UUID, welche zur eindeutigen Identifikation von Geräten notwendig sind, in Zukunft nur noch als Varchar in der Datenbank abgespeichert werden.

11.10. Loadtests

11.10.1. MQTT Tests

Im produktiven Einsatzt wird wie im Kapitel Anforderungen beschrieben, mit 20’000 Geräten, welche im Minutentakt Nachrichten senden, gerechnet. Dies resultiert in einem Nachrichtenfluss von ca. 333 Nachrichten pro Sekunde. Um zu testen, dass diese Menge an Nachrichten von der Plattform bewältigt werden kann, wurden Loadtests mit simulierten Geräten durchgeführt. Der Test soll sowohl den IoT Provider, als auch die Echtzeit Datenverarbeitung testen.

[5] https://console.developer.swisscom.com/

Testaufbau

Der Test ist nach dem Beispiel[6] von ConcurrencyLabs aufgebaut. Er basiert auf Locust[6], einem open source Loadtesting-Tool. Dieses simuliert, mit der Hilfe von MQTT-Locust[7], eine beliebige Anzahl an Geräten. Der Code von Mqtt-Locust wurde so angepasst, dass er die Geräte der Plattform simuliert. Jedes virtuelle Gerät schickt durchschnittlich einen Sensorwert pro Sekunde. Die gesendeten Daten sind identisch mit denen von realen Geräten. Die Requests werden von einem, von der Plattform unabhängigen, Server generiert und an AWS gesendet. Es wurden Tests mit 10, 50, 100, 125, 150, 200 und 330 Geräten durchgeführt. Die gesamte Plattform lief dabei auf einem Entwicklungsserver mit 6GB RAM.

Resultate

Gemessen wurden diverse Metriken:

- Gesendete Nachrichten von Locust
- Eingehende Nachrichten bei AWS
- Ausgehende Nachrichten von AWS
- In die Datenbank geschriebene Sensorwerte

Die untenstehende Tabelle zeigt die Resultate der durchgeführten Tests.

Tabelle 11.6.: Testresultate MQTT Loadtest

Msg/Sek.	Locust Publish	AWS Inbound	AWS Outbound	Database Inserts	Verlust	Erfolg (%)	Verlust (%)
10	468	468	468	468	0	100.00%	0.00%
50	3314	3314	3314	3314	0	100.00%	0.00%
100	9971	9971	9971	9971	0	100.00%	0.00%
125	9890	9890	8717	8717	1173	88.14%	11.86%
150	9674	9674	7219	7219	2455	74.62%	25.38%
200	19712	19712	10991	10991	8721	55.76%	44.24%
330	101358	101358	35784	35784	65574	35.30%	64.70%

[6] https://locust.io/
[7] https://github.com/concurrencylabs/mqtt-locust

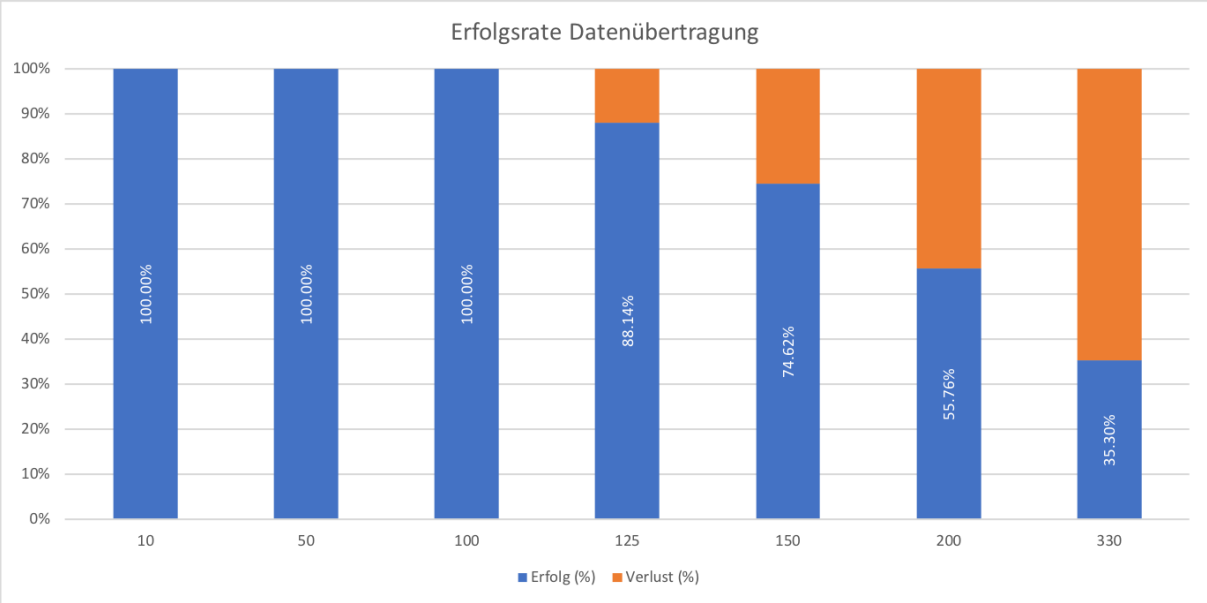


Abbildung 11.8.: Erfolgsraten Loadtests

Die Grösse einer Nachricht belief sich auf 236 bytes. Die Tests haben gezeigt, dass ab einem Durchsatz von 100 Nachrichten pro Sekunde nicht mehr alle Werte in der Datenbank landen. Bei genauer Betrachtung stellt man fest, dass die Nachrichten beim IoT Provider hängen bleiben. Mehr dazu im folgenden Abschnitt. Alle von AWS ausgesandten Nachrichten wurden auch in die Datenbank gespeichert. Eine Überwachung der Ressourcen hat gezeigt, dass der Server weder an RAM noch CPU Grenzen gelangte. Der durchschnittlich belegte Speicher war 4.2 von 6 GB.

Probleme

Wie in den Resultaten ersichtlich, beginnt AWS bei mehr als 100 Nachrichten pro Sekunde, überzählige Nachrichten nicht weiterzuleiten. Die Nachforschung zeigte, dass AWS IoT die Anzahl MQTT Nachrichten pro Sekunde pro Verbindung limitiert[8]. Deshalb wurde der Testaufbau genauer untersucht. Es stellte sich jedoch heraus, dass Locust die Geräte korrekt simulierte und auch für jedes eine neue Verbindung aufbaute. Der Fehler lag also nicht bei den Geräten. Damit ein Limit pro Zertifikat oder IP ausgeschlossen werden konnte, wurden noch weitere Tests mit zwei Server und unterschiedlichen Zertifikaten sowie auch MQTT Topics erfolglos durchgeführt. Da die 100 Nachrichten pro Sekunde genau die von AWS vorgegebene Limitierung pro Verbindung darstellt, war es naheliegend, dass dies die Ursache hinter dem Problem sein musste. Schlussendlich wurde das Problem gefunden. Es wurde nicht in Betracht gezogen, dass diese Limits auch für die ausgehenden Nachrichten gelten. Da die Plattform im Prototyp

[8] https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_iot

nur mittels einer MQTT Source im Dataflow mit AWS verbunden ist, welche auf ein Topic für alle Geräte hört, besteht auch nur eine Verbindung. Laut AWS werden pro Verbindung maximal 100 Nachrichten pro Sekunde versendet, daher die Limitation.

Lösungsansatz

Zur Lösung des oben beschriebenen Problems in der produktiven Umgebung gibt es mehrere Ansätze.

Zum einen könnte die Plattform mehrere Verbindungen zum IoT Provider aufrechterhalten, wenn der erforderliche Datendurchsatz wächst.
Dies kann z.B. durch Aufteilung von Geräten in Gruppen mit eigenen Topics geschehen. Die Plattform baut dann für jede Gruppe eine Verbindung auf. Natürlich kann diese Lösung auch dynamisch implementiert werden, ohne dass der Benutzer sich um die Einteilung in Gruppen kümmern muss. Andererseits könnten Nachrichten beim IoT Provider mittels Regeln gebündelt und dann weiter an die Plattform geschickt werden. Dies verringert den Durchsatz enorm. Auch auf Stufe der Geräte könnten die Daten gebündelt werden, wie es derzeit auch schon im Nachrichtenformat vorgesehen ist.

Mit einer eigenen Implementierung des IoT Providers könnte das Problem natürlich auch gelöst werden. Zu beachten gibt es dort jedoch, dass die Limits bei allen Anbietern existieren und teilweise sogar tiefer sind[1][11]. Eine eigene Implementierung des Message Brokers hat zusätzlich noch das Problem mit gleichzeitigen Verbindungen. Während AWS ein Limit von 500'000 simultanen Geräten hat, welches auf Anfrage sogar erhöht werden kann, unterstützt der populäre Message Broker Paho[9] nur 1000 gleichzeitige Verbindungen (was in der Realität ca. 340 Geräten entspricht)[8]

11.10.2. Client Test

Zusätzlich zum Loadtest von Dataflow wurde auch untersucht, wie sich der Client mit den grösseren Datenmengen verhält. Die API zeigte keine Probleme beim Umgang mit vielen Records. 50'000 Sensorwerte waren in 1.5 Sekunden geladen. Der Export dauerte ca. 2 Sekunden. Beim Client gab es bemerkbare Performanceprobleme bei der Darstellung von mehr als 1'000 Einträgen. Bei mehr als 10'000 Einträgen war die Anwendung nicht mehr akzeptabel bedienbar. Für die Visualisierung von grossen Datenmengen müssten bei der Implementation mit React noch einige Optimierungen vorgenommen werden. Zusätzlich müssen im Client API Mechanismen implementiert werden, um Daten effizienter bereitstellen zu können (Pagination, Queries, Ausdünnen der Daten).

[9] https://www.eclipse.org/paho/

12. Betriebskosten

Die Betriebskosten beschreiben die laufenden Kosten, welche für den Betrieb des Gesamtsystems anfallen. Der Kostenpunkt Wartung und Personalkosten werden hier bewusst weggelassen. Die Betriebskosten setzen sich aus den Kosten fürs Webhosting und Storage sowie den Kosten für den IoT-Service zusammen. In der folgenden Tabelle werden die Gesamtkosten der zwei während der Arbeit erprobten Deployment-Varianten verglichen. Der IoT-Service wurde während der Arbeit auf Basis von AWS IoT aufgebaut. In beiden Varianten wird daher mit den zu erwarteten Kosten bei AWS IoT gerechnet. Basis der Kostenberechnung stellt die maximal zu erwartende Belastung, 20'000 registrierte Geräte und 1'440 Messages pro Gerät pro Tag (1 Message pro Gerät pro Minute), während des Betriebs dar.

Tabelle 12.1.: Vergleich der Betriebskosten

Deployment Variante	Webhosting/Storage	IoT-Service	Gesamtkosten
Virtual Private Server	ca. 500 (CHF/Monat)	ca. 1'191 (CHF/Monat)	ca. 1'691 (CHF/Monat)
Cloud Foundry	12'396 (CHF/Monat)	ca. 1'191 (CHF/Monat)	ca. 13'587 (CHF/Monat)

12.0.1. IoT-Service

Der Hauptanteil der gesamten Betriebskosten steuern die IoT services bei. Die Kosten skalieren grundsätzlich, unabhängig von der Wahl der Serviceanbieter, mit der Anzahl der registrierten Geräte.

Der IoT-Service wurde während der Arbeit auf Basis von AWS IoT aufgebaut. Für den produktiven Einsatz mit bis zu 20000 registrierten Geräten und 1440 Messages pro Gerät pro Tag ergeben sich laufende monatliche Kosten von unter CHF 1200. Die detaillierte Kostenaufstellung ist in der Evaluation zu den IoT-Anbietern im Anhang einsehbar.

12.0.2. Webhosting und Storage

Virtual Private Server

Es ist möglich, die komplette Plattform auf einem einzigen Server zu betreiben. Dieser Ansatz wurde mit Hilfe eines Virutal Private Server umgesetzt, welcher uns von der Innofield AG[1] zur Verfügung gestellt wurde. Die Ausstattung der Server sind im Projektplan im Kapitel Infrastruktur beschrieben. Die Installation erfolgte auf Grundlage der entsprechenden Variante in der Installationsanleitung.

Die Kostenberechnung erfolgt bei Innofield dynamisch nach Auslastung. Es können weitere Optionen zur Skalierung und Hochverfügbarkeit hinzugefügt werden. Für den uns zur Verfügung gestellten Server ohne erweiterte Optionen, fallen in einer maximalen Auslastung Kosten von ca. CHF 500 pro Monat an.

Cloud Foundry

Der Cloud-Foundry-Ansatz wurde auf der Swisscom Application Cloud[2] umgesetzt. Alle folgenden Kostenberechnungen erfolgen daher auf Grundlage der von Swisscom erhobenen Preise. Die Installation erfolgte auf Grundlage der entsprechenden Variante in der Installationsanleitung.

Die Berechnung der Kosten erfolgt auf Basis des benötigten Arbeitsspeichers der Applikationen. Ein Monat wurde dabei als 30 Kalendertage lang definiert. Die zusätzlich verwendeten Services wie Datenbanken und RabbitMQ werden zusätzlich berechnet.

Tabelle 12.2.: Kostenberechnung Cloud Foundry

Applikation/Service	monatliche Kosten
Riot-Client	6 CHF
Riot-Kernsystem	48 CHF
Spring Cloud Data Flow	48 CHF
MQTT-Source	24 CHF
Json-Tranformer	24 CHF
Persistence-Sink	24 CHF
MariaDB mit 30GB (Riot)	120 CHF
MariaDB (Data Flow)	6 CHF

[1] https://innofield.com/
[2] Swisscom Application Cloud: https://developer.swisscom.com/

Applikation/Service	monatliche Kosten
RabbitMQ	12’096 CHF
Total	**12’396 CHF**

Die Data Flow Applikationen laufen auf Cloud Foundry als eigenständige Container. Daher werden diese in der Kostenberechnung auch einzeln aufgeführt. Die Datenbank für die Riot-Plattform wurde mit einem Speicher von 30 GB berechnet. Dies entspricht demselben Speicherplatz, welcher auch auf dem Virtual Private Server für Datenbanken zur Verfügung stehen würde. Auf Cloud Foundry würde dies jedoch dynamisch entsprechend des tatsächlich benötigten Speicherplatzes berechnet.

Der grösste Ausreisser in der Kostenberechnung stellt RabbitMQ dar. Riot produziert pro MQTT-Message zwei Messages auf RabbitMQ. Aufgrund des relativ hohen Preises von 0.000007 CHF/Message und monatlichen 1’728’000’000 Messages ist dies jedoch nicht vermeidbar.

13. Ergebnisdiskussion

13.1. Auswertung

Das Kapitel Auswertung fasst die Ergebnisse der durchgeführten Tests zusammen und zeigt die Differenz zwischen dem SOLL und dem IST Zustand.

13.2. Zielerreichung

Im Folgenden werden die Umsetzung mit der Aufgabenstellung und den Zielsetzungen verglichen.

Die Zielsetzungen leiten sich primär aus der Aufgabenstellung und sekundär aus den nicht funktionalen Anforderungen ab.

