HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

adnexo

LoRaWAN-Management Plattform

Bachelorarbeit

Frühjahrssemester 2019

Abteilung Informatik
HSR Hochschule für Technik Rapperswil

Autoren: Matthias Dunkel, Raffael Vögeli
Betreuer: Prof. Beat Stettler
Gegenleser: Prof. Dr. Thomas Bocek
Experte: Michael Schneider, CloudGuard Software AG
Projektpartner: adnexo GmbH

Inhaltsverzeichnis

Part I

Aufgabenstellung

Ausgangslage

Das Netzwerkprotokoll LoRaWAN ist auf Grund seiner Eigenschaften ideal für IoT-Applikationen geeignet, weshalb diese Technologie von immer mehr Unternehmen in der Schweiz verwendet wird, um Daten von analogen Gegenständen ins Internet zu bringen. Dabei stellen LoRaWAN Netzwerk Provider wie Swisscom, Loriot, TheThingsNetwork etc. die übertragenen Daten an einem sogenannten Netzwerkserver zum Abholen bereit. Die dabei verwendeten Protokolle (HTTPS, MQTT, WebSockets, etc.) und Nachrichten-Spezifikationen sind je nach provider unterschiedlich. Das bedeutet, das Client Applikationen für jeden Netzanbieter eigene Schnittstellen implementieren müssen, was die Komplexität und den Aufwand zur Umsetzung von IoT-Projekten unnötig erhöht.

Aufgabenstellung

In dieser Bachelorarbeit soll ein Proxy-Service implementiert werden, der mind. zwei Netzanbieter über ihre spezifischen Schnittstellen anbindet und die übertragenen Sensordaten den Client Applikationen über eine einheitliche Schnittstelle zur Verfügung stellt. Dazu müssen Sensor- und Metadaten von den providerspezifischen Formaten und Protokollen entgegengenommen, ausgewertet und in ein einheitliches Format an die Client Applikationen weitergeben. Umgekehrt müssen Daten der Client Applikationen gesichert an die Sensoren ausgeliefert werden können. Sämtliche Kommunikation soll verschlüsselt erfolgen. Neue Sensoren sollen von Client Applikationen beim Proxyservice registriert und überwacht werden können. Eine Alarmierung meldet fehlerhaftes Verhalten der Providernetze oder Sensoren. Der Proxy-Service muss auf mehrere 10'000 Geräte skalieren

Vorgehen

Im allgemeinen theoretischen Teil sollen IoT Netztechnologien miteinander verglichen und die optimalen Einsatzgebiete eines LoRaWAN Netzwerkes herausgearbeitet werden. Aus den spezifischen LoRaWAN Schnittstellen (Funktionen, Daten, Metadaten) der verschiedenen Provider muss eine einheitliche Schnittstelle für die Clientapplikationen definiert und implementiert werden. Die Verarbeitung von Sensordaten mehrerer 10'000 Sensoren bedingt eine skalierbare Software Architektur. Die Verwaltung (CRUD) der Sensoren sowie deren Überwachung sollte möglichst in Realtime erfolgen.

Prof. Beat Stettler

Part II

Abstract

LoRaWAN ist eine Technologie, die verwendet wird, um mit Internet of Things (IoT) Geräten zu kommunizieren. Dabei gibt es verschiedene Anbieter, welche ein LoRaWAN-Netz bereitstellen.

Da LoRaWAN nur den Weg bis zum Netzwerkanbieter standardisiert, muss die Anbindung der Endapplikation für jeden Netzwerkanbieter neu entwickelt werden. Auch hat man es oft mit sehr vielen IoT Geräten zu tun, welche verwaltet werden müssen.

Ziel der Arbeit ist es, eine Abstraktionsschicht zwischen Netzwerkanbietern und Endapplikation zu realisieren. Diese soll eine einheitliche Schnittstelle zu den verschiedenen Netzwerkanbieter bereitstellen, über welche man die üblichen Aktionen wie: Geräte erstellen, Geräte löschen, Geräte umregistrieren sowie auch Datenpakete senden kann. Damit die Plattform auch mit sehr vielen Geräten noch funktioniert, soll sie gut skalierbar sein.

Realisiert wurde eine Applikation, welche diese Ziele umsetzt. Zusätzlich ist es möglich, die Daten der IoT Geräte direkt zu parsen, bevor die Daten an die Client Applikation weitergegeben werden, sowie individuelle Alarme einzurichten, um die Geräte zu überwachen. Für den Benutzer bietet unsere Arbeit ein User Interface, mit welchem er die IoT Geräte verwalten kann.

Dank der Microservice-Architektur, kann die Applikation in Zukunft einfach erweitert werden. So können weitere Netzwerkanbieter hinzugefügt oder andere Protokolle zur Kommunikation mit der Client Applikation eingesetzt werden. Ausserdem wäre es denkbar, nicht nur LoRaWAN, sondern auch Mobilfunknetze mittels Narrowband-IoT zu unterstützen.

Part III

Management Summary

Problemstellung

Im Internet der Dinge (engl. Internet of Things), also die Vernetzung jeglicher Geräte über das Internet, gibt es in der Schweiz diverse unterschiedliche Anbieter von LoRaWAN-Netzen. LoRaWAN ist ein Netzprotokoll, welches eine Übertragung von Daten über grosse Reichweiten mit möglichst geringem Energieverbrauch bietet. Ein Datenpaket wird vom Gerät über eine Antenne zu einem Server eines Netzwerkanbieters geschickt. Dieser Weg ist im LoRaWAN-Standard spezifiziert. Von diesem Server gelangen die Daten dann zum Kunden und einer von ihm im Einsatz stehenden Applikation, welche die Gerätedaten empfangen will. Genau hier liegt allerdings das grosse Problem: Jeder Netzwerkanbieter hat eine eigene Spezifikation, wie die Kommunikationsschnittstellen vom Benutzer zu seinem Netzwerkserver umgesetzt werden.

Vorgehen

Zusammen mit dem Industriepartner Adnexo GmbH, wurden die Anforderungen an eine einheitliche Schnittstelle eruiert. Sie soll als eine Art Proxy zwischen den Netzwerkservern und den Client Applikationen fungieren, welche der Benutzer unabhängig vom jeweiligen Netzwerkanbieter benutzen kann. Die Schnittstelle beinhaltet das Management, sowie das Senden und Empfangen von Paketen der IoT-Geräte. Ausserdem soll es dem Benutzer möglich sein, die Plattform über ein einfaches User Interface zu bedienen.

Ein wichtiger Punkt bei der Evaluierung der Architektur kam der "Skalierbarkeit" zu Gute. Da mit mehreren 1000 Geräten gerechnet werden muss, ist die Stabilität der Architektur ein Kernstück der Arbeit. Mit Hilfe von Microservices, welche beliebig skaliert werden können, und Eureka, einer Service Registry entwickelt von Netflix, kann die Stabilität gewährleistet werden. Zusätzlich ist für die Persistenz der Daten eine NoSQL-Datenbank im Einsatz, welche bei Bedarf mit mehreren Instanzen gleichzeitig Daten verarbeiten kann. Die Microservices basieren auf den Spring Boot Framework, da es einfache REST API Strukturen anbietet und eine schnelle Einbindung von Eureka erlaubt.

Im Rahmen dieser Arbeit sind die Netzwerkanbieter Swisscom und Loriot berücksichtigt worden. Für jeden Netzwerkanbieter existiert ein eigenständiger Microservice, welcher ausschliesslich mit seiner Schnittstelle kommuniziert. Beide Anbieter besitzen REST-Schnittstellen.

Geräte können ausserdem mit Parser- und Alarmfunktionen versehen werden. Der Benutzer kann Pythoncode hinterlegen. Anhand diesem werden die Datenpakete geparst. Des Weiteren kann er auch Pythoncode für Alarme hinterlegen. Dieser wird dann nach dem parsen ausgeführt, und löst allenfalls einen Alarm aus.

Ergebnisse

Das Ergebnis beinhaltet ein REST-API, welche die angeforderten Funktionen umsetzt und mit welcher eine Client Applikation direkt kommunizieren kann. Es besteht die Möglichkeit, bei einem der beiden

Netzwerkanbietern Geräte zu registrieren bzw. zu deregistrieren, Downlinks an ein Gerät zu senden und Uplinks von Geräten zu empfangen. Ausserdem können einem Gerät Parser hinterlegt werden, um die hexadezimalen Daten des Uplinkpakets zu übersetzen.

Zusätzlich gibt es eine Frontend-Applikation, welche ebenfalls direkt mit der API kommuniziert. Auf ihr erhält der Benutzer eine Übersicht seiner Geräte und den jeweiligen Datenpaketen. Ausserdem bekommt er die Möglichkeit, Anpassungen am System vorzunehmen (Bsp. Anpassung der Konfiguration für den Netzwerkanbieter).

Beide Zugriffe, direkt auf die API oder auf das Frontend, werden mit Hilfe von Bearer Tokens durch einen unerlaubten Zugriff geschützt. Im Fall der API wird ein Token vorgeneriert, beim Frontendzugriff ist ein Login mit Accountname und Passwort notwendig, um die Plattform benutzen zu können.

Ausblick

Nachdem Swisscom und Loriot bereits als mögliche Netzwerkanbieter zur Verfügung stehen, kann die Plattform noch mit weiteren Anbietern, wie zum Beispiel dem in der Schweiz beliebten The Things Network ausgebaut werden. Ausserdem ist das Benutzermanagement der Plattform noch mit einer Rechtevergabe und entsprechenden Rollen erweiterbar. Man kann den Client Applikationen neben HTTP-Post und Influx-DB auch noch neue Output-Möglichkeiten zur Verfügung stellen, wie zum Beispiel Websockets.

Für die Zukunft wäre auch eine Unterstützung von unterschiedlichen Netzwerkprotokollen nebst dem LoRaWAN-Standard denkbar. Insbesondere Narrowband-IoT oder eine andere Technologie auf der Basis des neuen Mobilfunkstandards 5G wären hier interessant.

Part IV

Technischer Bericht

1 Einleitung

In der Grafik 1 sehen wir die klassische LoRaWAN Architektur. Innerhalb dieser Architektur betreibt ein Netwerkanbieter diverse Antennen (Gateways), welche Pakete der End Nodes (Sensoren) empfangen und an den Netzwerkserver des Netzweranbieters sendet. Der Netzwerkanbieter kümmert sich um die Infrastruktur und leistet Services wie z.B. Deduplikation der Pakete. Dieser Weg ist im LoRaWAN Standard standardisiert.

Danach muss das Paket an den Application Server weitergeleitet werden. Wie dies geschieht, ist leider nicht im LoRaWAN Standard festgehalten. Jeder Netzwerkanbieter kann dies unterschiedlich umsetzen.