Tabelle 13.1.: Zielsetzungen abgeleitet von der Aufgabenstellung

Beschreibung der Zielsetzung ausgehend von der Aufgabenstellung	Status
Lifecycle Management von IoT Geräten ermöglichen	teilweise erfüllt
IoT Geräte konfigurieren können	erfüllt
Bedienung über ein Web UI	erfüllt
Kommunikation der Geräte und Komponenten über sichere Kanäle mit authentifizierten und autorisierten Benutzern	erfüllt
Datenanalysen ermöglichen und visualisieren (Priorität 2)	teilweise erfüllt
Flexibles Deployment (Priorität 2)	erfüllt
Export von Daten ermöglichen (Priorität 3)	erfüllt
Schnittstellen zu bestehenden ERP schaffen (Priorität 3)	teilweise erfüllt

13.2.1. Bemerkung zu „Lifecycle Management von IoT Geräten ermöglichen“

Unter der Vorbedingung, dass manuell vorgängig beim IoT Provider ein CA Zertifikat hinterlegt wurde und auf den Geräten ein davon signiertes Client Zertifikat korrekt installiert wurde, ist das Lifecycle Management vollständig über die IoT Plattform abgebildet.

13.2.2. Bemerkung zu „Datenanalysen ermöglichen und visualisieren“

Sensor Rohdaten können visualisiert werden sowie exportiert werden. Eine weiterführende Datenanalyse mittels Spring Cloud Data Flow ist technisch vorbereitet, jedoch nicht fertig umgesetzt.

13.2.3. Bemerkung zu „Schnittstellen zu bestehenden ERP schaffen“

Es existiert eine LDAP Anbindung. Weiterführende Integrationen bedürfen weiterer Entwicklungsarbeit.

Tabelle 13.2.: Nichtfunktionale Anforderungen als Zielsetzungen

Beschreibung der Zielsetzung ausgehend von den nichtfunktionalen Anforderungen	Status
Auditability, Error Management: Zur Fehlerbehandlung und Nachvollziehbarkeit sind die System und Fehlermeldungen in einer Log Datei persistent abzulegen.	erfüllt
Auditability: Für die Nachvollziehbarkeit und Analysierbarkeit müssen alle eingehenden API Aufrufe geloggt werden.	teilweise erfüllt
Auditability: Logfiles müssen in einer für den Menschen lesbaren Form formatiert sein (kein BLOB).	erfüllt
Safety: Die Anwendung muss aus Datenschutzgründen eine Authentifizierungsmöglichkeit bieten.	erfüllt
Operability: Längere Arbeitsschritte des Systems werden dem Benutzer als solche erkenntlich gemacht. Die Benutzeroberfläche wird dabei nicht blockiert.	erfüllt
Accuracy: Sensordaten werden nach Genauigkeit der gelieferten Rohdaten, maximal 64 Bit, abgespeichert.	erfüllt
Recoverability: Fehleingaben von Benutzern müssen minimal serverseitig validiert werden und mittels verständlichen Fehlermeldungen kommuniziert werden.	erfüllt
Recoverability: Für kritische Operationen wie zum Beispiel das Löschen von Business relevanten Daten, müssen immer Bestätigungen von Seiten des Benutzers eingefordert werden.	erfüllt

Beschreibung der Zielsetzung ausgehend von den nichtfunktionalen Anforderungen	Status
Security: Die Integrität der Sensordaten muss sichergestellt sein, indem die Kommunikation von und zu Geräten ausschliesslich über sichere Kanäle erfolgt.	erfüllt
Capacity (IoT-Provider): Der Provider muss bis zu 10'000 Geräteanfragen pro Sekunden verarbeiten können.	erfüllt
Capacity (Benutzer-Plattform): Die Plattform muss bis zu 1'000 Benutzeranfragen pro Sekunde bewältigen können.	teilweise erfüllt
Localizability: Für eine Lokalisierung und Internationalisierung müssen die technischen Grundlagen bestehen, so dass die Übersetzung der Bildschirmtexte möglich ist. Die Codebasis muss für das Nachrüsten der Lokalisierung nicht verändert werden.	nicht erfüllt
Maintainability: Die Systemkonfiguration ist über ein Konfigurationsfile leicht anpassbar und erfordert keine Änderung an der Codebasis.	teilweise erfüllt

13.2.4. Bemerkungen zu „Für die Nachvollziehbarkeit und Analysierbarkeit müssen alle eingehenden API Aufrufe geloggt werden."

Die eingehenden API Requests werden zwar geloggt, jedoch sind über den Aufruf keine Details im Log ersichtlich, was eine Zurückverfolgung oder weiterführende Analyse verunmöglicht.

13.2.5. Bemerkungen zu „Der Provider muss bis zu 10'000 Geräteanfragen pro Sekunden verarbeiten können."

Die vom in diesem Projekt eingesetzten IoT Provider definierten Service Limits[1] erfüllen diese Anforderung.

13.2.6. Bemerkungen zu „Die Plattform muss bis zu 1'000 Benutzeranfragen pro Sekunde bewältigen können."

Es wurden keine für die API spezifischen Loadtests durchgeführt. Als mögliche Bottlenecks könnten das Client API oder die Datenbankzugriffe in Frage kommen.

[1] https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_iot

13.2.7. Bemerkungen zur nichtfunktionalen Anforderungen Localizability

Das Client UI ist technisch noch nicht vorbereitet, dass Bildschirmtexte austauschbar sein können.

13.2.8. Bemerkungen zur nichtfunktionalen Anforderungen Maintainability

Das Gesamtsystem ist zwar konfigurierbar, jedoch sind dies mehrere verschiedene Konfigurationen und das flexible Deployment auf andere PaaS Anbieter kann dazu führen, dass zusätzliche Konfigurationen oder Profile eingerichtet werden müssen oder im Extremfall sogar Codeanpassungen notwendig werden, wenn Teilsysteme untereinander nicht mehr kompatibel sind.

13.3. Bewertung

Der IoT Plattform Prototyp erfüllt die Aufgaben, welche hoch priorisiert wurden. Damit wurden viele technische Risiken überwunden und der proof of concept erbracht.

Im IoT Bereich sind die Punkte Skalierbarkeit ein wichtiger Punkt. Die Loadtests im Punkt Capacity (IoT Provider) haben gezeigt, dass die Lasten grundsätzlich bewältigt werden können, wenn die Anzahl Verbindungen zum Provider erhöht wird. Weil die Plattform mit dem Client API hinsichtlich der Capacity (Benutzer-Plattform)nicht getestet wurde, ist der Leistungsnachweis in diesem Punkt nicht erbracht worden.

Bequemlichkeiten für den Anwender fehlen mehrheitlich. Diese wurden bewusst tiefer priorisiert und stattdessen Wert auf die Funktionalität gelegt. Diese weisen ein geringes technisches Risiko auf und können durch kommende Updates einfach nachgeliefert werden.

14. Zusammenfassung und Ausblick

14.1. Zusammenfassung

Ausgehend von den Präqualifikationsberichten der SBB wurde zunächst eine Aufgabenstellung erarbeitet. Wesentliche Ziele waren das Verwalten des Geräte Lifecycles, das Konfigurieren der Geräte und das Visualisieren von Betriebsdaten. Die Anwendung sollte über ein Web UI bedient werden können und die Kommunikationskanäle sollten geschützt sein.

In einem ersten Schritt wurden IoT Dienstleister evaluiert, technische Grundlagen zu Kommunikationsprotokollen erarbeitet und dann die Kernfunktionalitäten für die IoT Plattform abstrahiert, um später einen reibungslosen Austausch des IoT Dienstleisters zu ermöglichen.

Der IoT Plattform Prototyp realisiert die Hauptanwendungsfälle Geräte Lifecycle Management, Gerätekonfiguration und Datenvisualisierung. Die relevanten Technologien für die Realisierung waren die Folgenden:

Tabelle 14.1.: Relevante eingesetzte Technologien

Technologie
Java Spring Boot Framework
Gradle
IntelliJ IDEA (JetBrains IDE)
Spring Cloud Data Flow
RabbitMQ
AWS IoT SDK

Die Gesamtanwendung gliedert sich in die Komponenten:

Tabelle 14.2.: Komponenten der IoT Plattform

Komponente
Web UI
Client API
IoT Service
Data Access
Model
AWS MQTT Source (Spring Cloud Data Flow)
Persistence Sink (Spring Cloud Data Flow)
Flow Model (Spring Cloud Data Flow)

Die entwickelte IoT Plattform vereinfacht die Verwaltung der IoT fähigen Geräte, indem sie die wesentlichen Funktionalitäten für den Benutzer in einer vereinfachten Form abstrahiert. Sie zeichnet sich weiter dadurch aus, dass sie flexibel deployed werden kann (on premise oder cloud) und die IoT Provider austauschbar sind.

14.2. Ausblick

Es folgt eine Auflistung der offenen Punkte und möglichen Weiterentwicklungen:

14.2.1. Provision API

Der Gerätezyklus ist nur eingeschränkt abgebildet. Aktuell muss ein Root CA Zertifikat manuell beim IoT Provider hinterlegt werden sowie die passenden Client Zertifikate auf den Geräten installiert sein. Hier wäre die nächste Ausbaustufe, dass gemäss Spezifikation die Geräte selbst über das Provision API ihre Zertifikate erhalten und die Reset Funktionalität anbieten.

14.2.2. Gerätegruppen

Aktuell sind Geräte nicht logisch zusammenfassbar. In einer Weiterentwicklung sollten gemäss Spezifikation Gerätegruppen eingeführt werden.

14.2.3. Benutzerrollen

Die Benutzerrollen sind in der aktuellen Version des Prototyps noch nicht implementiert und könnten in einem nächsten Entwicklungsschritt eingeführt werden.

14.2.4. Alarmierungen, Pushnotifications

Aktuell gibt es keinen Callback Mechanismus, durch den Gerätenachrichten bis zum Client gepusht werden. Hier wäre der nächste Entwicklungsschritt, dass der IoT Service ein Observerpattern anbietet, mitwelchem sich der Client als Observer registrieren kann und benachrichtigt werden kann.

14.2.5. Performanceoptimierungen für grosse Datenmengen

Aktuell ist die Client API nicht dafür optimiert, grosse Datenmengen zu liefern. In einem Weiterentwicklungsschritt sollte das API Pagination anbieten und es sollte eine Möglichkeit bestehen, Daten mittels Queries gezielter anzufordern. Eine mögliche Weiterentwicklung wäre das Zwischenschalten von Load Balancern. Dazu wäre einerseits ein Refactoring der Komponenten nötig und andererseits eine Konfiguration in der Hosting Umgebung oder Infrastruktur.

14.2.6. Konfiguration der Cloud Data Flow microservices

Die Konfiguration der stream processing pipeline erfolgt im Moment noch manuell. Eine mögliche Weiterentwicklung wäre das Anbieten von entsprechenden, vorgefertigten streaming pipelines, unter welchen man die passenden Komponenten selbst orchestrieren kann.

14.2.7. Konfiguration des IoT Providers

Die Konfiguration der IoT Anbindung erfolgt aktuell noch per Konfigurationsfile und ist für den normalen Webanwender nicht zugänglich. Eine mögliche Weiterentwicklung wäre, dass IoT Providerkonfigurationen über das UI angepasst werden können.

14.2.8. Anwendungsfall Cloud Data Flow

Aktuell werden die Rohdaten unverarbeitet über die stream processing Komponenten weitergeleitet. Hier bietet sich für die Zukunft an, Datenanalysen durchzuführen, indem Data Flow Processors in den Stream eingebunden werden.

14.2.9. UI Weiterentwicklungen

Das aktuelle Web UI ist schlicht und bietet wenige Bequemlichkeiten für den Anwender. Hier gibt es viele Möglichkeiten zur Weiterentwicklung und zum Feinschliff. Offene Punkte wären ein Dashboard, das Verwalten des Benutzeraccount oder die Filterung, Gruppierung und Sortierung der Geräte.

Literatur

[1] *Azure subscription and service limits, quotas, and constraints*. 2018. URL: https://docs.microsoft.com/en-us/azure/azure-subscription-service-limits#iot-hub-limits (besucht am 06/2018).

[2] *Cloud Foundry Certified Distributions*. 2018. URL: https://www.cloudfoundry.org/the-foundry/swisscom-application-cloud/ (besucht am 06/2018).

[3] Dan Abramov. *Presentational and Container Components*. 2015-03-23. URL: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0 (besucht am 06/2018).

[4] *Fetch API*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (besucht am 06/2018).

[5] James Friedman. *RMWC*. 2018. URL: https://github.com/jamesmfriedman/rmwc (besucht am 06/2018).

[6] *Hatch a swarm of AWS IoT things using Locust, EC2 and get your IoT application ready for prime time*. 2016. URL: https://www.concurrencylabs.com/blog/hatch-a-swarm-of-things-using-locust-and-ec2/ (besucht am 06/2018).

[7] HiveMQ. *MQTT Essentials Part 2: Publish & Subscribe*. 2015. URL: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe (besucht am 03/2018).

[8] *limit of 1024 connections*. 2017. URL: https://github.com/eclipse/paho.mqtt.python/issues/183 (besucht am 06/2018).

[9] *MariaDB Enterprise*. 2018. URL: https://docs.developer.swisscom.com/service-offerings/mariadb.html#limitations (besucht am 06/2018).

[10] *MariaDB Galera Cluster - Known Limitations*. 2018. URL: https://mariadb.com/kb/en/library/mariadb-galera-cluster-known-limitations/ (besucht am 06/2018).

[11] *Quotas and Limits*. 2018. URL: https://cloud.google.com/iot/quotas (besucht am 06/2018).

[12] React-Training. *React-Router Philosophy*. 2018. URL: https://reacttraining.com/react-router/web/guides/philosophy (besucht am 06/2018).

[13] *Redux - Actions*. 2018. URL: https://redux.js.org/basics/actions (besucht am 06/2018).

[14] *Redux - Presentational and Container Components*. 2018. URL: https://redux.js.org/basics/usage-with-react#presentational-and-container-components (besucht am 06/2018).

[15] *Redux - Reducers*. 2018. URL: https://redux.js.org/basics/reducers (besucht am 06/2018).

Tabellenverzeichnis

Abbildungsverzeichnis

Teil III.

Anhang

A. Aufgabenstellung

Aufgabenstellung Bachelorarbeit „Intelligente Güterwagen“

1. **Beteiligte**
   - Industriepartner: CloudGuard Software AG, Ansprechpartner: Herr Michael Schneider
   - Betreuer: Prof. Beat Stettler, Betreuer Stellvertreter: Herr Matthias Gabriel
   - Experte: Herr Marco Facetti
   - Gegenleser: Prof. Dr. Farhad Metha

2. **Studierende**
   Die Arbeit wird in einer gemischten SA / BA Gruppe mit drei Studierenden durchgeführt:
   - Herr Florian Bitterlin, führt die Arbeit als Bachelorarbeit durch
   - Herr Dominik Thamm, führt die Arbeit als Bachelorarbeit durch
   - Herr Giuliano De Gani, führt die Arbeit als Studienarbeit durch

3. **Ausgangslage**
   Die Möglichkeiten, welche sich durch den Einsatz von Internet of Things (IoT) fähigen Sensoren in Kombination mit energiearmen Kommunikationsmitteln eröffnen, haben in der Industrie und im Umfeld von logistischen Leistungserbringen Bedürfnisse geweckt.
   Angelehnt an Ausschreibungen der ÖBB und Präqualifikationsbericht der SBB soll im Rahmen dieser Bachelorarbeit eine IoT Plattform entwickelt werden. Industriepartner ist die Firma CloudGuard Software AG, welche im Lösungsmarkt für öffentliche Verkehrsmittel tätig ist.

4. **Beschreibung der Aufgabe**
   Der Industriepartner möchte einen Prototyp einer IoT Plattform, die folgende Ziele realisiert (mit Prioritäten 1 bis 3):
   - Das Lifecycle Management von IoT Geräten ermöglichen (P1)
   - IoT Geräte konfigurieren können (P1)
   - Datenanalysen ermöglichen und visualisieren (P2)
   - Schnittstellen zu bestehenden ERP schaffen (P3)
   - Export von Daten ermöglichen, (P3)
   - Bedienung über ein Web UI (P1)
   - Kommunikation der Geräte und Komponenten über sichere Kanäle mit authentifizierten und autorisierten Benutzern (P1)
   - Das Deployment flexibel halten (on premise oder verschiedene Cloud Provider) (P2)

   Aufgaben mit Priorität 1 sind zwingend zu erreichen, für die Aufgaben mit Prioritäten 2 und 3 reicht es, wenn erkennbar ist, dass die nötigen Grundlagen für eine Erweiterung geschaffen sind.

Im Übrigen gelten die Bestimmungen der Abteilung Informatik für Bachelorarbeiten.

Prof. Beat Stettler ▪ beat.stettler@ins.hsr.ch ▪ T +41 55 222 18 33
HSR Hochschule für Technik Rapperswil ▪ Oberseestrasse 10 ▪ CH-8640 Rapperswil
INS Institute for Networked Solutions ▪ www.ins.hsr.ch

B. Evaluation

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Bachelorarbeit

Intelligente Güterwagen

Evaluation

13. Juni 2018

Florian Bitterlin, Giuliano De Gani, Dominik Thamm

Betreuer
Prof. Beat Stettler

Industriepartner
CloudGuard Software AG, Zürich

Experte
Marco Facetti

Gegenleser
Prof. Dr. Farhad Mehta

Studiengang Informatik

Hochschule für Technik Rapperswil

Inhaltsverzeichnis

1 Einführung

1.1 Zweck

Dieses Dokument befasst sich mit der Evaluation der Technologien und den Einschränkungen bei der Umsetzung für die Bachelorarbeit/Studienarbeit welche unter dem Namen `Intelligente Güterwagen` an der Hochschule für Technik Rapperswil (HSR) im Frühjahrssemester 2018 bearbeitet wird.

1.2 Gültigkeitsbereich

In zeitlicher Hinsicht gilt dieses Dokument während des gesamten Projekts vom Datum seiner Erstellung bis zum Abschluss des gesamten Projekts ca. Mitte Juni 2018. Inhaltlich beschränken sich die nachfolgenden Ausführungen auf die Bachelorarbeit/Studienarbeit.

1.3 Referenzen

Die Referenzen für dieses Dokument finden sich - wo nötig - in den separaten Verzeichnissen „Tabellenverzeichnis“ und „Literaturverzeichnis“ im Anhang zu diesem Dokument.

2 Core Technologie

Für die Wahl des Technologie Stacks wurden Java Spring [4] und .NET Core [3] in die engere Auswahl genommen und verglichen.

2.1 Kriterien

Anhand der folgenden Kriterien, welche für das Projekt relevant sind, werden die beiden Technologien miteinander verglichen.

2.1.1 Maintainability

Maintainability beschreibt, wie gut das Gesamtsystem im Anschluss an die Projektarbeit vom Industriepartner gewartet werden kann.

Aufgrund der Schilderungen von CloudGuard AG während des Workshops vom 05.03.2018, wird dem Java Technology Stack eine bessere Wartbarkeit attestiert. Es ist davon auszugehen, dass der Integrationsaufwand in die bestehende Entwicklerumgebung geringer ist und das Knowhow grösser ist.

2.1.2 Deployment

Deployment beschreibt, wie einfach es ist, das System in Betrieb zu nehmen. Dazu zählen alle Anschaffungskosten und Aufwände für das Einrichten der Infrastruktur (wenn on premise Lösung gewählt).

Die auf Java basierenden Technologien sind erfahrungsgemäss sehr interoperabel und lassen sich auf alle Systeme deployen. Aufgrund der Verbreitung sind Deployment- Möglichkeiten und -Prozesse gut dokumentiert. Mit .NET Core hat Microsoft den .NET Technologie Stack modularisiert und es möglich gemacht, cross-plattform und light-weight zu deployen. Weil dies erst seit ungefähr 2 Jahren möglich ist, sind die Ressourcen und das Knowhow dazu weniger verbreitet. Aus Sicht der Lizenzgebühren sind beide Technologien vergleichbar, da unter .NET Core keine Notwendigkeit besteht, im Windows Ökosystem zu deployen.

2.1.3 Development Costs

Die Entwicklungskosten sind eine Masseinheit, die aussagt, wie hoch die Aufwände sind, um eine bestimmte Unit of work zu bewältigen. Sie hängt vom Skillset und den Erfahrungen der Projektbeteiligten ab. Auch ist zu berücksichtigen, dass ein besseres Knowhow die Qualität der Einheiten positiv beeinflusst.

Zwei Drittel der Projektmitarbeiter sind professionelle .NET Entwickler mit (kombiniert) 7 Jahren Erfahrung und favorisieren deshalb den .NET Core Stack gegenüber dem Java Technology Stack.

2.1.4 Integration

(Enterprise) Integration ist ein Kriterium, das die Interkonnektivität der Systeme beurteilt. Je einfacher die Gesamtlösung an bestehende Systeme angeschlossen werden kann, desto besser ist die Bewertung.

Beide Technologien sind verbreitet, reif und gut integrierbar.

2.1.5 Ecosystem

Dieses Kriterium befasst sich mit den Möglichkeiten, ob und wie viele Drittanbieter, Bibliotheken und Lösungen für das Projekt genutzt werden können und ob diese kostenpflichtig und quelloffen sind. Weiter wird der Umfang an verfügbarer Dokumentation bewertet.

Beide Technologien haben eine grosse Entwicklergemeinschaft und besitzen vergleichbare, quelloffene, kostenlose oder für kommerzielle Zwecke nutzbare Lösungen. Das Java Ökosystem ist grösser, jedoch ist die Rate an Neuentwicklungen im Verhältnis zur Verbreitung geringer als dies bei .NET (Core) Projekten.

2.1.6 Performance

Performance umfasst die Aspekte Robustheit, Laufzeitgeschwindigkeit, Skalierbarkeit und Sicherheit.

Java hat den Ruf, „battle-hardened“ und damit robust zu sein, weil es schon sehr lange besteht und weit verbreitet im Einsatz ist. Dem gegenüber steht ein relativ moderner und schlanker .NET Core Stack, der mit weniger Altlasten und Rückwärtskompatibilitäten im Design auskommt.

Gängige Software Benchmarks[12] mit Java 10 und C# zeigen, dass die Laufzeitgeschwindigkeit vergleichbar ist. Insgesamt lässt sich kein eindeutiger Gewinner eruieren, da die Grössen Laufzeitgeschwindigkeit, Skalierbarkeit und Sicherheit viel stärker von der tatsächlichen Implementierung abhängen, als von der zugrundeliegenden Technologie.

2.1.7 Soft Metrics

Hier werden alle weiteren Metainformationen zusammengefasst, die nur schlecht messbar sind, wie zum Beispiel Community Vitalität, Momentum, Anzahl Fans, Interesse in den vergangenen 12 Monaten usw.

Aufgrund der erfassten Daten gemäss stackshare.io[11] geben wir darum .NET einen leichten Vorzug.

2.2 Resultat

Die Gegenüberstellung der beiden Technology Stacks nutzt gewichtete Schulnoten 1-6 nach Schweizer Schulsystem, wobei 1 unbrauchbar und 6 sehr gut bedeutet.

Tabelle 1: Gegenüberstellung Java Spring vs .NET Core

Kriterium	Gewichtung	**Java Spring**	Value	**.NET Core 2.0**	Value
Maintainability	0.2	6	1.2	4	0.8
Deployment	0.15	5.5	0.825	4.5	0.675
Development Costs	0.25	3.5	0.875	5.5	1.375
Integration	0.25	5	1.25	5	1.25
Ecosystem	0.05	5.5	0.275	5	0.25
Performance	0.05	5	0.25	5	0.25
Soft Metrics	0.05	4.5	0.225	5	0.25
Gesamtbewertung	1	35	**4.9**	34.5	**4.85**

Es zeigt sich, dass kein klarer Gewinner ermittelt werden kann und beide Technologiestacks verwendet werden können. Wenn man die Gewichtung zugunsten der Development Costs optimiert, gewinnt .NET Core. Wenn hingegen die Maintainability stärker gewichtet wird, geht Java als Gewinner hervor.

3 IoT Provider

Viele Cloudanbieter haben bereits Lösungen zum Anbinden und Verwalten von IoT-Geräten entwickelt. Diese bieten skalierbare Plattformen und teilweise auch Infrastruktur um eine grosse Menge an Geräten zu verwalten. Funktionen reichen von sicheren Verbinden von Geräten, deren Zugriffssteuerung über Gruppenverwaltung, eventbasierten Aktionen bis hin zu Überwachung und Datenanalyse mit AI.

Auswahl: Bei der Auswahl möglicher Kandidaten wurden Anbieter, welche nur die Custom-Entwicklung von IoT Plattformen anbieten, aussenvor gelassen. Alle analysierten Anbieter verfügen über konkrete Produkte welche von Entwicklern mittels SDKs und/oder standardisierten Protokollen in eigene Applikation und Geräte-Hardware integriert werden können.

Folgende Plattformen wurden analysiert:

- AWS IoT
- Google IoT
- Eclipse IoT
- Azure IoT
- Everyware Cloud M2M/IoT Platform

3.1 Übersicht

Die Tabelle bietet einen Vergleich der verschiedenen Systeme anhand ausgewählter Kriterien.

Tabelle 2: Vergleich verschiedener Cloudplattformen und deren Angebot an Software

Kriterien / Anbieter	AWS	Google	Azure	Eclipse	Everyware Cloud
Device SDK	Ja	Ja	Ja	Nein	Ja
Device SDK Sprachen	C, JS, Arduino, Java, Python, iOS, Android, C++	Go, Java, .Net, Node.js, PHP, Ruby, Python	.Net, C, Java, Node.js	-	Java, C++
Device/Sen: Software	Amazon FreeRTOS	-	-	Constrained Device Stack	-
Gateway Software	AWS Greengrass	-	Azure Edge	Eclipse Kura	3rd Party (z.b. Kura)

Kriterien / Anbieter	AWS	Google	Azure	Eclipse	Everyware Cloud
Datenanaly	Ja	Ja	Ja	(Ja)	Ja
Protokolle	MQTT, HTTP	MQTT, HTTP	MQTT, HTTP, AMQT	MQTT	MQTT
Gateway Verarbeitun	Ja	-	Ja	(Ja)	-
Gateway Caching	Ja	-	Ja	Ja	-
IoT Plattform	AWS IoT Core	Google IoT Core	Azure IoT Hub	Eclipse Kapua	Everyware Cloud IoT Plattform
Sichere Verbindung	Ja	Ja	Ja	Ja	Ja
Cloud API	SDK	SDK	SDK	REST	REST
SDK Sprachen	Java, PHP, Node.js, .NET, Python, Ruby, C++, Go, JavaScript	Go, Java, .Net, Node.js, PHP, Ruby, Python	.Net, C, Java, Node.js, PHP, Python, Ruby	-	-

3.2 Analyse

Alle für die Analyse in Betracht gezogenen Lösungen bieten die Funktionalität um unsere Anforderungen zu erfüllen. Auf diese wird deshalb nicht weiter eingegangen. Betrachtet man den Umfang des Angebots bietet AWS somit das beste Paket. Funktionen, API und SDK sind umfangreich und gut dokumentiert. Azure und Google sind vergleichbar, liegen bei API und Dokumentation nicht auf gleicher Höhe. Die Projekte von Eclipse haben den open source Vorteil, sind aber teilweise noch in Beta und noch nicht so ausgereift. Everyware Cloud bietet wenig Dokumentation und keine öffentliche SDKs.

Tabelle 3: Evaluation der IoT Provider anhand der definierten Kriterien

	Gewichtung	AWS	Azure	Google	Eclipse	Custom
Plattformintegration	0.2	4 / 0.8	4 / 0.8	4 / 0.8	3 / 0.6	6 / 1.2
Betriebskosten	0.1	2 / 0.2	2 / 0.2	2 / 0.2	5 / 0.5	5 / 0.5
Skalierbarkeit	0.1	6 / 0.6	6 / 0.6	6 / 0.6	4 / 0.4	2 / 0.2
Sicherheit	0.2	5 / 1.0	5 / 1.0	5 / 1.0	3 / 0.6	3 / 0.6
Inbetriebnahme	0.2	6 / 1.2	5 / 1.0	5 / 1.0	5 / 1.0	2 / 0.4
Reife	0.2	6 / 1.2	5 / 1.0	5 / 1.0	3 / 0.6	2 / 0.4
Total	1.0	29 / 5.0	27 / 4.6	27 / 4.6	23 / 3.7	20 / 3.3

3.2.1 Plattformintegration

Wie einfach ist den Provider in die Plattform zu integrieren. Einfluss haben Dinge wie: verfügbare SDKs, Umfang angebotener APIs usw.

3.2.2 Betriebskosten

Die Kosten zum Betreiben der Infrastruktur und der Software über die Jahre.

3.2.3 Skalierbarkeit

Wie gut skaliert der Provider mit wachsender Anzahl Geräte in Bezug auf Software und Hardware? Sind grosse Anpassungen nötig?

3.2.4 Sicherheit

Was bietet der Provider in Sachen Sicherheit: Rechteverwaltung, gesicherte Verbindung, Geräte-Zertifikate, usw.

3.2.5 Inbetriebnahme

Wie gross ist der Aufwand zur Inbetriebnahme des Providers? Dazu zählen das Aufsetzen von Servern, Erstellen von Konten und die initiale Konfiguration.

3.2.6 Reife

Wie ausgereift ist die Software? Sind viele Bugs zu erwarten? Wird die Entwicklung von einer professionellen Firma vorangetrieben?

3.3 AWS IoT

AWS IoT besteht aus mehreren Softwareservices welche das Verwalten von IoT Geräte über die Cloud erlauben. Von der Cloudplattform bis hin zur embedded Device Lösung wird alles angeboten. Die IoT Geräte SDK zum Kommunikation mit der Plattform ist verfügbar in 8 Programmiersprachen.

3.3.1 Core

Ein Service welcher Geräte einfach und sicher mit der Cloud und anderen Diensten verbindet. Unterstützt werden HTTP, WebSockets, und MQTT als Protokol. Regeln ermöglichen das Reagieren auf bestimmte Events. „Shadow devices“ speichern den letzten Zustand falls ein Gerät mal nicht erreichbar ist. Offizielle Webseite: aws.amazon.com/iot

Infrastruktur: AWS IoT läuft auf den Servern von Amazon. Die Skalierung anhand der Anzahl Geräte funktioniert automatisch. Amazon stellt Infrastruktur für die Cloudserver und Endpunkte (Message Broker Gateways) bereit.

3.3.2 Device Management

Erlaubt das einfach Hinzufügen, Organisieren und Aktualisieren von IoT-Geräten in grosser Anzahl. Unterstütz das Verwalten von Geräteattributen, Zertifikaten und Zugriffsregeln für Gruppen von Geräten, und verteilt Firmwareupdates auf die Geräte.

3.3.3 Device Defender

Überwacht das Verhalten von Geräten und findet Anomalien. Kann Regeln für offene Ports, Datenverkehr und Verbindungen. Meldet Geräte mit verdächtigem Verhalten zur Überprüfung durch den Benutzer.