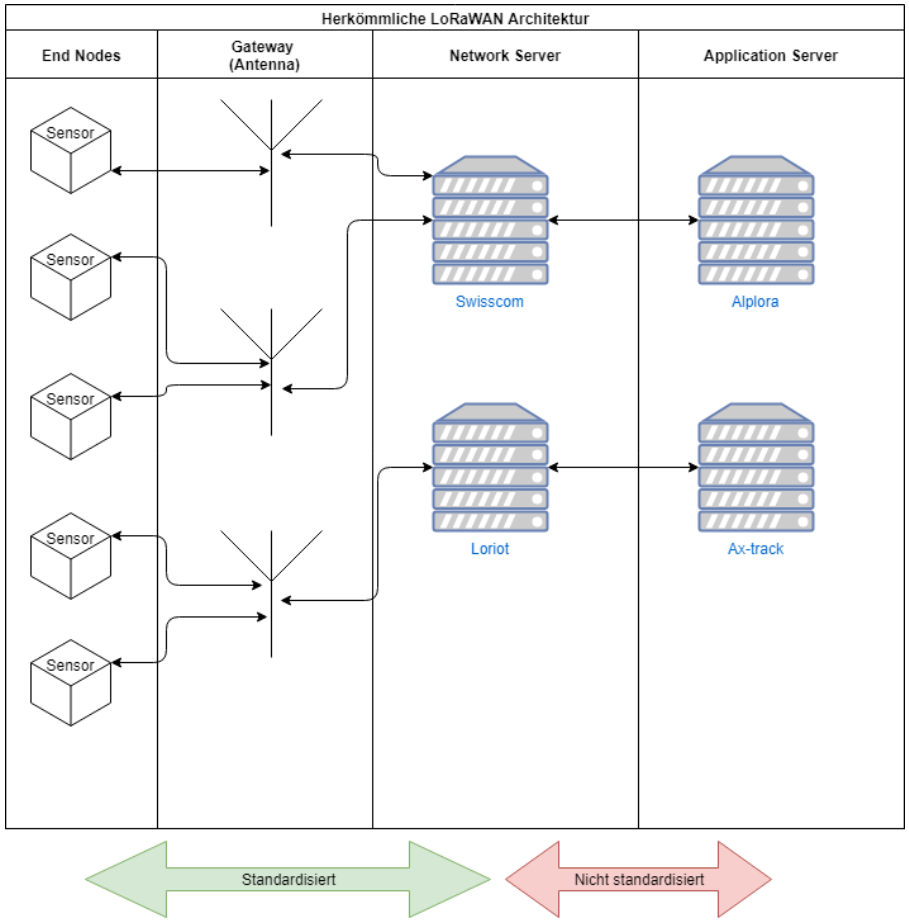


Abb. 1: Klassische LoRaWAN Architektur

Ein Teil dieser Arbeit befasst sich damit, diesen Missstand zu beheben. Dies wird erreicht, in dem eine neue Plattform zwischen den Netzwerk Server des Netzwerkanbieters und den Application Server des Kunden eingeführt wird. Diese Management Plattform bietet den Application Servern eine von uns standardisierte API, um mit den Netzwerkanbietern zu kommunizieren und die Pakete der End Nodes zu erhalten (Siehe Grafik 2).

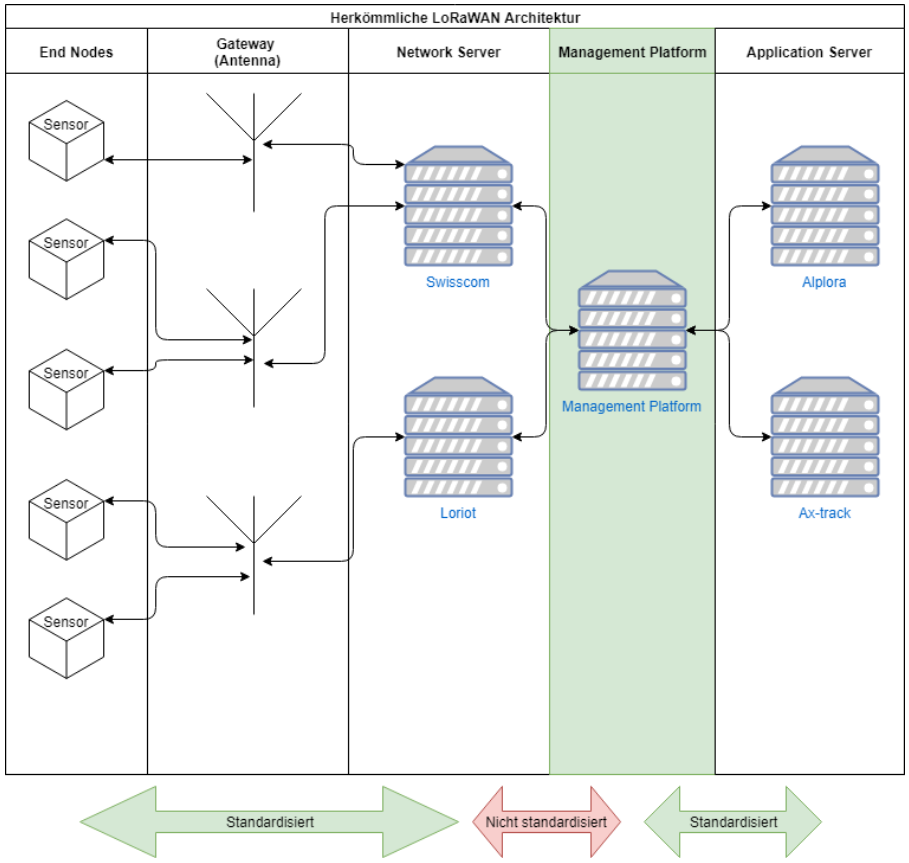


Abb. 2: LoRaWAN Architektur mit Management Plattform

Diese Architektur erlaubt es, weitere Aufgaben von den Application Servern auszulagern. Zum Beispiel soll die Plattform in der Lage sein, neue Sensoren beim Netzwerkanbieter zu registrieren. Natürlich muss dieses Interface jeweils für jeden Netzwerkbetreiber einzeln implementiert und den Application Servern eine standardisierte API zur Verfügung gestellt werden. Darüber soll es auch möglich sein, den Netzwerkbetreiber eines Sensors einfach zu wechseln.

Wenn ein End Node ein Datenpaket sendet, wird von einem Uplink gesprochen. Der Netzwerkbetreiber empfängt diesen und stellt sicher, dass er den Application Server erreicht. Allerdings kann auch der Application Server ein Paket an den Sensor senden. Zum Beispiel um diesem neue Konfigurationen zu senden. Ein solches Paket nennt man einen Downlink. Das Problem ist, dass der Netzwerkbetreiber nicht garantieren kann, dass ein solches Paket den End Node tatsächlich erreicht. Damit sich nicht jeder Application Server um diese Zustellung kümmern muss, soll die Management Plattform dieses Senden des Downlinks sicherstellen.

Da alle Pakete durch die Management Plattform gehen, ist dies auch der perfekte Ort um Analysen zu machen. Die Plattform soll erkennen, wenn Probleme mit einem Sensor auftreten. Dazu muss ein flexibles Regelsystem implementiert werden, mit welchem verschiedene Analysen der Pakete gemacht werden können.

Die Schwierigkeit besteht darin, eine Architektur zu finden, welche es ermöglicht, einfach neue Netzwerkanbieter zur Management Plattform hinzuzufügen. Es muss eine API definiert werden, welche sicher und möglichst einfach zu benutzen ist. Es ist nicht trivial, ein flexibles Regelsystem zu implementieren, mit welchem die Pakete effizient analysiert werden können. Zudem soll das ganze System möglichst

performant, sicher und skalierbar sein.

1.1 Systemgrenze

Die Management Plattform soll sich auf die Abstraktion zwischen dem Netzwerkanbieter und den Application Servern beschränken. Dazu soll möglichst viel Logik auf die Management Plattform ausgelagert werden können, damit sich die Application Server nicht um die Details der Kommunikation zum Netzwerkbetreiber kümmern müssen.

Die Arbeit beinhaltet des Weiteren ein Graphical User Interface (GUI), über welches die Geräte von einem technischen Mitarbeiter verwaltet werden können.

Dies alles muss so geschehen, dass die Application Server nicht im Funktionsumfang eingeschränkt werden.

2 Internet of Things (IoT)

In diesem Kapitel werden die wichtigsten Informationen für das Verständnis eines IoT-Netzwerks beschrieben und einzelne der bekanntesten Technologien im Umfeld näher gebracht. Es soll aufgezeigt werden, welche Technologie die Interessen der Arbeit am besten vertreten kann.

2.1 Einführung

Unter Internet of Things versteht man die vollkommene Vernetzung von Geräten jeglicher Art (Elektrogeräte, Fahrzeuge, etc.) an das Internet. Diese Geräte besitzen die Möglichkeit zur Kommunikation untereinander über grössere Distanzen, welche je nach Standardisierung variieren.

2.2 LPWAN[1]

LPWAN steht für Low Power Wide Area Network und bezeichnet ein Funknetz, welches einen grossen Entfernungsbereich abdecken kann. Ziel ist es, die grosse Reichweite mit möglichst geringen Energie- und Betriebskosten der Endgeräte zu erreichen.

Ein einzelnes Netzwerk besteht dabei aus Endgeräten, Gateways und Netzwerkservern.

Endgerät[4] Ein Endgerät beschreibt ein Gerät, welches an einen Anschluss des öffentlichen oder privaten Daten- bzw. Telekommunikationsnetzes angeschlossen ist. Der Verbindungsanschluss definiert sich über ein Kabel mit Steckverbinder oder auch über Funk.

In der Welt von IoT gibt es hierfür folgende Beispiele:

- Verkehrsmittel
- Wearables
- Diverse Sensoren

Gateway[5] Ein Gateway im IoT-Umfeld beschreibt eine Hardware-Komponente (Bsp. Antenne), welche zwei Systeme miteinander verbindet und die Daten entsprechend weiterleiten kann. Hierbei handelt es sich um die Verbindung zwischen dem Endgerät und dem Netzwerkserver.

Netzwerkserver Netzwerkserver können von mehreren unterschiedlichen Unternehmen oder Gruppierungen angeboten werden. Sie empfangen die Daten der Endgeräte und stellen sie den Kundenapplikationen zur Verfügung. In der Schweiz besitzen die Swisscom und TheThingsNetwork die grössten Abdeckungen.

2.3 Technologien

Für die Analysephase dieser Arbeit wurden in der Gruppe die folgenden drei Technologien, welche im IoT Umfeld verwendet werden, bestimmt. Jede einzelne Technologie besitzt bestimmte Eigenschaften und Vor- sowie Nachteile, welche einen Gebrauch stark einschränken können.

2.3.1 LoRaWAN[2] [3]

LoRaWAN (Low Power High Range WAN) ist eine Spezifikation der LoRa-Alliance, einer offenen, gemeinnützigen Vereinigung verschiedenster Unternehmungen.

Architektur Die Netzwerk-Architektur wird mit Hilfe einer Sterntopologie bereitgestellt, in welcher die einzelnen Gateways den Nachrichtenfluss zwischen den Endgeräten und einem zentralen Netzwerkserver weiterleiten. Die Gateways transformieren Radio-Frequenz-Pakete in IP-Pakete und umgekehrt und kommunizieren über die standardisierten IP-Verbindungen mit den Netzwerkservern.

Wichtig für die Realisierbarkeit einer solchen Topologie ist eine hohe Verfügbarkeit der Gateways. Sowie die Fähigkeit, Daten von einer grossen Anzahl von Endgeräten zu erhalten. Dies kann erreicht werden, in dem eine adaptive Datenrate verwendet wird, welche über einen mehrkanaligen Multi-Modem-Sendeempfänger im Gateway gesendet wird, um mehrere Kanäle verwenden zu können.

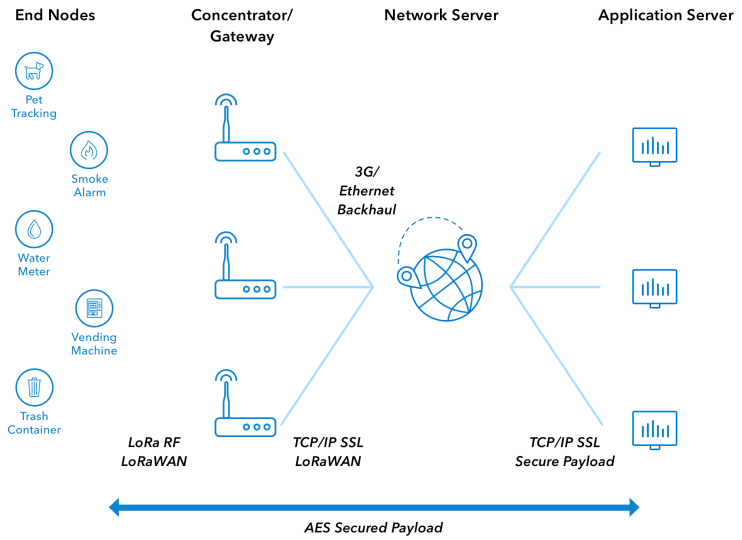


Abb. 3: LoRaWan Architektur

Die Endgeräte sind nicht direkt an einen Gateway angeschlossen. Die Geräte senden Daten ereignis- oder zeitplangesteuert und können von mehreren Gateways empfangen werden.