3.3.4 Amazon FreeRTOS

Basiert auf dem FreeRTOS-Kernel und beinhaltet Bibliotheken zum Zugriff auf das lokale Netzwerk. Erlaubt es OTA Updates für die Software. Garantiert die sichere Verbindung mittels Zertifikatmanage-

ment und unterstützt „Code signing“. Kann sich direkt oder über einen Greengrass Gateway mit AWS Core verbinden. Läuft auf ARM, MIPS und weiteren Architekturen. Die Software ist Opensource.

3.3.5 Greengrass

Optionaler Gateway zur Cloud, welcher Daten von lokalen Geräten filtert, zwischenspeichert und verarbeitet. Kann auf Linux ARM oder x86 installiert werden. Verbundene Geräte verwenden Amazon FreeRTOS oder die IoT Geräte SDK zur Kommunikation mit Greengrass.

3.3.6 Analytics

Erlaubt die detaillierte Analyse von Gerätedaten. Überwacht Veränderungen über die Zeit, filtert Daten anhand von Regeln und verarbeitet Daten für die spätere Analyse. Integriert künstliche Intelligentz um Gerätedaten auszuwerten.

3.3.7 Gebührenmodell

Bei einem Betrieb des IoT services mittels AWS IoT ergeben sich aufgrund des aktuellen `Gebühr enmodells` mit gesetzten Rahmenbedingungen folgende zu erwartende Kosten:

Rahmenbedingungen

- 20000 Geräte sind jederzeit mit AWS verbunden
- Jedes Gerät verschickt 1 Message pro Minute, also 1440 Messages pro Tag
- Jedes Gerät aktualisiert seinen Device Shadow halbstündlich, also 48-mal pro Tag
- Pro Tag werden bis zu 50 Geräte neu eingerichtet
- Die AWS Region ist EU Frankfurt

Kosten

Kostenpunkt	Kosten pro Monat in CHF
Connectivity	83
Messaging	1037
Device Shadow und Registry	88
Total	**1208**

Die Gebühren bei AWS IoT skalieren linear hauptsächlich mit der Anzahl Geräte und über das Messaging über MQTT.

3.4 Google Cloud IoT

Servicesuite für Verbindung und Verwaltung von IoT Geräten. Mittels SDK kann mit den Services kommuniziert werden. Die Cloud SDK unterstützt 7 Programmiersprachen. Offizielle Website: cloud.google.com/solutions/iot

Infrastruktur: Google stellt die Infrastruktur für die Cloudserver und Verbindungsknoten der Geräte zur Verfügung.

Im Gegensatz zu AWS IoT bietet Google keine eigene Software für Endgeräte ausser der SDK.

3.4.1 Core

Unterstützt die Protokolle HTTP und MQTT mit sicherer Verbindung mittels TLS und Zertifikaten.

3.4.2 Pub/Sub

Nachrichtenservice von Google zum Senden und Empfangen von themenbasierten Daten. Dient als Kommunikation zwischen Geräten und fungiert als Puffer für Lastspitzen.

3.4.3 Device Manager

Erlaubt die Verwaltung der Geräte. Dies beinhaltet Identifikation, Authentifikation und Autorisierung sowie auch Konfiguration von Geräten. Vergabe von Rollen für Zugriff auf Devices und Daten.

3.4.4 Datenanalyse

Integration von Google Big Data Analytics und weiteren smart Services für die Datenanalyse mit oder ohne Maschine Learning.

3.5 Eclipse IoT

Das Projekt von Eclipse stellt in erster Linie einen Software-Stack für den Bau einer IoT Plattform bereit. Die Software ist open source muss aber selbst gehostet werden. Offizielle Website iot.eclipse.org

Infrastruktur: Es werden keine Server, Access Points oder andere Hardware zur Verfügung gestellt.

3.5.1 IoT Constrained Device Stack (Device)

Software für den Bau einer Applikation auf einem Embedded Gerät. Beinhaltet folgende Technologien:

Eclipse Edje

Stellt eine API zum Ansteuern von Hardware-Features bereits. Abstrahiert damit native Bibliotheken und Treiber.

Eclipse Paho

Stellt die Kommunikation über das Netzwerk per MQTT-Protokoll bereit.

Eclipse Wakaama

Implementiert das OMA LWM2M Protokoll für das Remote Management des Gerätes.

3.5.2 IoT Gateway & Smart Device Stack (Gateway)

Software zur Installation auf einem IoT Gateway. Bestehend aus:

Eclipse Kura

Software für ein IoT Gateway bestehend aus: Remote Management, Messaging Service, Network Management (z.B. mit Kapua), Schnittstellen API und Runtime Environment mit OSGi Runtime. Läuft auf Linux.

Eclipse SmartHome

IoT Gateway spezialisiert für SmartHomes.

3.5.3 IoT Cloud Platform Stack (Cloud)

Cloudplattform zur Verwaltung von Geräten.

Eclipse Kapua

Modulare Plattform für das Verwalten von IoT Devices und Gateways. Bestehend aus: Gerätemanagement, Geräteregistration, Datenanalyse und Message Broker.

3.6 Azure IoT

Softwarepaket zur Vernetzung und Verwaltung von IoT Geräten. Die Azure IoT device SDK unterstützt 5 Programmiersprachen. Mit IoT Suite (PaaS) und IoT Central (SaaS) gibt es 2 Lösungsvarianten im Angebot.

Eine Übersicht gibt es hier: azure.microsoft.com/en-us/solutions/internet-of-things

Infrastruktur: Genutzt wird die Azure Cloud Infrastruktur welche Hardware für Server und Cloud Gateways bereitstellt.

3.6.1 Azure IoT Hub

Ist das Herzstück der Cloudplattform. Unterstützt AMQP 1.0 optional mit WebSocket, MQTT 3.1.17 und natives HTTP 1.1 mit TLS. Beinhaltet Funktionalität für Geräte Twins (Shadow Devices), Securitymanagement, Regelmanagement für Geräte und User, sowie Überwachung der Endgeräte.

3.6.2 Azure IoT Edge

Erlaubt die lokale Filterung, Bearbeitung und Überwachung von Daten auf den Geräten bzw. Gateways. Kann auch als Protokollkonverter für Geräte dienen die sonst nicht direkt mit dem Hub kommunizieren können.

3.6.3 Stream Analytics

Erlaubt Analysen und Auswertung grosser Datenmengen von IoT und nicht-IoT Geräten. Kann auch auf Azure Edge genutzt werden.

4 Gateways

Für die Wahl des Gateways wurden der Raspberry Pi und Pycom-Gertäe in die engere Auswahl genommen und verglichen.

4.1 Kriterien

Anhand der folgenden Kriterien, welche für das Projekt relevant sind, werden die Geräte miteinander verglichen.

4.1.1 Hardware & Konnektivität

Der Raspberry Pi ist an sich ein vollwertiger Computer und könnte theoretisch auch als Desktopersatz eingesetzt werden. In der Version 3 Model B bietet der Raspberry Pi, neben der davor schon gängigen Ethernetschnittstelle und den USB Typ A Ports, auch Wlan und Bluetooth[2]. Über seine GPIOs kann der Raspberry Pi mit einer Vielzahl an Erweiterungen, wie zum Beispiel dem Sense HAT[10], ausgestattet werden.

Die Hardware der Pycom-Geräte[6] ist aufgeteilt in Kommunikationsmodule und Sensor- oder Erweiterungsshields. Je nach Anwendungsfall können die Kommunikationsmodule auf andere Shields gesteckt und so individuell kombiniert werden. Alle Kommunikationsmodule beherrschen Wlan und Bluetooth. Andere Varianten unterstützen zusätzlich noch, z.B. LoRa, Sigfox, LTE-M, etc. oder eine Kombination davon. Pycom bietet zwei Sensorshields an. Eines fürs GPS-Tracking[9] und eines mit, unter anderem, Temperatur-, Feuchtigkeits- und Drucksensoren[8]. Zusätzlich gibt es ein Shield, welches über GPIOs erweiterbar ist.

4.1.2 Programmierbarkeit

Der Raspberry Pi bietet ein vollwertiges Betriebssystem und kann so um die Programmierbarkeit mit beinahe jeder gängigen Sprache erweitert werden. Die Programmierbarkeit der Sensorerweiterungen ist jedoch abhängig von den dazu zur Verfügung stehenden Bibliotheken. So könnte es auch vorkommen, dass unterschiedliche Programmiersprachen eingesetzt werden müssen, wenn mehrere Sensorerweiterungen nicht über eine gemeinsame Sprache angesprochen werden können.

Pycom-Geräte sind per MicroPython[1] programmierbar. Eine für MicroController optimierte Python 3 Implementation. Für sämtliche von Pycom hergestellten Sensorshields, werden auch Bibliotheken zur Verfügung gestellt um die verbaute Hardware relativ einfach per MicroPython-Code auszulesen.

4.1.3 Portabilität

Sowohl der Raspberry Pi wie auch die Pycom-Geräte lassen sich über einen Micro-USB-Anschluss mit einer 5V-Energiequelle versorgen, z.B. ein handelsübliches USB-Battery-Pack. Beim Pycom ist der zu erwartende Verbrauch mit den eingesetzten Sensorshields relativ konstant. Am Raspberry Pi können über die GPIOs und die USB-Ports mehrere externe Geräte angeschlossen werden. Dies bedeutet jedoch, dass je nachdem die Energiequelle den eingesetzten Erweiterungen angepasst werden muss, damit sie die benötigte Leistung auch liefern kann.

Für beide Geräte gibt es verschiedene Gehäuse zu kaufen um sie vor Transportschäden zu bewahren. Es sind ebenso für beide Geräte IP67-zertifizierte Gehäuse verfügbar welche gegen Spritzwasser und Staub schützen.

4.1.4 Dokumentation & Community Support

Der Raspberry Pi erfreut sich auch in der Bastler-Szene einer grossen Beliebtheit. So gibt es im Internet eine grosse Anzahl an Tutorials und How-To's zu verschiedenen Projekten. Ebenso Cloud-Anbieter im IoT-Bereich sind sich der Beliebtheit des Raspberry Pis bewusst. So bieten sie für ihre Services, z.B. AWS IoT, bereits Tutorials und für den Raspberry Pi angepasste SDKs an.

Pycom stellt für ihre Geräte eine ausführliche Dokumentation mit Installationsanleitung und vielen Code-Beispielen zur Verfügung[5]. Es wird sowohl auf das Handling der Konnektivität, als auch das Handling der Sensortechnik eingegangen und es stehen ebenso Firmware und API Referenzen zur Verfügung. Im Pycom-Forum wird rege zu Problemen und neuen Projekten der Community diskutiert.

4.1.5 Lizenzen

Zur Hardware der unterschiedlichen Raspberry Pi Varianten wurden zwar immer wieder unterschiedliche Layouts zur Dokumentation veröffentlicht, sie stehen jedoch nicht unter einer Open-Source-Lizenz. Es gibt verschiedene Betriebssysteme für den Raspberry Pi unter verschiedenen offenen oder proprietären Lizenzen. Die eingesetzten Sensorerweiterungen und ihre Firmware stehen jeweils unter einer eigenen Lizenz. Dies muss beim Kauf von Sensoren jeweils speziell beachtet werden.

Die Hardware der Pycom-Geräte stehen unter einer proprietären Lizenz. Genauer spezifiziert wird dies auf der Webseite des Unternehmens jedoch nicht. Die Firmware, wie auch die zur Verfügung gestellten Bibliotheken, stehen unter einer GPLv3 kompatiblen Lizenz[7] und ist somit offen zugänglich.

4.2 Resultat

Die Gegenüberstellung der Gateways nutzt gewichtete Schulnoten 1-6 nach Schweizer Schulsystem, wobei 1 unbrauchbar und 6 sehr gut bedeutet.

Tabelle 5: Gegenüberstellung Gateways

Kriterium	Gewichtung	**Pycom**	Value	**Raspberry Pi**	Value
Hardware	0.20	5.5	1.1	5	1.0
Programmierbarkeit	0.35	5.5	1.925	4.5	1.575
Portabilität	0.15	5	0.75	4.5	0.675
Dokumentation	0.25	5.5	1.375	5	1.25
Lizenzen	0.05	5	0.25	4.5	0.225
Gesamtbewertung	1	26.5	**5.4**	23.5	**4.725**

Der Raspberry Pi ist zwar weit verbreitet und bietet viele Möglichkeiten, er ist aber immer noch klar eine Bastler-Hardware. Sensorerweiterungen hätten zusätzlich und von anderen Herstellern bezogen werden müssen, ohne die Qualität der zur Verfügung gestellten Firmware zu kennen.

Die Pycom-Geräte bieten bereits Sensorerweiterungen aus gleichem Haus mit einer ausführlichen Dokumentation und sichergestellter Kompatibilität. Auf den Pycom-Geräten ist man zwar auf MicroPython beschränkt, dafür ist sichergestellt, dass nur eine Sprache für die Programmierung des Geräts verwendet werden muss. Die Dokumentation ist an einer Stelle zentral einsehbar.

5 Entscheidung

5.1 Technologiewahl

Im Folgenden werden die Beschlüsse bezüglich der Auswahl der zu verwendenden Technologien beschrieben.

5.1.1 User Interface

Das User Interface wird mit Hilfe von React und Redux umgesetzt. Diese Kombination bietet die Möglichkeit, das User Interface mit genügend Logik auszustatten, damit es als eigenständiger Client agieren kann. Die Kommunikation mit der Plattform soll ausschliesslich über die Client API geführt werden. Durch den Einsatz eines React UI Frameworks erhoffen wir uns die Entwicklungskosten für ein ansprechendes Grunddesign gering halten zu können.

Der Industriepartner setzt bei eigenen Projekten bereits auf React. Zusätzlich besitzt das Projektteam Erfahrungen im Umgang mit React und Redux. Wir sehen daher React als optimale Lösung, bei der uns unsere Erfahrung rasche Erfolge bescheren und wir trotzdem auf dem Technologie-Stack des Industriepartners bleiben können.

5.1.2 Gateway

Das Hauptaugenmerk der Arbeit liegt auf der Umsetzung der Plattform. Die Geräte dienen als Mittel zum Zweck um die Plattform unter realen Bedingungen testen zu können. Dafür bieten uns die Pycom-Geräte genug Einfachheit, um uns nicht mit zusätzlichen Problemen bei der Arbeit zu stören und genügend Möglichkeiten, um alle benötigten Features grundsätzlich abdecken zu können.

5.1.3 Core / Client API

Die Backend Anwendung wird als RESTful HTTP Web Service mit dem Java Spring Framework realisiert. Durch die Verwendung des Java Spring Frameworks erhoffen wir uns eine hohe Interoperabilität, ein einfaches Deployment und gut ausgebaute Integrationsmöglichkeiten zu Drittanbietern. Wir nehmen dabei höhere Entwicklungskosten aufgrund mangelnder Erfahrung mit Spring in Kauf.

5.1.4 IoT Provider

Als IoT Provider setzen wir auf AWS. Aus den evaluierten Produkten überzeugt Amazon mit einem reifen Produkt, gut ausgebauter SDK und umfangreicher Dokumentation. Im ersten Jahr ist der Service

bis zu einer bestimmten (für uns ausreichenden) Grösse gratis. Wir legen uns damit vorerst auf einen fixen Anbieter fest. AWS ermöglicht es uns, durch einen simplen Einstieg schnell einen Prototyp zu erstellen. Die Schnittstelle zum Provider wird so abstrahiert, sodass der Austausch des Providers zu einem späteren Zeitpunkt mit wenig Aufwand möglich ist.