Kommunikation [6] Die Kommunikation auf den Frequenzbändern und die Anzahl verwendbarer Kanäle variiert je nach Region (siehe nachfolgende Tabelle).

	Europa	**Nordamerika**	**China**	**Japan/Korea**
Frequenzband	867 - 869 Mhz	902 - 928 Mhz	470 - 510MHz	920 - 925MHz
Kanalanzahl	10	64 + 8 + 8	*	*
Data Rate	250bps - 50kbps	980bps - 21.9kbps	*	*

Tab. 1: LoRaWAN Kommunikationsdaten

**: Wird von den technischen Komitees der jeweiligen Regionen noch bestimmt*

Die möglichen Datenraten in einem LoRaWAN-Netzwerk liegen zwischen 0.3 bis 50 kBit/s, wobei pro Paket maximal 59 Byte an Nutzdaten enthalten sein können.

Die Reichweite der Kommunikation durch einen einzelnen Gateway beträgt somit bis zu 13km bei idealen Bedingungen und bis zu 3km in innerstädtischen Gegenden.

Gateways können über einen Kanal zur gleichen Zeit mehrere unterschiedliche Datenraten entgegen nehmen. Wenn sich ein Endgerät nah an der Antenne befindet und das Link-Budget (Funksignalstärke) hoch ist, kann eine hohe Datenrate verwenden werden. Je tiefer ein Link-Budget ist, desto niedriger ist die Datenrate. Als netten Nebeneffekt kann mit Hilfe von adaptiven Datenraten auch die Batterielaufzeit optimiert werden.

Sicherheit LoRaWAN verwendet Sicherheitsaspekte auf zwei verschiedenen Layern: Auf der Netzwerk- sowie auch auf der Applikationsschicht. Netzwerksicherheit wird mit Hilfe eines eindeutigen 128-bit Netzwerksitzungsschlüssels garantiert, welcher vom Endgerät und Netzwerkserver gemeinsam verwendet wird. Ebenfalls wird ein 128-bit Applikationsschlüssel, auch AppSKey genannt, auf der Anwendungsebene verwendet. Mit Hilfe von AES-Algorithmen wird die Authentifizierung und Integrität von Paketen für den Netzwerkserver und die End-zu-End-Verschlüsselung für den Application Server bereitgestellt.

2.3.2 ZigBee

Die ZigBee-Spezifikation wurde von der 2002 gegründeten ZigBee-Allianz entwickelt. Sie setzt den Fokus auf kurzreichweitige Netzwerke bis 100 Meter, wie zum Beispiel Hausautomation, Sensornetzwerke oder Lichttechnik.

Im Verlauf der Jahre wurden diverse Standards eingeführt und weiterentwickelt. Die aktuellste Technologie ist ZigBee 3.0, welche eine komplette IoT-Lösung zur Verfügung stellt. Diese reicht von vermaschter Topologie bis zu einer universellen Sprache für Endgeräte-Kommunikation.

Architektur [7] Der ZigBee-Standard kennt grundsätzlich drei verschiedene Topologietypen: Baum, Stern und Mesh. Da sich die Mesh-Topologie im ZigBee 3.0 Umfeld für IoT durchgesetzt hat, wird sich dieser Abschnitt nur mit der Mesh-Topologie befassen.

In einer kabellos vermaschten Struktur, auch Peer-to-Peer genannt, können Geräte miteineinander verbunden werden. Dabei werden verschiedene Typen von Geräten eingesetzt:

- Koordinator: Wird nominiert (weil er das erste Gerät ist oder bei Ausfall des bisherigen Koordinators gewählt wurde) und bestimmt die Netzwerkkonfigurationen
- Router: Kommunizieren mit mehr als einem Gerät und stellen so das Netzwerk-Backbone
- Endgerät: Kommunizieren nur mit einem Gerät

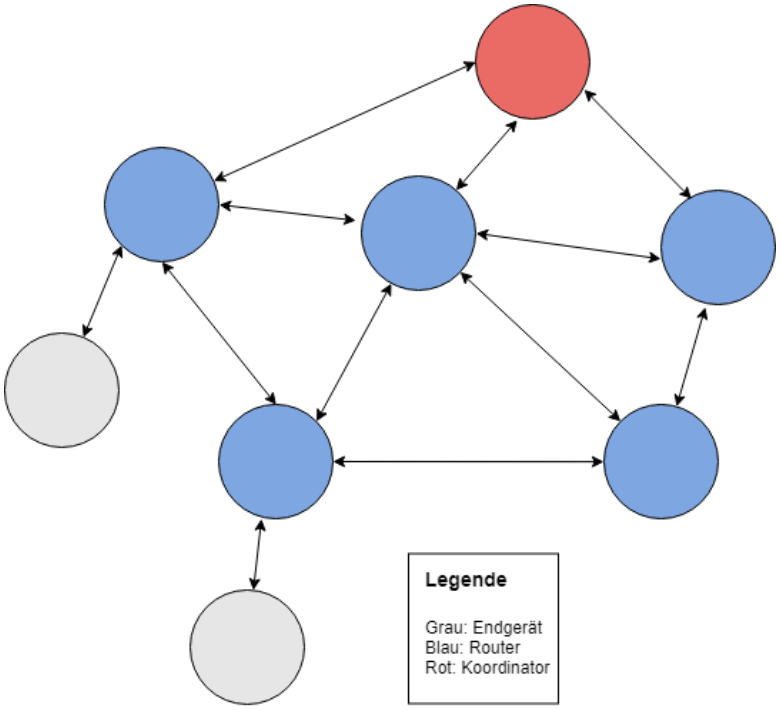


Abb. 4: ZigBee Mesh-Architektur

Die Kommunikationslogik in der vermaschten Topologie ist dezentralisiert, womit jedes Gerät mit einem Gerät in seinem Radius kommunizieren kann. Somit wird jedes Gerät auch zu einem Gateway.

Vorteil einer solchen Lösung ist die Robustheit des Netzes. Bei einem Ausfall eines Gerätes wird das Netzwerk nicht gross beeinflusst. Die Endgeräte können über ein anderes Gerät weiterkommunizieren. Dies macht das Netzwerk auch sehr flexibel, da Endgeräte leicht in das Netz integriert werden können.

Kommunikation[8] ZigBee basiert auf dem Kommunikationsstandard IEEE 802.15.4 (Wireless Personal Area Network (WPAN)) und operiert auf folgenden unterschiedlichen Frequenzbändern:

	Europa	**Nordamerika**	**Global**
Frequenzband	868 Mhz	915-921 Mhz	2.4 Ghz
Kanalanzahl	63	27	16
Data Rate	<100 kbps	<10 kbps	<250 kbps

Tab. 2: ZigBee Kommunikationsdaten

Damit auf dem 2.4GHz keine Interferenzen mit anderen Geräten im Wi-Fi Netz entstehen, erhalten ZigBee Produkte einen Zugriff auf 16 separate, 5MHz grosse Kanäle im 2.4GHz Band. Ausserdem verfügt ZigBee über ein nach IEEE 802.15.4 definiertes CSMA-CA-Protokoll, welches die Wahrscheinlichkeit einer Kollision mit anderen Benutzern reduziert. ZigBee verwendet eine automatische Weitermeldung der Daten, um die Robustheit des Netzes zu gewährleisten. Auch die kleine Grösse der einzelnen Pakete in einem ZigBee-Netz vermindert die Fehleranfälligkeit.

Sicherheit[9] Auf Applikations-Ebene wird zwischen zwei Geräten mit Hilfe eines eindeutigen Sets von AES-128-Schlüsseln eine Verbindung hergestellt.

Wenn ein Gerät das Netzwerk neu betreten will, benötigt es einen Install Code, welcher er vom Trust Center in Form eines Strings oder QR-Codes erhält. Der Install Code ist eine zufällige 128-Bit Nummer, geschützt durch einen 16-bit CRC. Danach vereinbaren sie einen 128-bit Trust Center Link Key, welcher vom Install Code mit Hilfe der Matyas-Meyer-Oseas (MMO) Hash-Funktion abgeleitet wird.

Ausserdem erstellt und verteilt das Trust Center periodisch neue Netzwerk-Schlüssel, welche mit dem Trust Center Link Key verschlüsselt an das Gerät gesendet werden.

2.3.3 Narrowband-IoT (NB-IoT)

Narrowband-IoT, eine Spezifizierung des 3rd Generation Partnership Project (3GPP), wird mit Hilfe von bestehenden und noch in Entwicklung stehenden Mobilfunktechnologien wie zum Beispiel GSM, LTE und in Zukunft auch 5G verwendet.

Die Technologie besitzt eine hohe Durchdringungskapazität und eignet sich daher sehr gut, um mit Geräten an empfangstechnisch ungünstigen Standorten zu kommunizieren (Bsp. Abgelegene Gegenden, Unter der Erde, etc.).

Architektur[10] Die Architektur eines Narrowband-IoT-Netzes benötigt folgende Komponenten:

- Antenne: Im NB-IoT Umfeld werden die Antennen von den Mobilfunkanbietern verwendet.
- NB-IoT Modem/Modul: Der Chipsatz des Moduls sollte schon im Endgerät integriert worden sein.

- Host: Auf dem Host wird die Applikation ausgeführt. Sie dient als Schnittstelle zu den Sensoren und steuert das Modul.
- Cellular Network: Über das Mobilfunknetz werden die niedrigen Datenmengen zwischen den Antennen gesendet.

Kommunikation Da Narrowband IoT über das zellulare Mobilfunknetz und deren lizenzierten Frequenzen eingesetzt wird, gibt es eine hohe Anzahl an verfügbaren Frequenzen. In der Schweiz wird auf der Frequenz 800 Mhz gearbeitet, auf welcher man eine Datenrate zwischen 0.4 - 60 kBit erreichen kann.

Ausserdem besteht durch die Mobilfunkantennen bereits eine hohe Abdeckung innerhalb unserer Landesgrenzen. Outdoor beträgt die Abdeckung der Swisscom 98.8%, Indoor 96.6%.

Sicherheit Narrowband-IoT verwendet die von LTE schon vordefinierten Sicherheitsmechanismen, basierend auf dem EPS (Evolved Packet System).

2.4 Evaluation

Folgende Punkte fliessen in die Evaluation der brauchbarsten Technologie für die Plattform ein:

Kommunikationsradius Die Plattform soll über grosse Distanzen erreichbar sein und damit einen grossen Kommunikationsradius abdecken können.

Batterielaufzeit Die Kommunikation im Netz soll die Batterielaufzeit nicht erheblich beeinträchtigen.

Netwerksicherheit Innerhalb eines Netzes müssen die Pakete verschlüsselt übertragen werden können.

Erfahrungswerte Die Technologie und deren Funktionsweise ist bekannt und kann schnell eingesetzt werden.

2.4.1 Entscheid

Die Batterielaufzeit bewegt sich bei allen Technologien in einem akzeptablen Rahmen.

Zigbee wirkt als eine sehr elegante Lösung, eignet sich aufgrund der zu kurzen Distanzen allerdings nicht.

Narrowband-IoT ist in der Schweiz noch sehr neu und nicht vollständig eingesetzt. Informationen zum Einsatz der Technologie sind sehr rar. Es würde sich als interessante Technologie empfehlen, allerdings

wird LoRaWAN vom Industriepartner schon verwendet und deshalb für den Rahmen dieser Arbeit bevorzugt.