5.1.5 Inter-System-Kommunikation

Für die Kommunikation der IoT Geräte mit der Plattform und zum Client werden die weitverbreiteten und erprobten Kommunikationsprotokolle wie MQTT und HTTPS verwendet. Wir erwarten dadurch gut dokumentierte Schnittstellen und hohe Sicherheit der Kommunikationskanäle. Wir verzichten bewusst auf den Support von weiteren Protokollen, um den Umfang einzugrenzen und den Fokus auf die Plattformentwicklung zu setzen.

5.2 Umfang

Das Projekt wird grundsätzlich agil geführt. Deshalb wird fortlaufend gemäss der Priorisierung der Use Cases an der Umsetzung gearbeitet.

Aufgrund der abgesteckten Projektzeit werden aber Zielsetzungen mittels Meilensteinplanung definiert, welche richtungsweisend sind für die Einplanung und Umsetzung der Arbeitspakete.

Literatur

[1] Damien George. *The Micro Python Website*. 2017. URL: http://micropython.org/ (besucht am 03/2018).

[2] https://www.modmypi.com/. *ModMyPi Raspberry Pi Comparison Chart*. 2017. URL: https://www.modmypi.com/download/raspberry-pi-comparison-chart.pdf (besucht am 03/2018).

[3] Microsoft. *.NET Core*. 2018. URL: https://www.microsoft.com/net/ (besucht am 03/2018).

[4] Pivotal Software. *Java Spring*. 2018. URL: https://spring.io/ (besucht am 03/2018).

[5] Pycom Ltd. *Pycom Documentation*. 2018. URL: https://docs.pycom.io/ (besucht am 03/2018).

[6] Pycom Ltd. *Pycom Hardware*. 2018. URL: https://pycom.io/hardware/ (besucht am 03/2018).

[7] Pycom Ltd. *Pycom Ltd. Licenses v2.2*. 2017. URL: https://pycom.io/wp-content/uploads/2018/02/Pycom-Licences-v2.2.pdf (besucht am 03/2018).

[8] Pycom Ltd. *Pycom Pysense*. 2018. URL: https://pycom.io/hardware/pysense-specs/ (besucht am 03/2018).

[9] Pycom Ltd. *Pycom Pytrack*. 2018. URL: https://pycom.io/hardware/pytrack-specs/ (besucht am 03/2018).

[10] Raspberry Pi Foundation. *Sense HAT*. 2018. URL: https://www.raspberrypi.org/products/sense-hat/ (besucht am 03/2018).

[11] StackShare. *StackShare Comparison Java .NET*. 2018. URL: https://stackshare.io/stackups/dot-net-vs-spring (besucht am 03/2018).

[12] unbekannt. *Benchmarksgame*. 2018. URL: http://benchmarksgame.alioth.debian.org/u64q/csharp.html (besucht am 03/2018).

Tabellenverzeichnis

C. API-Dokumentation

riot - Client API

Contents

Overview

Public client API for the riot platform

Version information

Version : 0.1.0

Contact information

Contact : Client

License information

License : HSR-License *License URL* : http://www.hsr.ch

URI scheme

Host : client.riot.bitterlin.net *BasePath* : /api

Tags

- config-controller : Configurations endpoint
- devices-controller : Device endpoint
- export-controller : Export endpoint
- sensor-data-controller : SensorData endpoint

Paths

Create config

`POST /config`

Description

Creates a new configuration.

Parameters

Type	Name	Description	Schema
Body	**createConfigDto** *optional*	The new configuration to create.	CreateDeviceConfigDTO

Responses

HTTP Code	Description	Schema
200	OK	DeviceConfigDTO
400	Bad request.	No Content
422	Unprocessable Entity. Provided config type does already exist.	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- config-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Get configurations

`GET /config`

Description

Gets all configurations.

Responses

HTTP Code	Description	Schema
200	OK	< DeviceConfigDTO > array
400	Bad request.	No Content

Produces

- `application/json`

Tags

- config-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Get config

`GET /config/{type}`

Description

gets the configuration with the provided identifier.

Parameters

Type	Name	Description	Schema
Path	**type** *optional*	The type (identifier) of the requested configuration.	string

Responses

HTTP Code	Description	Schema
200	OK	DeviceConfigDTO
404	Not found.	No Content

Produces

- `application/json`

Tags

- config-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Update config

`PUT /config/{type}`

Description

updates the associated configuration with the identifier and the new configuration provided.

Parameters

Type	Name	Description	Schema
Path	**type** *required*	The configuration type which acts as the identifier.	string
Body	**config** *required*	config	UpdateDeviceConfigDTO

Responses

HTTP Code	Description	Schema
200	OK	DeviceConfigDTO
400	Bad request.	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- config-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Delete config

`DELETE /config/{type}`

Description

Deletes the associated configuration with the identifier provided.

Parameters

Type	Name	Description	Schema
Path	**type** *required*	The configuration type which acts as the identifier.	string

Responses

HTTP Code	Description	Schema
200	OK	No Content
204	No Content. Successful deletion.	No Content
400	Bad request.	No Content

Produces

- `application/json`

Tags

- config-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Create a new device

`POST /devices`

Parameters

Type	Name	Description	Schema
Body	**deviceDto** *required*	deviceDto	CreateDeviceDTO

Responses

HTTP Code	Description	Schema
200	OK	DeviceDTO
201	Device created	DeviceDTO
400	Error in payload	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Gets list of all devices

`GET /devices`

Responses

HTTP Code	Description	Schema
200	Success	< DeviceDTO > array

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Get information about a device

`GET /devices/{uid}`

Parameters

Type	Name	Description	Schema
Path	**uid** *required*	The uuid of the device requested.	string

Responses

HTTP Code	Description	Schema
200	Success	DeviceDTO
404	Device not found	No Content

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Update information of a device

`PUT /devices/{uid}`

Parameters

Type	Name	Description	Schema
Path	**uid** *required*	The uuid of the device to update.	string
Body	**updateDeviceDTO** *required*	updateDeviceDTO	UpdateDeviceDTO

Responses

HTTP Code	Description	Schema
200	Success	DeviceDTO
400	Error in payload	No Content
404	Device not found	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Delete a device

`DELETE /devices/{uid}`

Parameters

Type	Name	Description	Schema
Path	**uid** *required*	The uuid of the device to delete.	string

Responses

HTTP Code	Description	Schema
200	OK	No Content
204	Success	No Content
404	Device not found	No Content

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Set the certificate of a device

`PUT /devices/{uid}/certificate`

Parameters

Type	Name	Description	Schema
Path	**uid** *required*	The uuid of the device to update.	string
Body	**certificateDTO** *required*	certificateDTO	UpdateDeviceCertificateDTO

Responses

HTTP Code	Description	Schema
200	OK	No Content
204	Success	No Content
400	Error in payload	No Content
404	Device not found	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Delete the certificate of a device

`DELETE /devices/{uid}/certificate`

Parameters

Type	Name	Description	Schema
Path	**uid** *required*	The uuid of the device to update.	string

Responses

HTTP Code	Description	Schema
200	OK	No Content
204	Success	No Content
400	Something is wrong with the device	No Content
404	Device not found	No Content

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Get configuration of a device

`GET /devices/{uid}/config`

Parameters

Type	Name	Description	Schema
Path	**uid** *required*	The uuid of the device requested.	string

Responses

HTTP Code	Description	Schema
200	Success	< string, object > map
404	Device not found	No Content

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Update configuration of a device

`PUT /devices/{uid}/config`

Parameters

Type	Name	Description	Schema
Path	**uid** *required*	The uuid of the device to update.	string
Body	**configDTO** *required*	configDTO	UpdateDeviceConfigDTO

Responses

HTTP Code	Description	Schema
200	Success	< string, object > map
400	Error in payload	No Content
404	Device not found	No Content

Consumes

- `application/json`

Produces

- `application/json`

Tags

- devices-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Export sensor data

`GET /export`

Description

Retrieving and filtering sensor data to export them as files. The file format can be choosen by setting the accept header in the request

Parameters

Type	Name	Description	Schema
Query	**deviceUid** *optional*	The device's uid to retrieve sensor data for.	string
Query	**maxLimit** *optional*	Maximal number of sensor data to return. Will be ignored if below 1.	integer (int64)
Query	**newer** *optional*	Only sensor data newer than this Timestamp will be returned.	string (date-time)

Type	Name	Description	Schema
Query	**older** *optional*	Only sensor data older than this Timestamp will be returned. Will be ignored if it is earlier than newer.	string (date-time)

Responses

HTTP Code	Description	Schema
200	Success	No Content
400	Bad request	No Content
406	Not Acceptable: Possibly no supported response content-type found in the accept header field	No Content

Produces

- `text/csv`

Tags

- export-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

POST /login

Description

Authenticates a user

Parameters

Type	Name	Description	Schema
Body	**userLoginDto** *required*	userLoginDto	UserLoginDTO

Responses

HTTP Code	Description	Schema
200	OK **Headers** : `Authorization` (string) : Contains the JWT in a Bearer Scheme: *Bearer AAA...*	SuccessfulLoginDTO
403	Forbidden	No Content

Get sensor data

`GET /sensordata`

Description

Retrieving and filtering sensor data

Parameters

Type	Name	Description	Schema
Query	**deviceUid** *optional*	The device's uid to retrieve sensor data for.	string
Query	**maxLimit** *optional*	Maximal number of sensor data to return. Will be ignored if below 1.	integer (int64)
Query	**newer** *optional*	Only sensor data newer than this Timestamp will be returned.	string (date-time)
Query	**older** *optional*	Only sensor data older than this Timestamp will be returned. Will be ignored if it is earlier than newer.	string (date-time)

Responses

HTTP Code	Description	Schema
200	Success	< SensorDataDTO > array
400	Bad request	No Content

Produces

- `application/json`

Tags

- sensor-data-controller

Security

Type	Name	Scopes
apiKey	**apiKey**	global

Definitions

CreateDeviceConfigDTO

Name	Schema
configuration *optional*	object

Name	Schema
type *optional*	string

CreateDeviceDTO

Name	Schema
serialNumber *optional*	string
type *optional*	string

DeviceConfigDTO

Name	Schema
configuration *optional*	object
type *optional*	string

DeviceDTO

Name	Schema
config *optional*	object
serialNumber *optional*	string
state *optional*	enum (REGISTERED, PROVISIONED, DECOMMISSIONED)
type *optional*	string
uid *optional*	string

SensorDataDTO

Name	Schema
deviceUid *optional*	string
timestamp *optional*	string (date-time)
type *optional*	enum (TEMPERATURE, LOCATION, HUMIDITY)
value *optional*	number (double)

SuccessfulLoginDTO

Name	Schema
token *optional*	string
username *optional*	string

UpdateDeviceCertificateDTO

Name	Schema
certificate *optional*	string

UpdateDeviceConfigDTO

Name	Schema
configuration *optional*	object

UpdateDeviceDTO

Name	Schema
serialNumber *optional*	string
type *optional*	string

UserLoginDTO

Name	Schema
password *optional*	string
username *optional*	string

Security

apiKey

Type : apiKey *Name* : Authorization *In* : HEADER

D. Installationsanleitung

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

CLOUD GUARD

Bachelorarbeit

Intelligente Güterwagen

Installationsanleitung - Riot v0.1.0

13. Juni 2018

Florian Bitterlin, Giuliano De Gani, Dominik Thamm

Betreuer
Prof. Beat Stettler

Industriepartner
CloudGuard Software AG, Zürich

Experte
Marco Facetti

Gegenleser
Prof. Dr. Farhad Mehta

Studiengang Informatik

Hochschule für Technik Rapperswil

Inhaltsverzeichnis

1 Einleitung

1.1 Zweck

Dieses Dokument befasst sich mit der Installation des Softwareprodukts der Bachelorarbeit/Studienarbeit unter dem Namen `Intelligente Güterwagen` welche an der Hochschule für Technik Rapperswil (HSR) im Frühjahrssemester 2018 bearbeitet wurde.

1.2 Gültigkeitsbereich

Dieses Dokument bezieht sich auf Riot in der Version `0.1.0`. Inhaltlich beschränken sich die nachfolgenden Ausführungen auf die Bachelorarbeit/Studienarbeit.

1.3 Referenzen

Die Referenzen für dieses Dokument finden sich - wo nötig - in den separaten Verzeichnissen „Abbildungsverzeichnis“, „Abkürzungsverzeichnis“ und dem „Literaturverzeichnis“ im Anhang zu diesem Dokument.

2 Übersicht

Riot kann auf verschiedene Arten deployed werden. Aufgrund seiner Microservice-Architektur kann Riot auf einer einzigen Maschine oder verteilt in einer Cloud-Umgebung aufgesetzt und benutzt werden. Das folgende Diagramm zeigt das Deployment von Riot. Die verschiedenen Services, egal ob von Riot selbst oder die benötigten Supportservices, sind nicht an eine Maschine gebunden, solange die Services unter einander erreichbar sind.

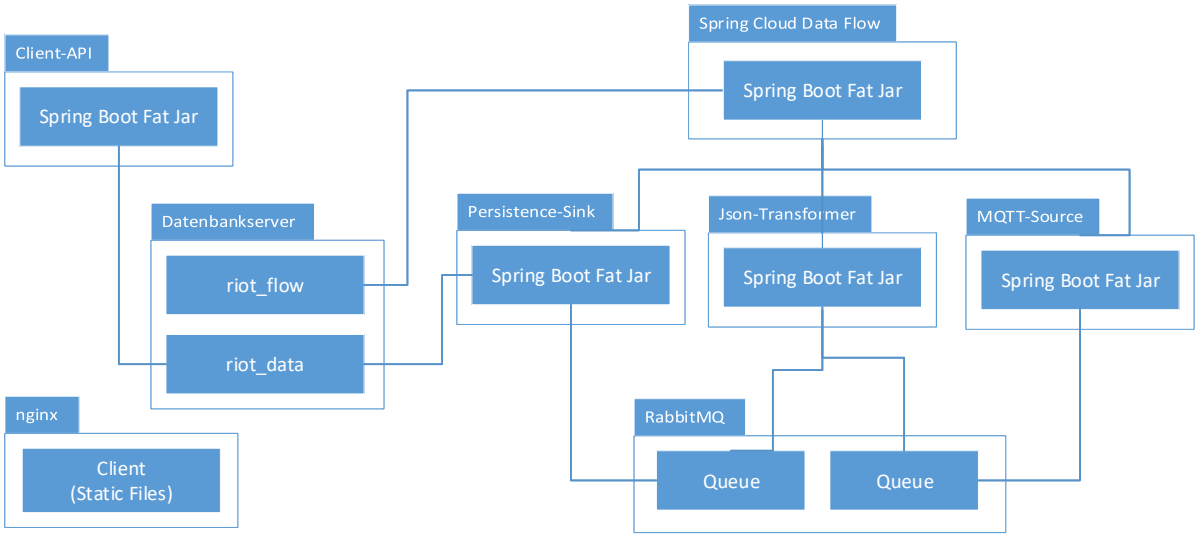


Abbildung 1: Riot Deployment Diagramm

In den folgenden Kapiteln wird aufgezeigt, wie Riot auf einer einzigen Maschine, inkl. aller Services, sowie auf einer Cloud Foundry Umgebung installiert werden kann.

3 Standalone Server

In den nachfolgenden Kapiteln findet sich die Anleitung zur Installation von Riot auf einer einzigen Maschine. Die Schritt für Schritt-Anleitungen wurden auf einem Ubuntu Server 16.04 unter dem `root`-Benutzer durchgeführt. Die Installation erfolgt hauptsächlich per Terminal.