3 Problemanalyse

3.1 Domain Analyse

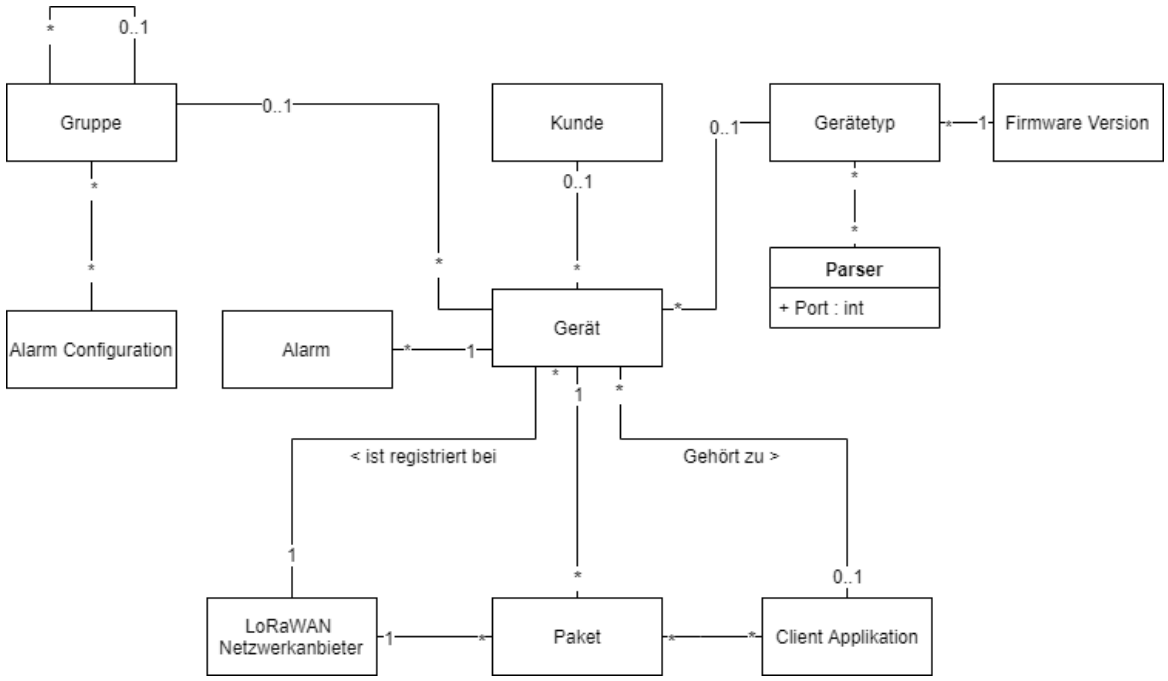


Abb. 5: Domain Model

Das Domain Model wurde aus Sicht der zu entwickelnden Applikation gezeichnet, und beinhaltet deshalb bewusst nicht die long range wide-area network (LoRaWAN) Komponenten wie z.B. den Gateway.

3.2 Systemspezifikation

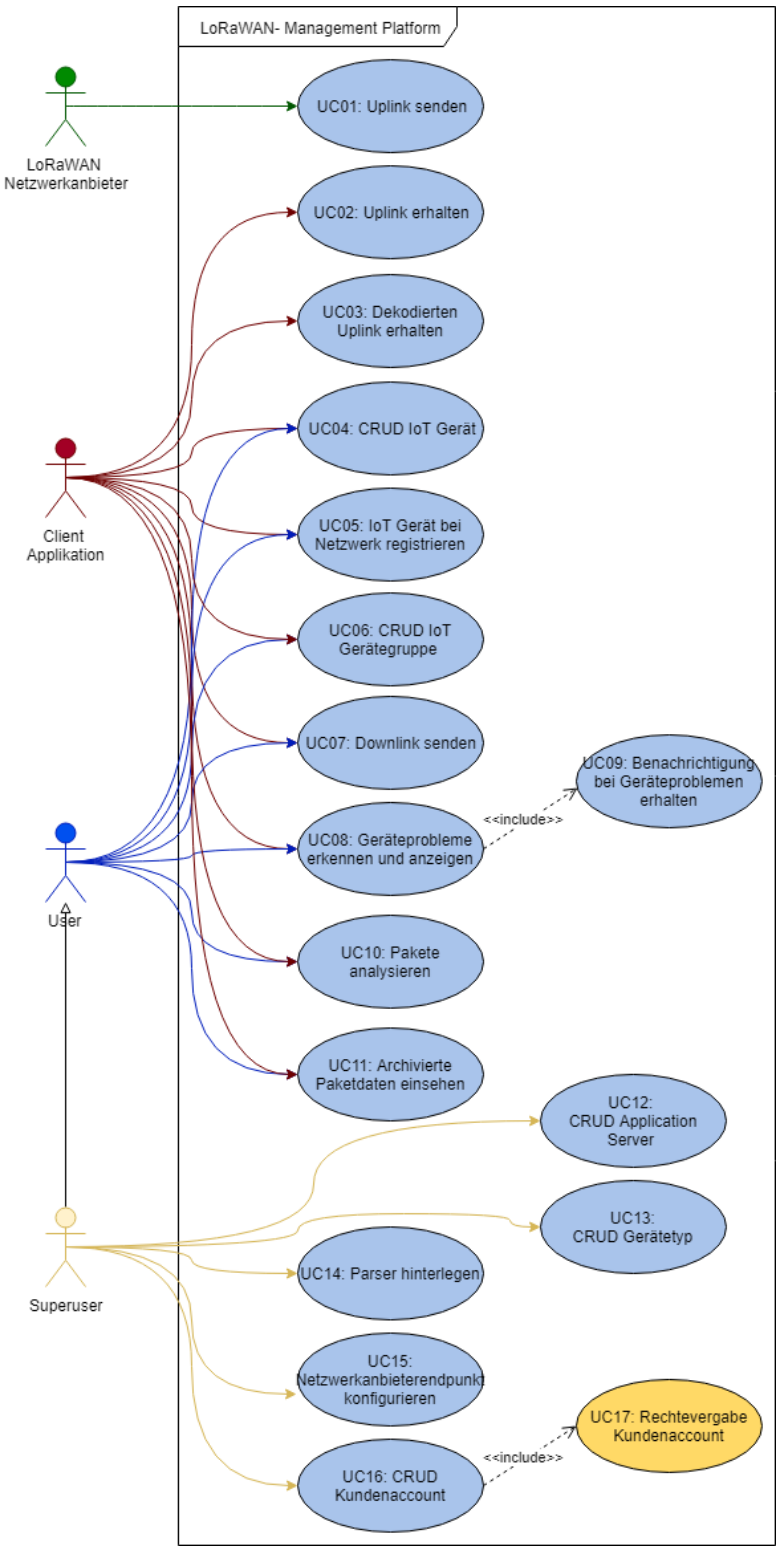


Abb. 6: Use Cases

Priorisierung Die Use-Cases werden mit einer Priorisierung entsprechend ihrer Wichtigkeit in der Umsetzung versehen. Folgende Werte können sie annehmen:

- **Blau**: Priorität 1, wird in jedem Fall umgesetzt
- **Gelb**: Priorität 2, optionales Feature welches bei freien Zeitressourcen noch umgesetzt wird.

3.2.1 Die Akteure

LoRaWAN Netzwerkanbieter Der LoRaWAN Netzwerkanbieter stellt das Netzwerk, über welche die IoT Geräte kommunizieren, bereit. Er empfängt die Uplinks auf seinem Netzwerkserver, und leitet diese entsprechend weiter. Will man dem Gerät einen Downlink senden, so muss man dies über den LoRaWAN Netzwerkanbieter und dessen Netzwerkserver machen.

Client Applikation Dies ist die Applikation, welche an den Paketen der IoT Geräten interessiert ist, und welche das Senden vom Downlinks auslösen kann. Client Applikationen können von den unterschiedlichsten Unternehmen bereitgestellt werden. Client Applikationen können auch als Application Server benannt werden.

User Der User ist eine Person welche die persönlichen IoT Geräte verwalten und überwachen möchte.

Superuser Ist ein User mit zusätzlichen Berechtigungen.

3.2.2 Funktionale Anforderungen

In den nachfolgenden Paragraphen werden die einzelnen Use Cases im Brief-Format beschrieben.

UC01: Uplink senden

Der LoRaWAN Netzwerkanbieter erhält ein Paket von einem IoT Gerät. Dieses Paket möchte er nun über die Plattform an die Applikation weiterleiten können.

UC02: Uplink erhalten

Die Client Applikation erhält den Uplink, welcher zuvor vom LoRaWAN Netzwerkanbieter an die Applikation gesendet wurde. Dabei soll es möglich sein, dass die Client Applikation mehrere URLs angiebt, an die der Uplink gesendet werden soll. Für die Client Applikationen soll es keine Rolle spielen, über welchen Netzwerkanbieter die Pakete gesendet werden.

Ein Paket eines unbekannten Geräts soll in das System aufgenommen und gekennzeichnet werden.

UC03: Dekodierten Uplink erhalten

Die Client Applikation erhält den dekodierten Uplink, welcher zuvor vom LoRaWAN Netzwerkanbieter an die Applikation gesendet wurde.

UC04: CRUD IoT Gerät

Ein Benutzer kann ein IoT Gerät erstellen, anschauen, anpassen oder löschen. Beim Erstellen des Gerätes muss zusätzlich ein Gerätetyp hinterlegt werden.

UC05: IoT Gerät bei Netzwerk registrieren

Ein Benutzer kann ein IoT Gerät, welches er bereits erstellt hat, nun bei einem LoRaWAN Netzwerkanbieter registrieren. Dabei kann er es auch von einem bestehenden Anbieter zu einem anderen verschieben.

UC06: CRUD IoT Gerätegruppe

Ein Benutzer kann eine Gerätegruppe erstellen, anschauen, anpassen oder löschen. Zudem kann er die einzelnen Geräte einer Gerätegruppe zuordnen.

UC07: Downlink senden

Die Client Applikation kann ein Downlink Paket an das IoT Gerät senden. Dazu muss er die Nachricht mitgeben und definieren, ob er die Nachricht "confirmed" senden will oder nicht. Confirmed bedeutet, das der Netzwerkanbieter ein "confirm" vom Gerät erhält.

UC08: Geräteprobleme erkennen und anzeigen

Ein Benutzer kann alle Geräte anzeigen lassen, welche Probleme haben. Probleme werden im Kapitel 3.2.4 näher beschrieben. Ausserdem kann ein Benutzer einem Gerät oder einer Gerätegruppe eine selbstdefinierte Alterting-Regel hinterlegen.

UC09: Benachrichtigung bei Geräteproblemen erhalten

Ein Benutzer kann bei Problemen eines Geräts (beschrieben in UC7: Geräteprobleme erkennen und anzeigen) benachrichtigt werden. Die Benachrichtigung erfolgt über eine vom Benutzer hinterlegte E-Mail-Adresse oder die Plattform kann die Client Applikation benachrichtigen, damit diese weitere Schritte implementieren kann.

UC10: Pakete analysieren

Ein Benutzer kann die einzelnen Paketinformationen inkl. geparste Payload-Daten pro Gerät einsehen.

UC11: Archivierte Paketdaten einsehen

Ein Benutzer kann archivierte Paketedaten einsehen und mit Hilfe diverser Filterfunktionen suchen.

UC12: CRUD Application Server

Ein Superuser kann einen Application Server erstellen, anzeigen, anpassen oder löschen. Dabei muss er die gewünschte Kommunikationsart bestimmen, über welche die Uplinks an den Application Server gesendet werden. Es können auch mehrere URLs pro Application Server hinterlegt werden, z.B. ein Backup Server, falls der Primäre ausfallen sollte.

UC13: CRUD Gerätetyp

Ein Superuser kann einen IoT-Gerätetyp erstellen, anzeigen, anpassen oder löschen. Jeder Gerätetyp besitzt eine Firmwareversion. Die Firmwareversion kann separat verwaltet werden.

UC14: Parser hinterlegen

Ein Superuser kann einer Firmwareversion einen Parser hinterlegen. Dieser Parser soll den Uplink des IoT Gerätes decodieren können.

UC15: Netzwerkanbieterendpunkt konfigurieren

Ein Superuser kann eine implementierte Netzwerkanbieter-Konfiguration bei der Management Plattform konfigurieren, oder eine neue Konfiguration hinzufügen.

UC16: CRUD Kundenaccount

Ein Superuser kann Kundenaccounts erstellen, anzeigen, anpassen oder löschen.