3.1 Services

3.1.1 Datenbankserver

Die Referenz-Installation verwendet PostgreSQL als Datenbankserver. Es ist ebenso möglich eine andere relationale Datenbank einzusetzen. Dies hat jedoch zur Folge, dass die Konfiguration der Datenbankverbindung entsprechend angepasst werden muss.

Installation

PostgreSQL kann per `apt-get` installiert werden.

```
1 $> apt-get install postgresql postgresql-contrib
```

Einrichten

Folgende CLI-Befehle werden unter dem Benutzer `postgres` ausgeführt.

```
1 $> su - postgres
```

Benutzer

Erstelle eine Benutzerrolle auf PostgreSQL für die Riot-Datenbanken und setze ein entsprechendes

```
1 $> psql
2 psql=# create user riot ;
3 psql=# alter user riot with encrypted password '<password>' ;
```

Datenbanken

Riot benötigt zwei Datenbanken. `riot_data` wird das allgemeine Schema zur Haltung der Sensor- und Gerätedaten verwalten. `riot_flow` wird von Spring Cloud Data Flow benötigt um die kreierten Streams persistieren zu können.

```
psql=# create database riot_data owner riot;
psql=# create database riot_flow owner riot;
```

Privileges

Folgende Kommandos übertragen unserem vorgängig erstellten Benutzer die Privilegien an den beiden Datenbanken.

```
psql=# grant all privileges on database riot_data to riot ;
psql=# grant all privileges on database riot_flow to riot ;
```

Connect

Zum Schluss wird überprüft ob der Benutzer Zugriff auf die beiden Datenbanken hat. Ist der Zugriff auf beide möglich, kann man sich mit `exit` wieder als `postgres`-Benutzer abmelden.

```
$> psql -h localhost -p 5432 -d riot_data -U riot -W
psql=# \q
$> psql -h localhost -p 5432 -d riot_data -U riot -W
psql=# \q
$> exit
```

3.1.2 RabbitMQ

RabbitMQ ist ein Open-Source Message Broker. Er erlaubt den Spring Cloud Data Flow Applikationen die Kommunikation untereinander über Message-Queues.

Folgende Installationsinstruktionen wurden teilweise aus dem angegebenen Tutorial übernommen[1].

Installation der Dependencies

```
$> wget 'https://packages.erlang-solutions.com/erlang-solutions_1.0_all
    .deb'
$> dpkg -i erlang-solutions_1.0_all.deb
```

```
$> apt-get update
$> apt-get install erlang-nox
```

[1] https://tecadmin.net/install-rabbitmq-server-on-ubuntu/

Installation von RabbitMQ

```
$> echo 'deb http://www.rabbitmq.com/debian/ testing main' | sudo tee /
    etc/apt/sources.list.d/rabbitmq.list
$> wget -O- 'https://www.rabbitmq.com/rabbitmq-release-signing-key.asc'
     | sudo apt-key add -
```

```
$> apt-get update
$> apt-get install rabbitmq-server
```

```
$> systemctl enable rabbitmq-server
$> systemctl start rabbitmq-server
$> systemctl status rabbitmq-server
```

Benutzerkonfiguration

Ein Benutzer `guest` mit Administrator-Berechtigungen wird automatisch erstellt. Dieser Benutzer wird gelöscht. Dafür wird ein Administrator-Konto mit entsprechendem Namen und zusätzlich ein Konto für Spring Cloud Data Flow erstellt.

```
$> rabbitmqctl add_user admin <password>
$> rabbitmqctl set_user_tags admin administrator
$> rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
$> rabbitmqctl add_user flow <password>
$> rabbitmqctl set_user_tags flow policymaker
$> rabbitmqctl set_permissions -p / flow ".*" ".*" ".*"
$> rabbitmqctl delete_user guest
```

Web Management Konsole

```
$> rabbitmq-plugins enable rabbitmq_management
```

3.1.3 Nginx

Das Referenz-Deployment von Riot benutzt Nginx als Reverse-Proxy und als Webhost des Clients. Es folgen die Konfigurationen für die Services von Spring Cloud Data Flow, RabbitMQ und Riot. Für die Benutzer muss nur die Riot-Konfiguration zugänglich sein. Es wäre also möglich, die anderen Services nur berechtigten Benutzern zur Verfügung zu stellen. Es wird empfohlen alle externen Webservices per TLS abzusichern.

Installation

Nginx kann sehr einfach per `apt-get` installiert werden. Danach können die Konfigurationsdateien für die nachfolgenden Service-Konfigurationen angelegt werden.

```
$> apt-get install nginx
$> touch /etc/nginx/site-available/rabbit.conf
$> touch /etc/nginx/site-available/flow.conf
$> touch /etc/nginx/site-available/client.conf
$> ln -s /etc/nginx/site-available/rabbit.conf /etc/nginx/site-enabled/
    rabbit.conf
$> ln -s /etc/nginx/site-available/flow.conf /etc/nginx/site-enabled/
    flow.conf
$> ln -s /etc/nginx/site-available/client.conf /etc/nginx/site-enabled/
    client.conf
```

RabbitMQ

Mit `nano` kann die unten angegebene Konfiguration eingefügt werden. Die FQDN sollte unbedingt auf die richtige angepasst werden. Nach der Anpassung kann man nginx die Konfiguration neu laden lassen.

```
$> nano /etc/nginx/site-available/rabbit.conf
$> service nginx reload
```

Listing 1: rabbit.conf

```
server {
    listen 80;
    listen [::]:80;
    server_name rabbit.riot.your-domain.net;
    server_tokens off;
    access_log /var/log/nginx/rabbit_access.log;
    error_log /var/log/nginx/rabbit_error.log;
    return 302 https://$server_name$request_uri;
}
server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2;
    server_name rabbit.riot.your-domain.net;
    server_tokens off;
    root /var/www/client;
    try_files /maintenance.html @proxy;
```

```
17
18      ssl on;
19      ssl_certificate /etc/letsencrypt/live/riot.your-domain.net/
            fullchain.pem;
20      ssl_certificate_key /etc/letsencrypt/live/riot.your-domain.net/
            privkey.pem;
21
22      ssl_protocols TLSv1.2;
23      ssl_ciphers 'EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH!DSS';
24      ssl_prefer_server_ciphers on;
25      ssl_ecdh_curve secp384r1;
26      ssl_stapling on;
27      ssl_stapling_verify on;
28      ssl_session_cache shared:SSL:10m;
29      ssl_session_timeout 10m;
30
31      add_header Strict-Transport-Security "max-age=31536000;
            includeSubDomains" always;
32      add_header Referrer-Policy 'no-referrer';
33      add_header X-Content-Type-Options nosniff;
34      add_header X-Frame-Options "SAMEORIGIN";
35      add_header X-XSS-Protection "1; mode=block";
36      add_header X-Robots-Tag none;
37
38      access_log /var/log/nginx/rabbit_access.log;
39      error_log /var/log/nginx/rabbit_error.log;
40
41      location @proxy {
42          proxy_pass          http://localhost:15672;
43          proxy_redirect      http://rabbit.riot.your-domain.net https://
                rabbit.riot.your-domain.net;
44          proxy_set_header    Host $host;
45          proxy_set_header X-Real-IP $remote_addr;
46          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
47          proxy_set_header X-Forwarded-Proto $scheme;
48      }
49  }
```

Client

Mit nano kann die unten angegebene Konfiguration eingefügt werden. Die FQDN sollte unbedingt auf die richtige angepasst werden. Nach der Anpassung kann man nginx die Konfiguration neu laden

lassen.

```
$> nano /etc/nginx/site-available/client.conf
$> service nginx reload
```

Listing 2: riot.conf

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    server_name client.riot.your-domain.net;
    server_tokens off;
    access_log /var/log/nginx/riot_access.log;
    error_log /var/log/nginx/riot_error.log;
    return 302 https://$server_name$request_uri;
}
server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2;
    server_name client.riot.your-domain.net;
    server_tokens off;
    root /var/www/client;

    ssl on;
    ssl_certificate /etc/letsencrypt/live/riot.your-domain.net/
        fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/riot.your-domain.net/
        privkey.pem;

    ssl_protocols TLSv1.2;
    ssl_ciphers 'EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH!DSS';
    ssl_prefer_server_ciphers on;
    ssl_ecdh_curve secp384r1;
    ssl_stapling on;
    ssl_stapling_verify on;
    ssl_session_cache shared:SSL:10m;
    ssl_session_timeout 10m;

    add_header Strict-Transport-Security "max-age=31536000;
        includeSubDomains" always;
    add_header Referrer-Policy 'no-referrer';
    add_header X-Content-Type-Options nosniff;
    add_header X-Frame-Options "SAMEORIGIN";
```

```
34      add_header X-XSS-Protection "1; mode=block";
35      add_header X-Robots-Tag none;
36
37      access_log /var/log/nginx/riot_access.log;
38      error_log /var/log/nginx/riot_error.log;
39
40      location / {
41                  try_files $uri /index.html;
42      }
43
44      location /api/ {
45          proxy_pass          http://localhost:9000;
46          proxy_redirect      off;
47          proxy_set_header    Host $host;
48          proxy_set_header X-Real-IP $remote_addr;
49          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
50          proxy_set_header X-Forwarded-Proto $scheme;
51      }
52
53      location /status/ {
54          rewrite /status/(.*) /actuator/$1  break;
55          proxy_pass          http://localhost:9001;
56          proxy_redirect      off;
57          proxy_set_header    Host $host;
58          proxy_set_header X-Real-IP $remote_addr;
59          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
60          proxy_set_header X-Forwarded-Proto $scheme;
61      }
62  }
```

Spring Cloud Data Flow

Mit nano kann die unten angegebene Konfiguration eingefügt werden. Die FQDN sollte unbedingt auf die richtige angepasst werden. Nach der Anpassung kann man nginx die Konfiguration neu laden lassen.

```
1  $> nano /etc/nginx/site-available/flow.conf
2  $> service nginx reload
```

Listing 3: flow.conf

```
1  server {
```

```
2       listen 80;
3       listen [::]:80;
4       server_name flow.riot.your-domain.net;
5       server_tokens off;
6       access_log /var/log/nginx/flow_access.log;
7       error_log /var/log/nginx/flow_error.log;
8       return 302 https://$server_name$request_uri;
9   }
10  server {
11      listen 443 ssl http2;
12      listen [::]:443 ssl http2;
13      server_name flow.riot.your-domain.net;
14      server_tokens off;
15      root /var/www/client;
16      try_files /maintenance.html @proxy;
17
18      ssl on;
19      ssl_certificate /etc/letsencrypt/live/riot.your-domain.net/
            fullchain.pem;
20      ssl_certificate_key /etc/letsencrypt/live/riot.your-domain.net/
            privkey.pem;
21
22      ssl_protocols TLSv1.2;
23      ssl_ciphers 'EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH!DSS';
24      ssl_prefer_server_ciphers on;
25      ssl_ecdh_curve secp384r1;
26      ssl_stapling on;
27      ssl_stapling_verify on;
28      ssl_session_cache shared:SSL:10m;
29      ssl_session_timeout 10m;
30
31      add_header Strict-Transport-Security "max-age=31536000;
            includeSubDomains" always;
32      add_header Referrer-Policy 'no-referrer';
33      add_header X-Content-Type-Options nosniff;
34      add_header X-Frame-Options "SAMEORIGIN";
35      add_header X-XSS-Protection "1; mode=block";
36      add_header X-Robots-Tag none;
37
38      access_log /var/log/nginx/flow_access.log;
39      error_log /var/log/nginx/flow_error.log;
40
41      location @proxy {
```

```
42          proxy_pass          http://localhost:9393;
43          proxy_redirect      http://flow.riot.your-domain.net https://
                flow.riot.your-domain.net;
44          proxy_set_header    Host $host;
45          proxy_set_header X-Real-IP $remote_addr;
46          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
47          proxy_set_header X-Forwarded-Proto $scheme;
48      }
49  }
```

3.2 Benutzer und Berechtigungen

Riot-Services sollten unter einem eigenen Benutzer und nicht als `root`-Benutzer ausgeführt werden.

3.2.1 Benutzer

Ein `riot`-Benutzer wird mit seiner eigenen Gruppe angelegt.

```
1 $> adduser --system --group riot
```

3.2.2 Ordner

```
1 $> mkdir /opt/riot
2 $> mkdir /opt/riot/client-api
3 $> mkdir /opt/riot/dataflow
4 $> chown riot:riot -R /opt/riot
5 $> chmod 750 /opt/riot
6 $> chmod 770 /opt/riot/client-api
7 $> chmod 770 /opt/riot/dataflow
```

3.3 Client-API

Die Client-API wird als Systemd-Service auf der Maschine eingerichtet. Die Installation folgt dabei den Vorschlägen der Spring-Boot-Referenz[2].

3.3.1 Installation

Zuerst wird ein neuer Systemd-Service erstellt.

```
$> nano /etc/systemd/system/riot-client-api.service
```

Das Fat Jar wird dabei im Kapitel Benutzer erstellten dafür vorgesehenen Ordner abgelegt.

Listing 4: riot-client-api.service

```
[Unit]
Description=riot-client-api
After=syslog.target

[Service]
User=riot
WorkingDirectory=/opt/riot/client-api
ExecStart=/opt/riot/client-api/riot-client-api.jar
SuccessExitStatus=143
TimeoutStopSec=10
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

3.3.2 Konfiguration

Mit folgendem Kommando kann der Service über die Umgebungsvariablen individuell konfiguriert werden.

```
$> systemctl edit riot-client-api
```

Das folgende Beispiel zeigt die Konfiguration der Datenbank.

[2] https://docs.spring.io/spring-boot/docs/current/reference/html/deployment-install.html

```
[Service]
Environment=SPRING_PROFILES_ACTIVE=production
Environment=SPRING_DATASOURCE_URL='jdbc:postgresql://localhost:5432/
    riot_data'
Environment=SPRING_DATASOURCE_USERNAME=riot
Environment=SPRING_DATASOURCE_PASSWORD=<password>
Environment=SPRING_DATASOURCE_DRIVER_CLASS_NAME=org.postgresql.Driver
Environment=SPRING_JPA_DATABASE_PLATFORM=org.hibernate.dialect.
    PostgreSQLDialect
Environment=SPRING_JPA_HIBERNATE_DDL_AUTO=update
```

3.3.3 Service starten

`daemon-reload` bewirkt, dass die vorhergehenden Konfigurationsänderungen beim nächsten Neustart des Service übernommen werden. `enable` registriert den Service als solcher, welcher beim Systemstart mit gestartet werden soll.

```
$> systemctl daemon-reload
$> systemctl enable riot-client-api.service
$> systemctl start riot-client-api.service
$> systemctl status riot-client-api.service
```

3.3.4 Liste der eigenen Umgebungsvariablen

Die Client-API verlangt, bzw. stellt neben den Standart-Spring-Variablen folgende weitere Environmentvariablen zur Konfiguration zur Verfügung.

Name	Pflicht	Beschreibung
IOT_PROVIDER	Ja	Entweder „dummy“ oder „aws“. Entscheided welcher Iot-Service beim Start der Anwendung geladen wird.
IOT_ACCESSKEYID	Ja	AccessKeyId welcher zur Authetifizierung des IoT-Service
IOT_SECRETKEY	Ja	AccessKeyId welcher zur Authetifizierung des IoT-Service

Name	Pflicht	Beschreibung
JWT_EXPIRATION	Nein	Timeout in Sekunden bis ein Json Web Token ungültig nach seiner Austellung ungültig wird. Standart: 72’800’000s
JWT_TOKENPREFIX	Nein	Prefix welches dem Token im Header vorangestellt werden muss. Standart: Bearer
JWT_TOKENHEADER	Nein	HTTP-Header in welchem das JWT transportiert wird. Standart: Authorization
JWT_SECCRET	Ja	Secret zur Verschlüsslung des JWT vor der Übertragung.
LDAP_BASE_DN	Ja	Base-dn unter welchem im LDAP-Verzeichnis nach Benutzern gesucht wird.
LDAP_CREDENTIAL_USERNAME	Ja	DN des LDAP-Benutzers mit welchem sich die API gegenüber dem LDAP authentifizieren sollte.
LDAP_CREDENTIAL_PASSWORD	Ja	Passwort des LDAP-Benutzers
LDAP_URL	Ja	Url des LDAP-Servers dem gegenüber authentifiziert werden muss. Z.B.: ldap://localhost:8389

3.4 Spring Cloud Data Flow

Spring Cloud Data Flow wird als Systemd-Service auf der Maschine eingerichtet. Ergänzende Angaben finden sich in der Spring Cloud Data Flow Referenz[3].

Es wird vorausgesetzt, das folgende Services bereits installiert sind:

- PostgreSQL
- RabbitMQ
- Nginx

3.4.1 Installation

Download

Spring Cloud Data Flow kann in verschiedenen Varianten aus dem Internet bezogen werden. Riot v0.1.0 wurde mit Spring Cloud Data Flow-1.5.0.RELEASE getestet und wird hier in der local-Variante benötigt.

```
1 $> wget 'https://repo.spring.io/libs-snapshot/org/springframework/cloud
     /spring-cloud-dataflow-server-local/1.5.0.RELEASE/spring-cloud-
     dataflow-server-local-1.5.0.RELEASE.jar'
2 $> mv spring-cloud-dataflow-server-local-1.5.0.RELEASE.jar /opt/riot/
     dataflow/scdf-1.5.jar
3 $> chown riot:riot -R /opt/riot/dataflow
```

Service erstellen

Die Registrierung des Data-Flow-Service ist äquivalent zum Service für die Client-API.

```
1 $> nano /etc/systemd/system/dataflow.service
```

Listing 5: dataflow.service

```
1 [Unit]
2 Description=dataflow
3 After=syslog.target
4
5 [Service]
6 User=riot
7 WorkingDirectory=/opt/riot/dataflow
8 ExecStart=/usr/bin/java -jar /opt/riot/dataflow/scdf-1.5.jar
```

[3] https://docs.spring.io/spring-cloud-dataflow/docs/1.5.0.RELEASE/reference/htmlsingle/

```
SuccessExitStatus=143
TimeoutStopSec=10
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Service konfigurieren

Spring Cloud Data Flow lädt registrierte Stream-Applikationen extern nach. Applikations-Artefakte, also die gebauten Jars, müssen daher über eine auflösbare URI erreichbar sein. Dies kann per HTTP, das Filesystem oder ein Maven-Repository erfolgen. Falls ein eigenes Maven-Repository verwendet werden soll, ist es wichtig, dass in der Konfiguration die Pfade dazu angegeben werden. Ansonsten können die diese Stream-Applikationen nicht gefunden werden.

```
$> systemctl edit dataflow
```

```
[Service]
Environment=spring_datasource_url=jdbc:postgresql://localhost:5432/
    riot_flow
Environment=spring_datasource_username=riot
Environment=spring_datasource_password=<password>
Environment=spring_datasource_driver_class_name=org.postgresql.Driver
Environment=spring_rabbitmq_host=127.0.0.1
Environment=spring_rabbitmq_port=5672
Environment=spring_rabbitmq_username=flow
Environment=spring_rabbitmq_password=<password>
Environment=security_basic_enabled=true
Environment=security_user_name=riot
Environment=security_user_password=<password>
Environment=security_user_role=VIEW,CREATE,MANAGE
Environment=maven_remote_repositories_repo1_url=https://repo.your-
    domain.net/libs-snapshot
Environment=maven_remote_repositories_repo2_url=https://repo.your-
    domain.net/libs-release
```

Service starten

```
$> systemctl daemon-reload
```

```
$> systemctl enable dataflow.service
$> systemctl start dataflow.service
$> systemctl status dataflow.service
```

3.4.2 Stream definieren und deployen

Nachdem Spring Cloud Data Flow als Service gestartet wurde, kann die URI, welche per Nginx definiert wurde, im Browser aufgerufen werden.

Applikationen registrieren

Über `Apps > Add Applications` können mehrere Stream-Applikationen als Bulk importiert werden. Dazu kann der Inhalt der folgenden Datei kopiert und als Properties importiert werden.

Listing 6: riot-stream-apps.descriptor

```
sink.riot-persistence-sink='maven://ch.hsr.riot:persistence-sink:jar
    :0.1.0-RELEASE'
processor.riot-model-transformer='maven://ch.hsr.riot:json-transformer:
    jar:0.1.0-RELEASE'
source.riot-aws-mqtt='maven://ch.hsr.riot:aws-mqtt-source:jar:0.1.0-
    RELEASE'
```

Stream definieren

Nachdem der Import der Applikationen geklappt hat, können diese für die Definition von Streams benutzt werden. Riot v0.1.0 benötigt lediglich einen Stream zur Verarbeitung und Persistierung von Sensordaten.

Unter `Streams > Create Stream` lässt sich dieser erstellen. Streams lassen sich entweder mit der Maus per Drag-and-drop zusammenklicken und die einzelnen Applikationen per Dialog konfigurieren. Andererseits können sie auch in ihrer eigenen DSL namens Flo definiert werden. Alle Streams müssen benannt werden. Die Namen dienen nur dem Benutzer zur Identifikation der Streams und haben ansonsten keine Funktion.

Der für Riot benötigte Stream besteht aus einer AWS-MQTT-Source welche sich auf das Temperatur-Topic subscribed. Die so abgegriffenen Datenströme sollen dann an den Json-Transformer weitergeleitet werden, welcher das vom Gerät übertragene Json in Java-Objekte transformiert. Das Persistence-Sink speichert die so erhaltenen Daten am Ende des Streams in die Datenbank.

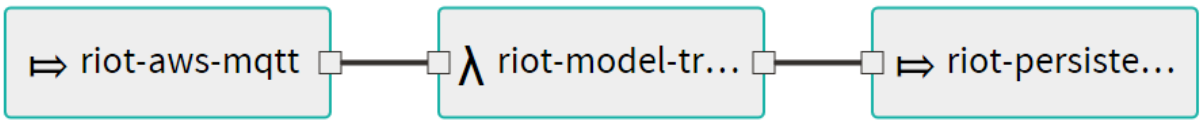


Abbildung 2: Definition des Riot-Data-Flows

Nachfolgend findet sich eine Definition des benötigten Stream in der Flo-DSL.

Listing 7: Riot-Flow Stream in der Flo-DSL definiert

```
riot-aws-mqtt --topics=/data/+/temp --persistence=memory --ca-file-path
    =/home/riot/.ssh/ca_root.pem --client-id=2 --client-key-file-path=/
    home/riot/.ssh/key_client.key --client-crt-file-path=/home/riot/.ssh
    /cert_client.pem --url='ssl://ao5hia4odanw4.iot.eu-central-1.
    amazonaws.com:8883' | riot-model-transformer | riot-persistence-sink
     --password='******' --driver-class-name=org.postgresql.Driver --url
    ='jdbc:postgresql://127.0.0.1:5432/riot_data --username=riot'
```

Speziell sind die im Beispiel vorliegenden Konfigurationen zu beachten:

- Das CA-Zertifikat, das Client-Zertifikat und der entsprechende Private-Key müssen im PEM-Format abgespeichert auf der Festplatte vorhanden sein. Die Pfade müssen entsprechend angepasst werden.
- Die Url zu AWS muss auf die eigene angepasst werden.
- Das Datenbank Passwort muss entsprechend angepasst werden.

Stream deployen

Wurde der Stream definiert, kann dieser und dem Tab `Streams` durch Klick auf den Play-Button deployed werden. Dies bedeutet, Spring Cloud Data Flow startet die drei involvierten Applikationen als eigene Prozesse auf der Maschine und überwacht deren Gesundheitszustand. Der Zustand des Streams sollte nach ein paar Minuten seinen Zustand auf `DEPLOYED` wechseln.

3.5 Client

Zur Installation des Clients muss `npm` zur Verfügung stehen. Um den Client auf ein neues Deployment anzupassen, muss entsprechend der Reverse-Proxy-Definition die API- und ACTUATOR_BASEURL in der Datei `webpack.prod.js` angepasst werden.

Listing 8: Client API-Url Definition

```
new webpack.DefinePlugin({
  API_BASEURL: "/api/",
  ACTUATOR_BASEURL: "/status/",
  ...
}),
```

Durch einen erneuten Build mit Webpack werden die neuen API-Urls als Konstanten im Code eingefügt und der Client ist für das Deployment konfiguriert. Der Inhalt des `dist`-Ordners stellt das Build-Produkt dar.

```
$> cd src
$> npm run build:prod
$> mv -rf ./dist /var/www/client
```

3.6 AWS IoT

Um den IoT Provder AWS mit der Plattform verwenden zu können müssen folgende Schritte unternommen werden:

Erstellen eines AWS Account: Folge den Anweisungen unter https://aws.amazon.com/.

Anlegen eines Benutzer für die Plattform: Nach dem einloggen in die AWS Console[4] können unter `Services > IAM` (unter dem Punkt `Security, Identity & Compliance`) neue Benutzer angelegt werden. Erstelle einen Benutzer für die Plattform mit `Programmatic access`. Der Benutzer braucht die Berechtigungen `CloudWatchReadOnlyAccess` und `AWSIoTFullAccess`. Der generierte AccessKey kann dann in der Plattform hinterlegt werden.

CA registrieren: Falls gewünscht kann die eigene CA in AWS registriert werden. Die Anleitung dazu ist im Kapitel Umsetzung IoT-Provider zu finden.

Endpunkt einsehen: Der von der Plattform und Geräten benötigte Endpunkt kann unter `Services > Iot Core > Settings` eingesehen werden. Beachte, dass sich dieser von Region zu Region unterscheidet.

[4] https://console.aws.amazon.com

4 Cloud Foundry

In den nachfolgenden Kapiteln findet sich die Installationsanleitung zur Installation von Riot auf Cloud Foundry. Die Swisscom Application Cloud dient als Beispiel einer Cloud Foundry Instanz für diese Installationsanleitung. Aufgrund von Unterschieden je nach Anbieter können andere Cloud Foundry Instanzen ein Abweichen von dieser Anleitung verlangen.

4.1 Vorbedingungen

Diese Installationsanleitung geht davon aus, dass bereits eine Cloud Foundry Organization und ein Space sowie ein Account mit mindestens Space-Developer-Berechtigungen vorhanden sind.

Das Cloud Foundry CLI muss installiert sein[5].

Bei allen nachfolgenden Cloud Foundry CLI Kommandos muss zuerst das Login erfolgen.

```
1 $> cf login -a 'https://api.lyra-836.appcloud.swisscom.com' -u your.
     account@email.net
```

4.2 Services

Riot benötigt verschiedene Services welche von Cloud Foundry zur Verfügung gestellt werden, vorher aber noch bezogen werden müssen. Benötigt werden eine Datenbank zur Speicherung der Applikationsdaten von Riot. Sowie ein RabbitMQ-Service und eine Datenbank für Spring Cloud Data Flow.

Der bezug dieser Services kann über das CF-CLI erfolgen:

```
1 $> cf create-service mariadbent usage riot-data
2 $> cf create-service mariadbent usage flow-data
3 $> cf create-service rabbitmqent usage flow-rabbit
```

4.3 Client-API

Für die Swisscom Application Cloud wurde die Riot-Client-API bereits für den Einsatz mit einer MariaDB entsprechend konfiguriert. Soll die Installation auf einer Cloud Foundry Instanz eines anderen Anbieters durchgeführt werden, muss die Konfiguration entsprechend angepasst werden. Dies kann entweder durch das Anpassen des Cloud-Profils der Client-API geschehen oder durch das das Yaml-Manifest. Wird das Cloud-Profil der Applikation angepasst, muss diese mit Gradle neu gebaut und deployed werden.

[5] https://docs.developer.swisscom.com/cf-cli/install-go-cli.html

4.3.1 Konfiguration

Die Konfiguration für die Swisscom Application Cloud sieht wie folgt aus:

Listing 9: application-cloud.yml

```yaml
spring:
  jpa:
    database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        show_sql: false
        use_sql_comments: false
        format_sql: false
        dialect: org.hibernate.dialect.MySQL5InnoDBDialect

iot:
  provider: aws
  accessKeyId: ABCDEF
  secretKey: ABCDEF
```

Im Manifest-Yaml können Umgebungsvariablen gesetzt werden um die Applikation entsprechend anzupassen[6].

4.3.2 Deployment der Client-API

Die Definition einer Applikation welche auf Cloud Foundry deployed werden soll, erfolgt durch eine `manifest.yml`-Datei.

Listing 10: Client-API - Cloud Foundry - manifest.yml

```yaml
---
applications:
- name: riot-client-api
  memory: 2G
  instances: 1
  buildpack: java_buildpack
  path: build/libs/client-api-0.1.0-RELEASE.jar

  routes:
```

[6] https://docs.developer.swisscom.com/devguide/deploy-apps/manifest.html#env-block

```
10    - route: riot-client-api.scapp.io
11
12    services:
13    - riot-data
```

Darin müssen je nachdem folgende Parameter angepasst werden.

- **name**: Muss innerhalb des Space eindeutig sein.
- **path**: Muss auf das lokal vorhanden Jar der Client-API zeigen.
- **route**: Darf nicht bereits durch eine andere Applikation belegt sein.
- **services**: Benötigt mindestens einen Datenbank-Service. Der Service wird hier durch seinen Namen referenziert.

Wurden die vorhergehenden Parameter entsprechend angepasst, kann die Client-API nun deployed werden.

```
1  $> cf push
```

4.4 Spring Cloud Data Flow

Spring Cloud Data Flow benötigt einen Cloud Foundry Account mit mindestens Space-Developer-Berechtigungen um selber Applikation deployen zu können[7].

4.4.1 Download

Spring Cloud Data Flow kann in verschiedenen Varianten aus dem Internet bezogen werden. Riot v0.1.0 wurde mit Spring Cloud Data Flow-1.5.0.RELEASE getestet und wird hier in der Cloud-Foundry-Variante benötigt.