UC17: Rechtevergabe Kundenaccount

Ein Superuser kann einem Kunden diverse Zugriffsrechte vergeben.

3.2.3 Nichfunktionale Anforderungen

Skalierbarkeit

Es soll möglich sein, 10'000 Geräte auf der Applikation zu verwalten. Auch muss sichergestellt werden, dass 500 Down- und 500 Uplinks pro Minute verarbeitet werden können.

Erweiterbarkeit

Die Applikation muss um weitere LoRaWAN Netzwerkanbieter erweitert werden können.

Sicherheit

Alle Verbindungen müssen verschlüsselt werden. Auch dürfen Benutzer nur authentifiziert und autorisiert auf die Applikation zugreifen.

Eine Client Applikation muss sich versichern können, dass der Absender der Daten auch wirklich die Management Plattform darstellt.

Falls die LoRaWAN Netzwerkanbieter diese Möglichkeit anbieten, müssen die Verbindungen zu diesen verschlüsselt geschehen.

3.2.4 Geräteprobleme / Alerting

Die Applikation soll die Uplink-Pakete auswerten und einen Benutzer über das User Interface benachrichtigen, wenn eine der folgenden Bedingungen eintritt. Diese Benachrichtigungen können pro Gerätegruppe ein oder ausgeschaltet werden.

Schwacher Empfang des Geräts Fällt der Spreading Factor (SF) über einen vorkonfigurierten Wert, wird diese Benachrichtigung ausgelöst.

Unter- oder Überdurchschnittlicher Versand von Nachrichten Wenn diese Benachrichtigung aktiviert wird, wird der durchschnittliche Versand von Paketen pro Tag der letzten zwei Wochen errechnet. Weicht der aktuelle Versand pro Tag stark davon ab, wird diese Benachrichtigung ausgelöst.

Definierte Anzahl Pakete wird unter- bzw. überschritten Oft weiss der Betreiber eines Sensors, wie oft dieser Pakete senden sollte. Deshalb kann man bei dieser Benachrichtigung ein fixes Minimum und Maximum pro Tag setzen. Wird dieses unter- oder überschritten, wird die Benachrichtigung ausgelöst.

Received signal strength indication (RSSI) oder Signal Noise Ratio (SNR) verschlechtern sich Bei dieser Benachrichtigung soll erkannt werden, ob sich der Empfang verschlechtert. Zum Beispiel wenn sich ein Gerät weiter weg von einer Antenne bewegt.

3.2.5 Netzwerkanbieter

Die Management Plattform unterstützt die zwei in der Schweiz weitverbreitesten LoRaWAN-Netzwerkanbieter:

- Swisscom
- Loriot

Es ist möglich, weitere Anbieter bei Bedarf einzubinden.

3.2.6 Output an Application Server

Die Management Plattform unterstützt folgende zwei Application Output-Möglichkeiten:

- HTTP-Post
- InfluxDB

Es ist möglich, weitere Anbieter bei Bedarf einzubinden.

4 Evaluation

4.1 Technologien Evaluation

4.1.1 Datenbank

Bei der Evaluation der Datenbank liegt der Hauptaugenmerk auf der Geschwindigkeit von Lese- und Schreiboperationen. Da die Management Plattform bei jedem Empfang eines Paketes mindestens eine Schreiboperation ausführen muss, muss besonders auf die Performance von Schreiboperationen geachtet werden.

RDBMS

Die meisten relationalen Datenbanken bieten vertikale Skalierung und eine eingebaute Replikation der Daten an, allerdings kann mit Hilfe des Load Balancings nur das Lesen auf vereinzelte Nodes verteilt werden.

Bei PostgreSQL gibt es zum Beispiel das Master-Slave Server Pattern, bei der ein Master-Server Read/Write Zugriffe erlaubt und diverse zusätzliche Slave-Server sich nur um Read-Zugriffe kümmern können. Eine synchronisierte Multimaster Replikation wird nicht unterstützt. [11]

NoSQL

NoSQL besitzt den Vorteil, eine horizontale Skalierung bereitzustellen. Es ist möglich, die Kapazität auf verschiedene Datenbank-Server zu verteilen, welche die Daten bei Bedarf neu verteilt. Desweiteren kann jeder dieser Server Lesen sowie Schreiben.

Als NoSQL-Datenbank Typ eignet sich eine dokumentenorientierte Datenbank.

Evaluation

Mit Hilfe einer verteilten relationalen Datenbank können die an die Management Plattform gestellten Anforderungen nicht erreicht werden. Bei einer hohen Last an Schreiboperationen würde das System langsam und nicht performant funktionieren, da der Master-Server alle Schreib-Operationen selber verarbeiten muss.

NoSQL resp. MongoDB löst dieses Problem von Grund auf und skaliert bei hohen Lasten sehr gut, da jede Datenbankinstanz auch Schreiboperationen durchführen kann. Die Relationen zwischen den einzelnen Objekten können mit Hilfe der dokumentenstrukturierten Architektur ebenfalls abgebildet werden. Ausserdem können Queries mit Hilfe von Indices auf einzelnen Dokumenteigenschaften noch beschleunigt werden.

Somit wird für die Persistenz der Daten MongoDB verwendet.

4.1.2 Backend

Beim Backend haben wir die Wahl von diversen Programmiersprachen. Da wir eine Sprache verwenden wollen in der wir beide Erfahrung haben, entscheiden wir uns für Java. Wir haben bereits die Studien Arbeit erfolgreich damit umgesetzt, und hoffen dass wir einiges davon übernehmen können (z.B. continuous integration (CI)).

Als Framework eignet sich Spring Boot, mit welchem wir auch schon Erfahrung haben.

Da wir eine Application Programming Interface (API) entwickeln müssen, die von externen Applikationen angebunden wird, macht es Sinn, diese möglichst gut zu dokumentieren. Dazu eignet sich ein Framework wie Swagger, mit dem man die API designen und dokumentieren kann. Auch kann sich aus der Definition der API direkt Code generieren lassen, welchen wir als Ausgangslage für die Implementation verwenden können. Ein weiterer Vorteil von Swagger ist, dass man die API mit einem User Interface (UI) erkunden kann, was die Verwendung deutlich vereinfachen kann.

4.1.3 Frontend

Dieses Projekt wird neben dem Backend auch ein GUI beinhalten. Es macht Sinn, dieses vom Backend zu entkoppeln. Dies hat den Vorteil, dass man bei Änderungen am Backend oder Frontend nicht das jeweils andere mitdeployen muss. Auch ist man durch diese Entkopplung vorbereitet, falls man andere GUIs bereitstellen möchte (z.B. eine Android App).

Das Frontend soll ein Webseite sein. Dies ist bei der Adnexo GmbH so üblich und gewünscht. Auch ermöglicht dies ein einfaches, zentrales updaten des Frontends, falls nötig.

Es gibt diverse Java Script Libraries die verwendet werden können. Bei der Adnexo GmbH wird Vue.js eingesetzt. Des Weiteren haben wir beide schon Vorkenntnisse damit. Deshalb haben wir uns dazu entschieden, im Frontend Vue.js einzusetzen.

4.2 Architektur

Die Applikation teilt sich in ein Frontend und ein Backend auf. Die Daten werden in einer NoSQL-Datenbank verwaltet.

4.2.1 Frontend

Da im Frontend das Framework Vue.js eingesetzt wird, nutzen wir einen komponentenbasierten Ansatz. Das heisst, dass das Frontend in einzelne Komponente aufgeteilt wird. Diese sind lose gekoppelt, was eine Wiederverwendung sehr leicht macht.

4.2.2 Backend

Monolithisch Die klassische Variante wäre, eine grosse, monolithische Backend Applikation zu programmieren, welche die gesamte (Backend-)Logik dieser Arbeit enthält. Natürlich würde man die

Logik in Schichten aufteilen, welche möglichst voneinander entkoppelt wären.

Der Vorteil ist, dass man Komponenten und Libraries zwischen den Komponenten teilen kann.

Der Nachteil ist allerdings, dass die Applikation schlecht skaliert. Es muss die gesamte Applikation als ganzes skaliert werden. Es werden also auch Komponenten skaliert welche gar nicht ausgelastet sind.

Microservices Man kann die Applikation als eine Sammlung von Microservices entwickeln. Dann kann man sich auch entscheiden, diese Services als eigenständige Container zu deployen.

Der Vorteil ist, dass man dadurch nur jene Container skalieren kann, welche auch tatsächlich höher ausgelastet sind. Auch ist die Weiterentwicklung einfacher und fehlerunanfälliger, da nicht eine ganze monolithische Applikation angepasst und neu deployed werden muss, sondern nur ein kleiner, eigenständiger Teil.

Ein Nachteil ist die zusätzliche Komplexität. Wird ein Service als eigener Container deployed, muss immer eine Kommunikation zwischen den Services stattfinden. Unter Umständen über das Netzwerk, was fehleranfällig ist, und mit zusätzlichem Aufwand verbunden.

Message Broker oder Service Regitry Wenn man den Microservice Ansatz wählt, müssen die einzelnen Services miteinander kommunizieren. Dazu haben wir uns zwei Ansätze angeschaut.

Der erste ist, einenen Message Broker einzusetzen. Bei diesem Ansatz abonieren die einzelnen Services ein "Topic", und reagieren, wenn darin neue Nachrichten publiziert werden. Der Service muss also nur den Message Broker kennen. Ein Vorteil davon ist auch, dass Nachrichten asynchron abgearbeitet werden können.

Der zweite Ansatz wäre eine Service Registry, wie zum Beispiel Eureka von Netflix, zu verwenden. Bei diesem registrieren sich die einzelnen Instanzen der Services. Und wenn diese eine Nachricht an einen anderen Service senden wollen, können sie bei der Service Registry den Standort (Hostname, Port, ...) erhalten. Der Vorteil ist, dass der Service nur die Service Registry kennen muss. Auch kann eine solche Service Registry als Load Balander dienen, und die Clients können die Verbindungsinformationen zwischenspeichern, was die Last auf die Service Registry verringert.

Entscheid Um die Anforderung der Skalierbarkeit zu befriedigen, werden wir den Microservices Ansatz wählen. Dazu nutzen wir eine Service Registry. Wir bevorzugen diesen Ansatz, weil die meisten Aufrufe Synchron geschehen werden. Und da beim Service Registry Ansatz die Nachrichten direkt von Service zu Service gesendet werden, und nicht noch einen Umweg über einen Broker machen müssen, muss die Service Registry bei hoher Last nicht gleich stark skaliert werden, wie wenn man einen Message Broker einsetzen würde.

4.3 Benutzercodeausführung

Uplink Pakete können geparst und mit Alarmen versehen werden. Für beide Fälle kann der Benutzer der Platform Code hinterlegen. Dies ist offensichtlich ein Sicherheitsrisiko, wobei man grundsätzlich zwischen Performance und Sicherheit abwägen muss. Folgende Varianten sind umsetzbar:

Keine Sicherheit Maximale Performance erreicht man, wenn man dem Usercode vertraut und keine Sicherheit einbaut. Dadurch könnte man den Code direkt im Hauptprozess auf der selben Maschine ausführen. Der Nachteil ist natürlich, dass der Benutzer grossen Schaden anrichten kann. Entweder willentlich, oder durch fehlerhaften Code.

Chroot Mittels Chroot könnte man das root-Verzeichnis eines Prozesses ändern und ihn so daran hindern andere Programme auszuführen oder Dateien zu manipulieren. Diese Methode hätte geringe Performanceeinbussen, allerdings teilen sich die Prozesse immer noch die selben Hardwareressourcen und den selben Kernel. Der unsichere Benutzercode könnte also immer noch das System zum Stoppen bringen, zum Beispiel mit einer Fork Bomb.

AppArmor Mittels AppArmor kann man die Fähigkeiten eines Programmes einschränken. So kann man zum Beispiel den Netzwerkzugriff verbieten, und das Schreiben und Lesen von Dateien. Die Performanceeinbussen sind sehr gering, allerdings kann man nicht verhindern, dass der Prozess zu viele Hardwareressourcen in Anspruch nimmt.

cgroups Mittels cgroups kann man die Ressourcen eines Prozesses einschränken.