```
1  $> wget 'https://repo.spring.io/release/org/springframework/cloud/
       spring-cloud-dataflow-server-cloudfoundry/1.5.0.RELEASE/spring-cloud
       -dataflow-server-cloudfoundry-1.5.0.RELEASE.jar'
```

4.4.2 Deployment von Cloud Foundry

Die Definition des Deployments von Data Flow auf Cloud Foundry erfolgt äquivalent zur Client-API. Die Konfiguration erfolgt ähnlich wie bei der Standalone Maschine, wird hier aber im Manifest-Yaml vorgenommen.

[7] https://docs.developer.swisscom.com/concepts/roles.html#roles

Listing 11: Spring Cloud Data Flow Cloud Foundry manifest.yml

```
---
applications:
- name: riot-data-flow
  memory: 2G
  disk_quota: 4G
  instances: 1
  path: ./spring-cloud-dataflow-server-cloudfoundry-1.5.0.RELEASE.jar
  env:
    SPRING_APPLICATION_NAME: riot-data-flow
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_URL: https://api.lyra-836.
        appcloud.swisscom.com
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_ORG: HSR-Riot
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SPACE: riot
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_DOMAIN: scapps.io
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_USERNAME: your.user@email.net
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_PASSWORD: <password>
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_STREAM_SERVICES: flow-rabbit
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_TASK_SERVICES: flow-data
    SPRING_CLOUD_DEPLOYER_CLOUDFOUNDRY_SKIP_SSL_VALIDATION: false
    SPRING_APPLICATION_JSON: '{ "security.basic.enabled": "true", "
        security.user.name": "riot", "security.user.password": "rolling
        ", "security.user.role": "VIEW,CREATE,MANAGE", "maven": { "
        remote-repositories": { "repo1": { "url": "https://repo.riot.
        bitterlin.net/libs-snapshot", "policy_update-policy": "always"},
         "repo2": { "url": "https://repo.riot.bitterlin.net/libs-release
        ", "policy_update-policy": "always"}, "repo3": { "url": "https
        ://repo.spring.io/libs-release"} } } }'

  routes:
  - route: riot-data-flow.scapp.io

  services:
  - flow-data
  - flow-rabbit
```

Folgende Parameter im Manifest individuell angepasst werden.

- **name**: Muss innerhalb des Space eindeutig sein.
- **path**: Muss auf das lokal vorhanden Jar von Data Flow zeigen.
- **route**: Darf nicht bereits durch eine andere Applikation belegt sein.
- **services**: Benötigt mindestens einen Datenbank- und ein RabbitMQ-Service. Die Service werden

hier durch ihre Namen referenziert.

Wurden die vorhergehenden Parameter entsprechend angepasst, kann Data Flow nun deployed werden.

```
$> cf push
```

4.4.3 Stream definieren und deployen

Nachdem Spring Cloud Data Flow als Service gestartet wurde, kann die URI, welche per definierter Router, im Browser aufgerufen werden.

Das Importieren von Applikationen und das Erstellen von Streams erfolgt in der Cloud Foundry Variante praktisch identisch wie auf der lokalen Variante. Unterschiede ergeben sich jedoch in der Konfiguration des Streams.

Nachfolgend findet sich die Definition des benötigten Stream in der Flo-DSL.

Listing 12: Riot-Flow Stream in der Flo-DSL definiert

```
riot-aws-mqtt --topics=/data/+/temp --persistence=memory --ca-cert='PEM
    ' --client-id=2 --client-key='PEM' --client-crt='PEM' --url='ssl://
    ao5hia4odanw4.iot.eu-central-1.amazonaws.com:8883' | riot-model-
    transformer | riot-persistence-sink
```

Da auf Cloud Foundry den Applikationen kein persistentes Filesystem zur Verfügung steht, welches einen Neustart übersteht, können die Zertifikate zur Authentifizierung gegenüber AWS nicht darin abgespeichert werden. Die kann mit einem NFS-Service umgangen werden. Da die Swisscom Application Cloud dies jedoch nicht anbietet, müssen die Zertifikate als Properties mitgegeben werden. Wie in der obigen Stream-Definition angegeben, werden die Zertifikate im PEM-Format erwartet. Sie dürfen keine Zeilenumbrüche enthalten.

Die Konfiguration des Persistence-Sinks erfolgt während des Deployen des Streams. Dazu muss als auf dem Sink das Deployment-Property `cloudfoundry.services` mit dem Wert `riot-data` gesetzt werden. Dadurch wird dem Sink dieselbe MariaDB-Instanz zur Verfügung gestellt wie der Client-API und Abspeichern der prozessierten Sensordaten ist von nun an möglich.

4.5 Client

Um den Client zu deployen wird das Staticfile-Buildpack benutzt[8]. Dieses verwendet einen Nginx-Server als Webserver. Dieser Nginx-Server kann durch Angaben in der Datei `Staticfile` konfiguriert

[8] https://docs.cloudfoundry.org/buildpacks/staticfile/index.html

werden. Durch die Konfiguration des `pushstate`- Routing wird den Nginx-Server entsprechend für eine Single Page Application konfiguriert.

Listing 13: Staticfile

```
pushstate: enabled
```

Die restliche Definition der erfolgt durch eine `manifest.yml`-Datei.

Listing 14: Client - Cloud Foundry - manifest.yml

```
---
applications:
- name: riot-client
  memory: 256M
  disk: 48M
  instances: 1
  buildpack: staticfile_buildpack

  routes:
  - route: riot-client.scapp.io
```

Darin müssen je nachdem folgende Parameter angepasst werden.

- **name**: Muss innerhalb des Space eindeutig sein.
- **route**: Darf nicht bereits durch eine andere Applikation belegt sein.

Die Konfiguration der Client-API im Client erfolgt gleich wie beim Deploment auf einer einzelnen Maschine. Die URI der API wäre in diesem Fall nach der der Konfiguration der CLient-API also https://riot-client-api.scapp.io/api.

Sowohl das Manifest-Yaml sowie die `Staticfile`-Datei müssen im selben Ordner wie die Artefakte des Webpack-Builds abgelegt werden. Danach kann der Client aus diesem Ordner auf Cloud Foundry deployed werden.

```
$> cf push
```

Codeverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

E. Testprotokoll

Testprotokoll

Vorraussetzungen

Bevor mit der Testdurchführung begonnen wird, müssen folgende Vorbedingungen erfüllt sein:

- Die Testperson hat einen aktuellen Webbrowser, ausgenommen Internet Explorer oder Edge, mit Internetanschluss eingerichtet.

Bemerkungen

- Die Tests leiten sich mehrheitlich aus den Anforderungen ab, insbesondere den Use cases sowie nicht funktionalen Anforderungen.
- Für nicht implementierte Funktionalitäten im Rahmen der Projektarbeit sind keine Testfälle abgebildet, da diese nicht getestet werden können.
- Aufgrund der Prototypisierung wurden für das Userinterface keine expliziten Anforderungen erhoben, da diese einem raschen Wandel oder Weiterentwicklung ausgesetzt sind. Testfälle sind darum, wenn möglich, Userinterface neutral beschrieben.

Tests

T1: Intialisierung

Ausgangslage

Der Browsercache ist gelöscht.

Verhalten

GET https://client.riot.bitterlin.net

Endzustand SOLL

Der Browser wurde auf https://client.riot.bitterlin.net/login weitergeleitet und das Loginmenü erscheint.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T2: Login

Ausgangslage

Der Browsercache ist gelöscht.

Verhalten

POST https://client.riot.bitterlin.net/login mit Benutzername „riot“ und Passwort „rolling“

Endzustand SOLL

Statuscode 200. Der Browser wurde auf https://client.riot.bitterlin.net weitergeleitet und der Benutzer befindet sich auf der Ansicht der Devices.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T3: Registrieren eines Devices

Ausgangslage

Der Benutzer hat sich erfolgreich angemeldet und befindet sich auf der Seite der Devices.

Verhalten

Der Benutzer registriert ein neues Device mit dem existierenden Type „test“ und einer Serialnumber, welche keine Sonderzeichen und Leerzeichen enthält.

Endzustand SOLL

Statuscode 201. Das neu registrierte Device erscheint im UI. Der Status ist „Registered“. Die angezeigte Serialnumber entspricht der Eingabe beim Registrieren.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T4: Gerätekonfiguration anpassen

Ausgangslage

Der Benutzer hat sich erfolgreich angemeldet und befindet sich auf der Seite der Devices. Es existiert ein Device im Status „Registered“. Das Configure Device Formular eines registrierten Devices wurde geöffnet.

Verhalten

Über das Configure Device Formular wird ein neues Key Value Paar eingetragen, welches noch nicht existiert. Der Schlüssel hat den Prefix „device.“ . Der Configure Button wird betätigt.

Endzustand SOLL

Statuscode 200. Das eingetragene Key Value Paar erscheint als Zeile in der Configuration.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T5: Gerätekonfiguration zurücksetzen

Ausgangslage

Der Benutzer hat sich erfolgreich angemeldet und befindet sich auf der Seite der Devices. Es existiert ein Device im Status „Registered“. Das Configure Device Formular eines registrierten Devices wurde geöffnet.

Verhalten

Über das Configure Device Formular wird zunächst ein neues Key Value Paar hinzugefügt und anschliessend ein Reset an der Configuration durchgeführt, indem auf den Reset Button gedrückt wird.

Endzustand SOLL

Statuscode 200. Die nicht gespeicherten Änderungen wurden verworfen und es sind die ursprünglich eingetragenen Werte in der Configuration enthalten.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T6: Gerätekonfiguration anlegen

Ausgangslage

Der Benutzer hat sich erfolgreich angemeldet und befindet sich auf der Seite der Gerätetypen https://client.riot.bitterlin.net/types.

Verhalten

Über den Hinzufügen Button wird ein neue Device Configuration angelegt. Als Name wird ein nicht existierender Bezeichner eingetragen.

Endzustand SOLL

Statuscode 200. Die neu hinzugefügte Configuration hat den Bezeichner und die Key Value Paare, die eingetragen wurden. Zusätzlich ist der „device.isActive“ Eintrag enthalten.

Zusätzlich ist die neu hinzugefügte Configuration beim Registrieren eines Devices als neuer Type verfügbar.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T7: Gerätekonfiguration löschen

Ausgangslage

Der Benutzer hat sich erfolgreich angemeldet und befindet sich auf der Seite der Gerätetypen https://client.riot.bitterlin.net/types.

Verhalten

Über den Löschen Button einer Configuration wird die Standard Configuration gelöscht.

Endzustand SOLL

Vor dem DELETE wird der Benutzer vor dem Löschen gewarnt.

Nach dem Löschen der Configuration ist diese mit GET https://client.riot.bitterlin.net/types nicht mehr verfügbar. Ebenso ist sie nicht mehr im Register Device Formular als Device type verfügbar.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T8: Betriebsdaten auslesen

Ausgangslage

Der Benutzer hat sich erfolgreich angemeldet und hat ein Gerät damit vorkonfiguriert, so dass es mit dem IoT Provider kommunizieren kann (Client Zertifikat installiert). Weiter wurde das Device in Betrieb genommen und ist online.

Verhalten

Auf den Data Button klicken, um Sensordaten anzuzeigen.

Endzustand SOLL

Der Benutzer erhält wahlweise eine Auflistung oder grafische Repräsentation der gesendeten Betriebsdaten des Devices. Jeder Datenpunkt besitzt einen Zeitstempel, einen Wert und ist genau einem Device zugeordnet.

Standardmässig sind die Sensordaten absteigend nach Zeitpunkt sortiert, d.h. zuoberst befinden sich die neusten Daten.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T9: Betriebsdaten eines Devices exportieren

Ausgangslage

Der Benutzer hat sich erfolgreich angemeldet und hat ein Gerät vorkonfiguriert, so dass es mit dem IoT Provider kommunizieren kann (Client Zertifikat installiert). Weiter wurde das Device in Betrieb genommen und ist online. Der Benutzer befindet sich in der Ansicht für die Visualisierung der Betriebsdaten.

Verhalten

Der Export Button wird gedrückt.

Endzustand SOLL

Der Browser öffnet einen File Download Dialog, in welchem eine *.csv Datei heruntergeladen werden kann. Die Datei ist mit Microsoft Excel oder einem gängigen OpenOffice Pendant auslesbar. Die Datei enthält die Auflistung aller Datenpunkte, welche mit dem aktuellen Device assoziiert sind.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

T10: Betriebsdaten aller Devices exportieren

Ausgangslage

Der Benutzer hat sich erfolgreich angemeldet und hat mindestens ein Gerät vorkonfiguriert, so dass es mit dem IoT Provider kommunizieren kann (Client Zertifikat installiert). Weiter wurde mindestens ein Device in Betrieb genommen und ist online. Der Benutzer befindet sich in der Ansicht der Devices.

Verhalten

Über das Burgermenü wird der Export Button betätigt oder der Benutzer navigiert auf https://client.riot.bitterlin.net/export.

Es wird der Download Button betätigt, ohne weitere Einstellungen vorzunehmen.

Endzustand SOLL

Der Browser öffnet einen File Download Dialog, in welchem eine *.csv Datei heruntergeladen werden kann. Die Datei ist mit Microsoft Excel oder einem gängigen OpenOffice Pendant auslesbar. Die Datei enthält die Auflistung aller Datenpunkte aller Devices.

Endzustand IST

[Wird vom Tester/der Testerin eingetragen.]

F. Testprotokoll - 10.06.2018

Testprotokoll

Testumgebung

Firefox Quantum 60.0, Windows 10

Testperson

Giuliano De Gani

Testfälle

Testfall	Zustand IST / Bemerkungen
T1: Intialisierung	Das Loginfenster erscheint. Browser URL : https://client.riot.bitterlin.net/login
T2: Login	Statuscode 200 (Firefox Debugger). Man landet auf der Seite der Geräte.
T3: Registrieren eines Devices	neues Device mit „testprotokoll“ als Serialnumber und „test“ als Type konnte hinzugefügt werden. Es hat den Status „Registered“. Response code „Created“ (201) (Firefox Debugger)
T4: Gerätekonfiguration anpassen	IST Zustand deckungsgleich mit SOLL Zustand. Die Anpassungen an der Device Config über das Formular ist intuitiv und leicht zu bedienen aber etwas umständlich, da man nicht weiss, welche Regeln im Hintergrund gelten (Prefix) oder welche Schlüssel tatsächlich verwendet werden.
T5: Gerätekonfiguration zurücksetzen	Deckungsgleich mit SOLL Zustand. „Reset“ könnte man auch auffassen als das Zurücksetzen in die Ur-konfiguration (wie die Configuration als Schablone hinterlegt ist) und nicht nur als „Undo“ Operation. Hier wäre ein Renaming sinnvoll.
T6: Gerätekonfiguration anlegen	gem. SOLL.
T7: Gerätekonfiguration löschen	gem. SOLL. Der Abfangdialog schafft Klarheit darüber, was mit Geräten geschieht, die mit dieser Schablone als Type erfasst wurden.

Testfall	Zustand IST / Bemerkungen
T8: Betriebsdaten auslesen	Deckungsgleich mit SOLL Zustand. Die Filterung der Daten ist umständlich, weil die Timestamps von Hand eingetragen werden müssen.
T9: Betriebsdaten eines Devices exportieren	gem. SOLL. Die resultierende csv Datei enthält sehr viel redundante Daten, wie z.b. die UUID des Gerätes, welches sich für jeden Datenpunkt wiederholt. Hier bietet sich ein Refactoring an, damit bei grossen Datenmengen nicht unnötig viel Speicher benötigt wird.
T10: Betriebsdaten aller Devices exportieren	gem. SOLL. Siehe Bemerkung in T9

Beurteilung der Resultate

In allen Punkten sind die Testergebnisse mit dem Soll Zustand deckungsgleich. Das UI implementiert an den wichtigen Stellen eine Validierung oder warnt den Benutzer vor dem Löschen. Dies entspricht unseren nicht funktionalen Anforderungen Safety, Operability und Recoverability. Es sind weitere Convenience-Features im UI sinnvoll, wie z.B. bei der Filterung der Sensordaten oder beim Erfassen eines neuen Key-Value pair in den Konfigurationen. Das API bietet aktuell keine Pagination für die Sensordaten, was zu einer längeren Wartezeit im UI führt, wenn viele Daten vorhanden sind.