Docker Eine weitere Möglichkeit ist Docker. Docker nutzt chroot, AppArmor und cgroups um Container vom Rest des Betriebsystems abzuschotten. Der Vorteil von Docker ist, dass es einfach zu konfigurieren ist, und bereits sehr gut getestet ist. Natürlich teilen sich Docker Container und Host immer noch den selben Kernel, allerdings ist dieser ziemlich gut geschützt. Das Starten eines Docker Container und die Ausführung eines simplen Hello-World-Python-Scripts dauerte in unserem Test ziemlich genau eine Sekunde. Der direkte Aufruf des Hello-World-Python-Scripts 0.035 Sekunden.

Der Rückgabewert an das System ist der geparste String oder eine jeweilige Fehlermeldung, falls fehlerhafter oder nicht ausführbarer Code angegeben wurde.

Virtualisation Wenn man den Kernel trennen möchte, muss man Virtualization einsetzen. Dies ist die sicherste Variante, aber braucht viel mehr Ressourcen auf dem Server. Auch ist es sehr langsam für jedes Ausführen des Codes eine neue virtualisierte Umgebung zu starten.

Entscheid Wir haben uns entschieden den Parsing Code in einem Docker Container auszuführen. Das ist ein guter Kompromiss zwischen Sicherheit und Performance.

Wir werden ein bereits bestehendes Projekt benutzen, welches Docker Container nutzt, um Python Code auszuführen, und dieses unseren Bedürfnissen anpassen: `https://github.com/christophetd/docker-python-sandbox`

5 UI Mockups

Im folgenden Kapitel werden erste GUI Überlegungen in Form von Mockups visualisiert.

5.1 Login

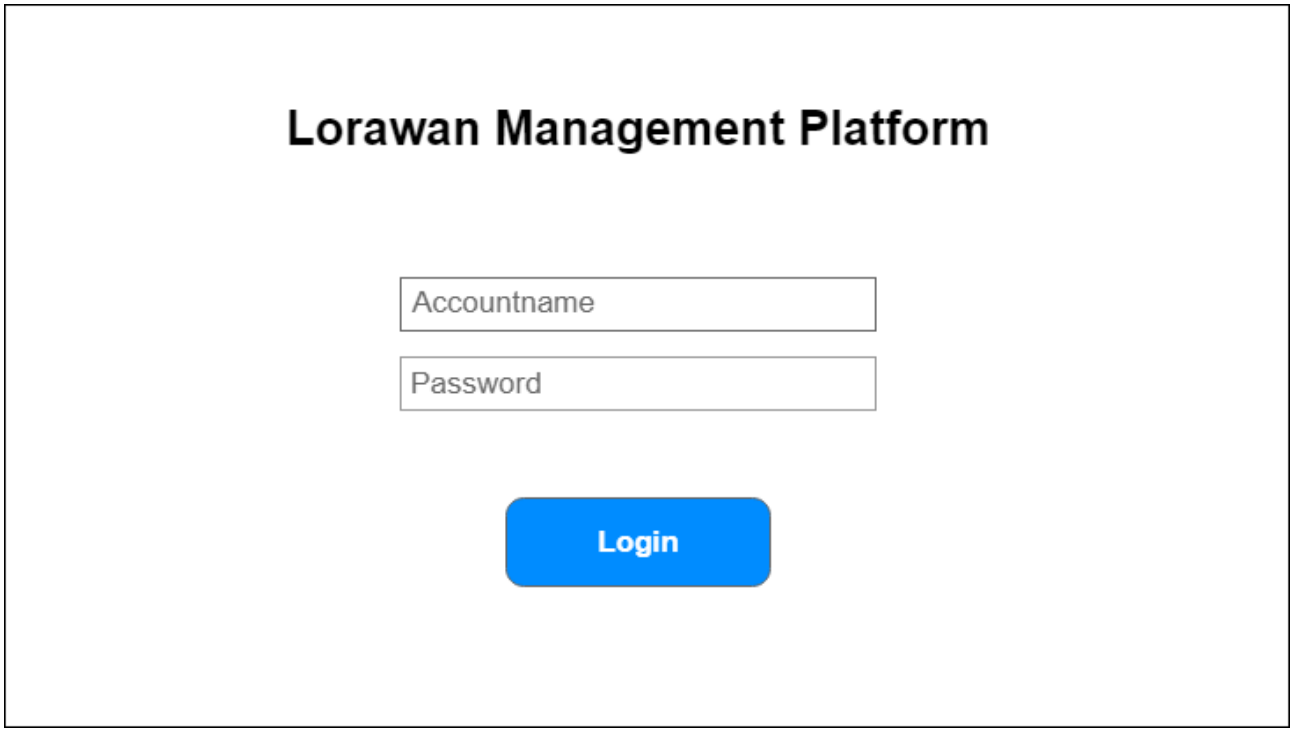


Abb. 7: Mockup Login

5.2 Accountmanagement

Accountübersicht

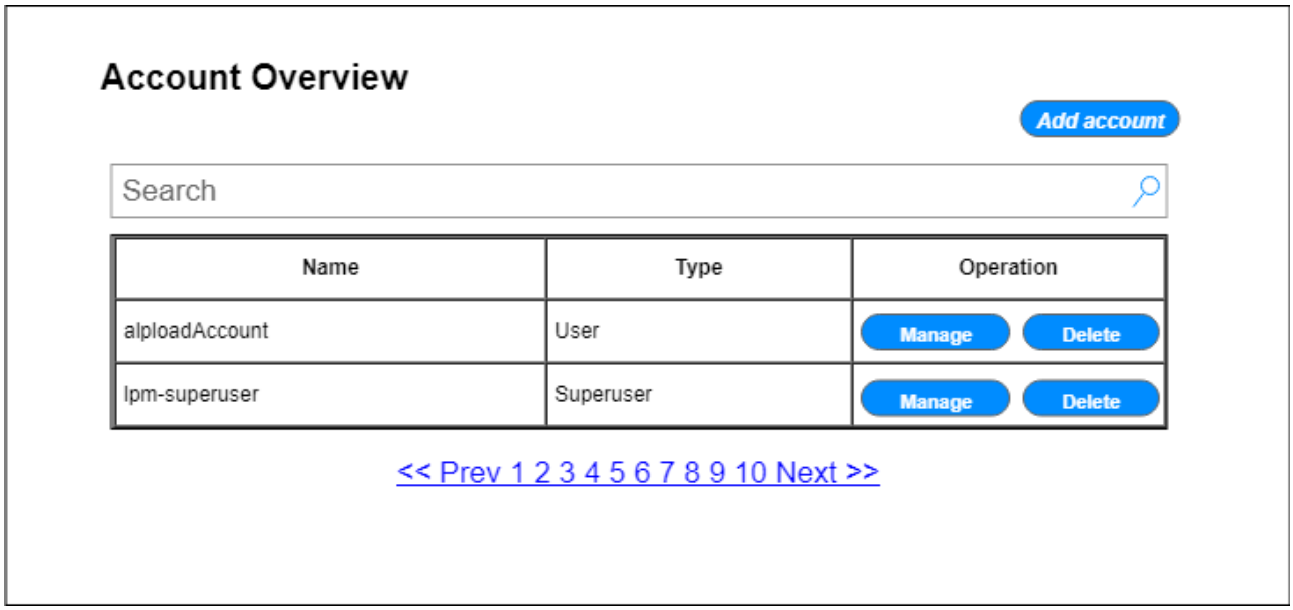


Abb. 8: Mockup Accountübersicht

CRUD Account

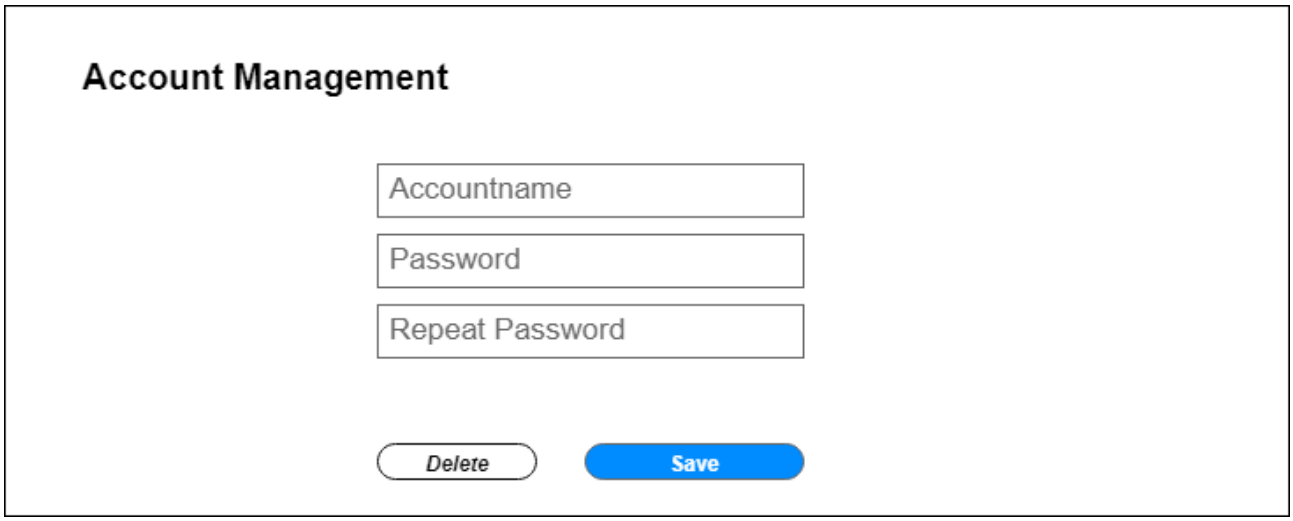


Abb. 9: Mockup CRUD Account

5.3 Gerätemanagement

Geräteübersicht

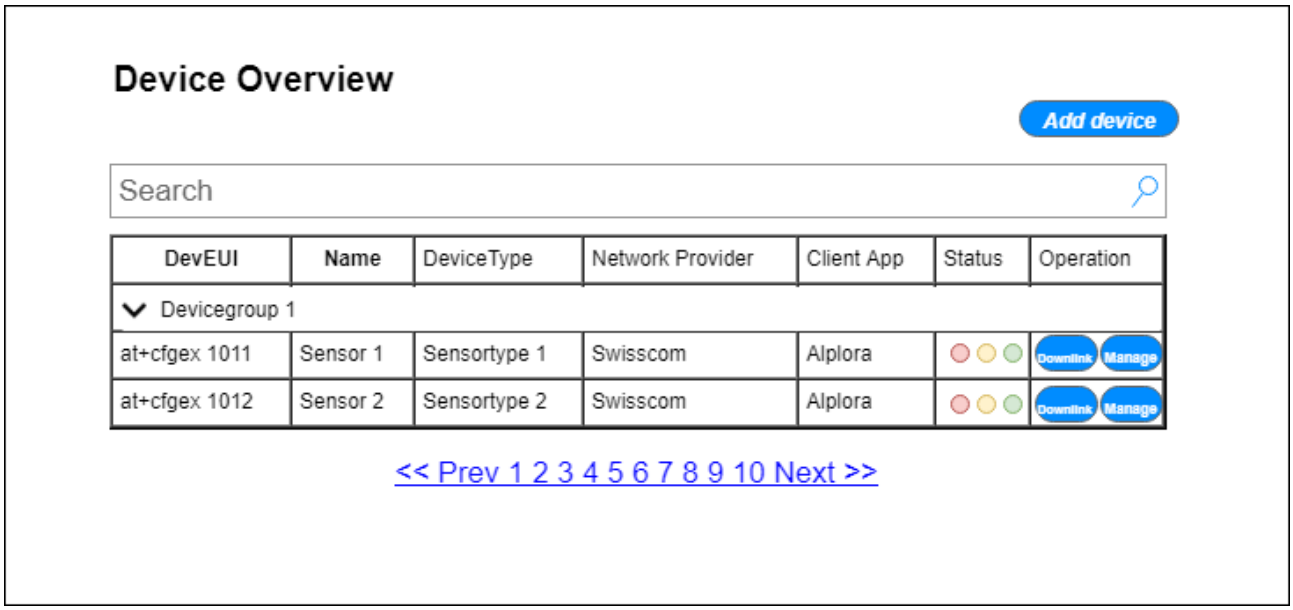


Abb. 10: Mockup Geräteübersicht

CRUD Device

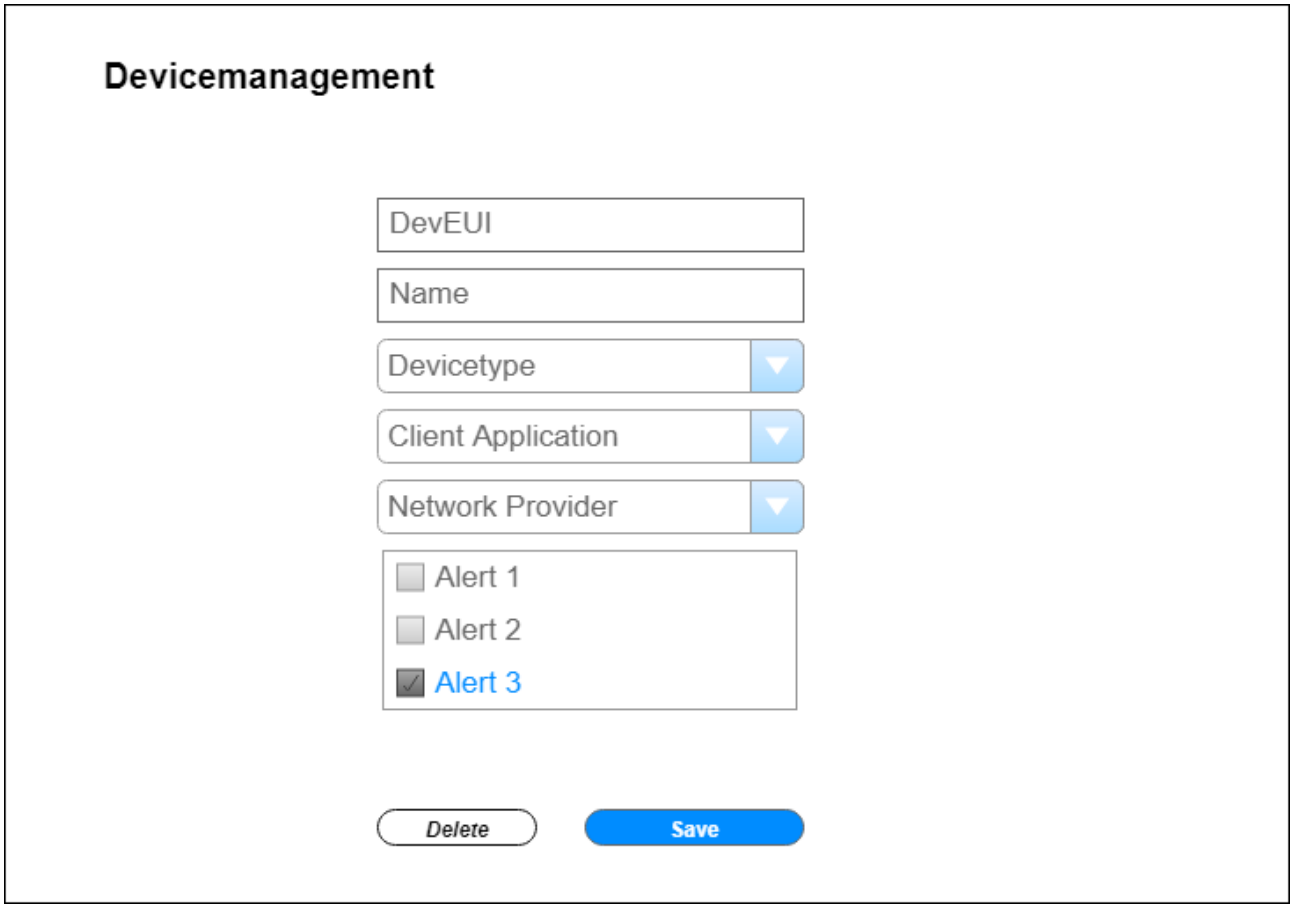


Abb. 11: Mockup CRUD Device

5.4 Gerätegruppenmanagement

CRUD Devicegroup

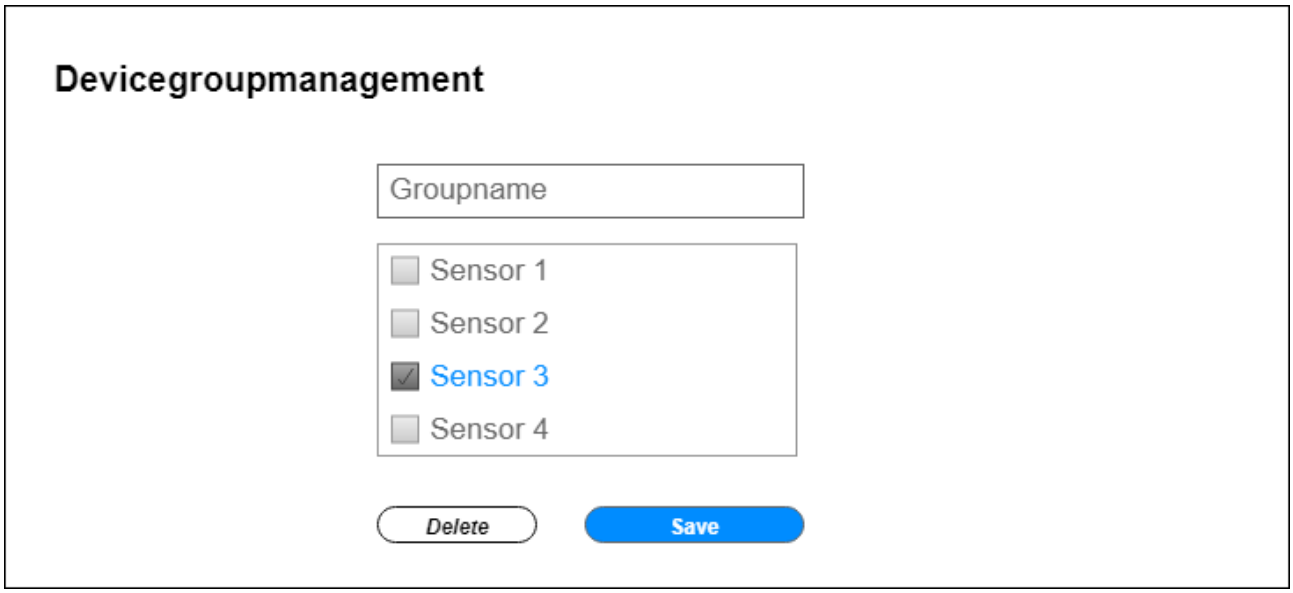


Abb. 12: Mockup CRUD Devicegroup-Screen

5.5 Gerätetypenmanagement

CRUD Devicetype

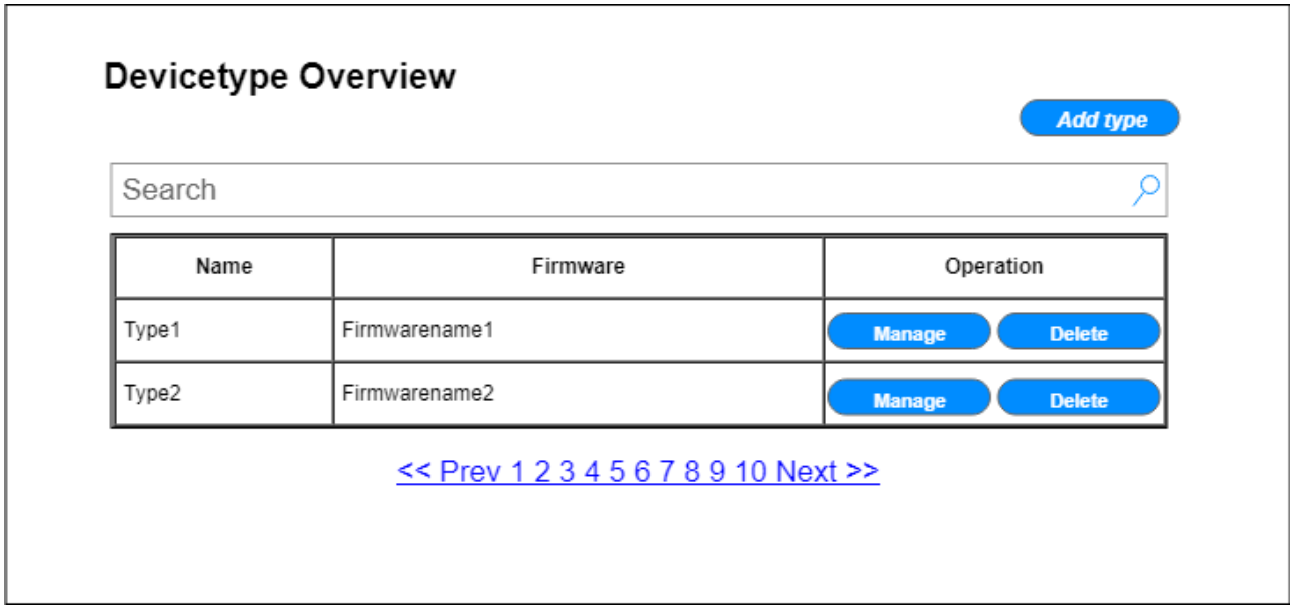


Abb. 13: Mockup Gerätetypenübersicht

CRUD Devicetype

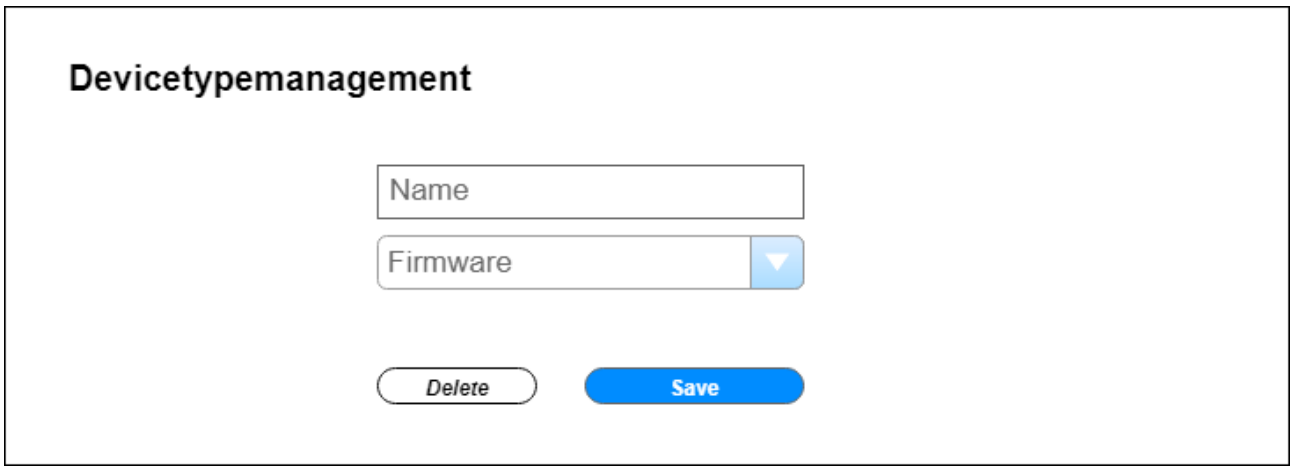


Abb. 14: Mockup CRUD Gerätetyp

5.6 Firmwaremanagement

Firmwareübersicht

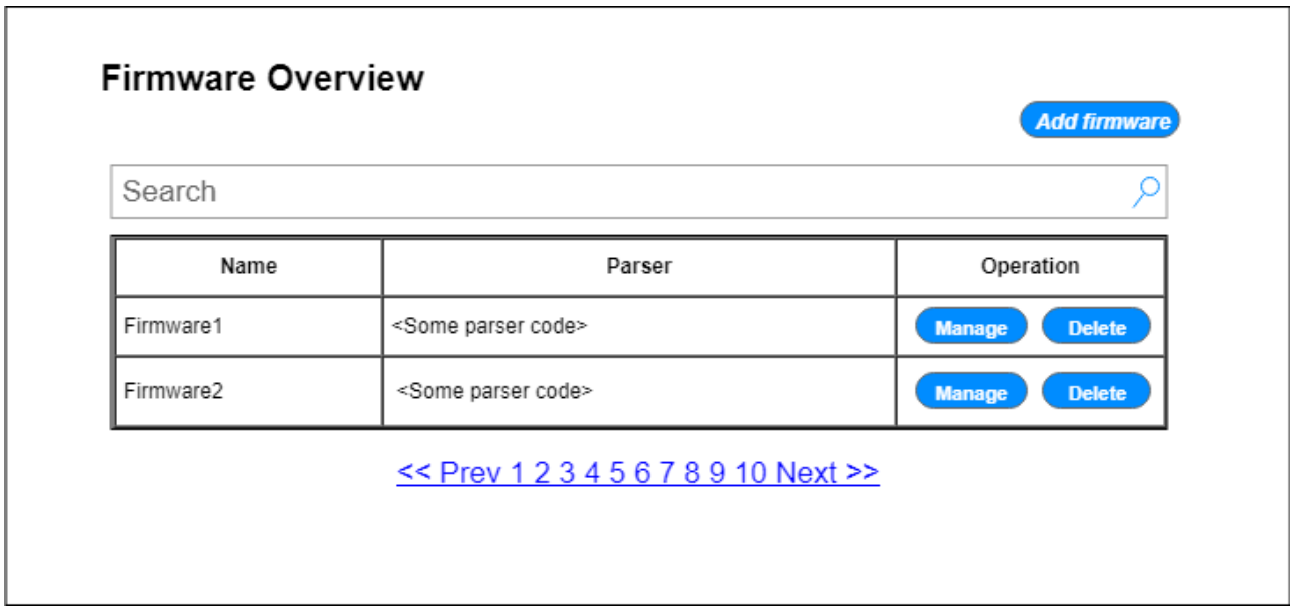


Abb. 15: Mockup Firmwareübersicht

CRUD Firmware

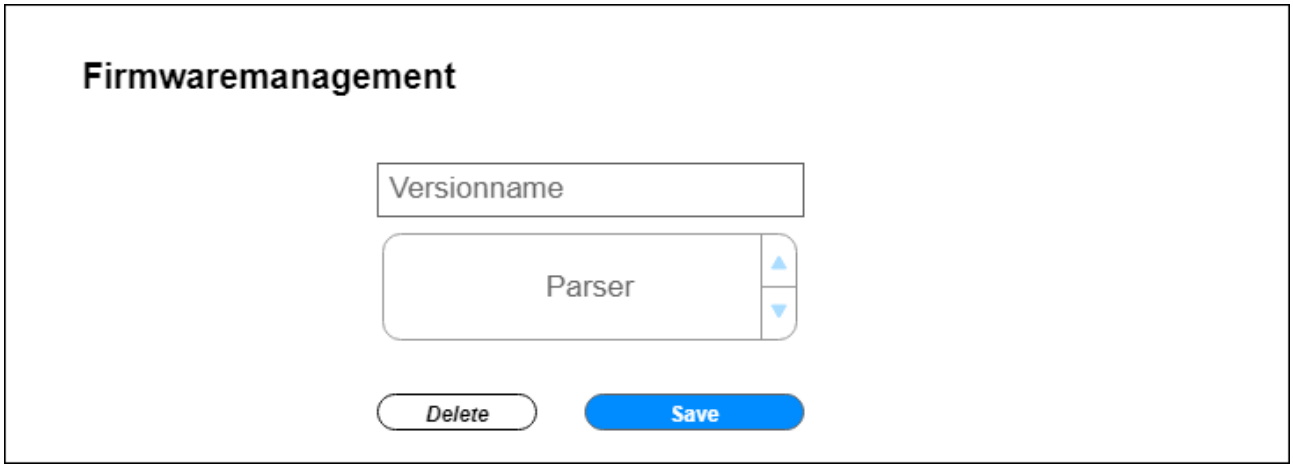


Abb. 16: Mockup CRUD Firmware

5.7 Client Application Management

Client Application Übersicht

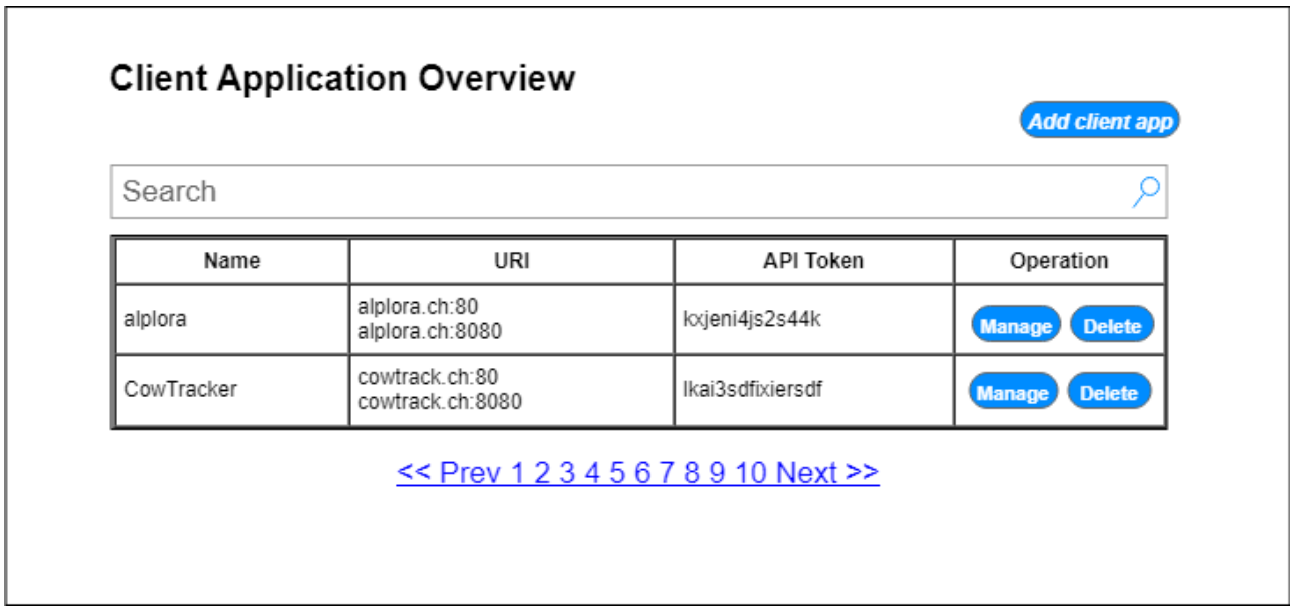


Abb. 17: Mockup Client Application Übersicht

CRUD Client Application

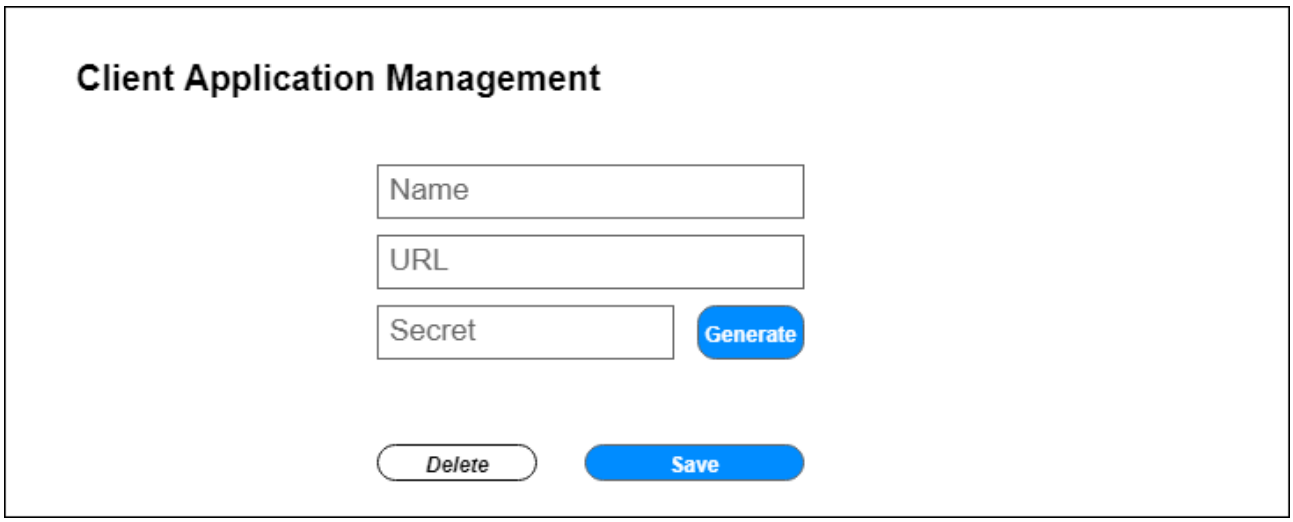


Abb. 18: Mockup CRUD Client Application

5.8 Network Provider Management

Network Provider Übersicht

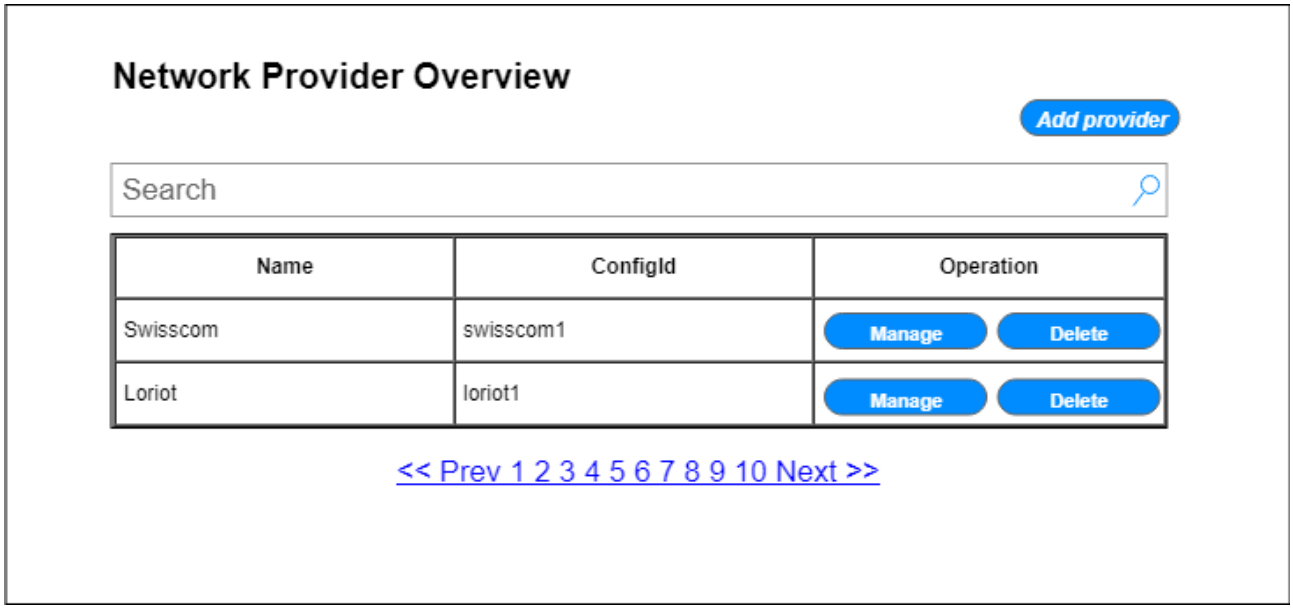


Abb. 19: Mockup Network Provider Übersicht

CRUD Networkprovider

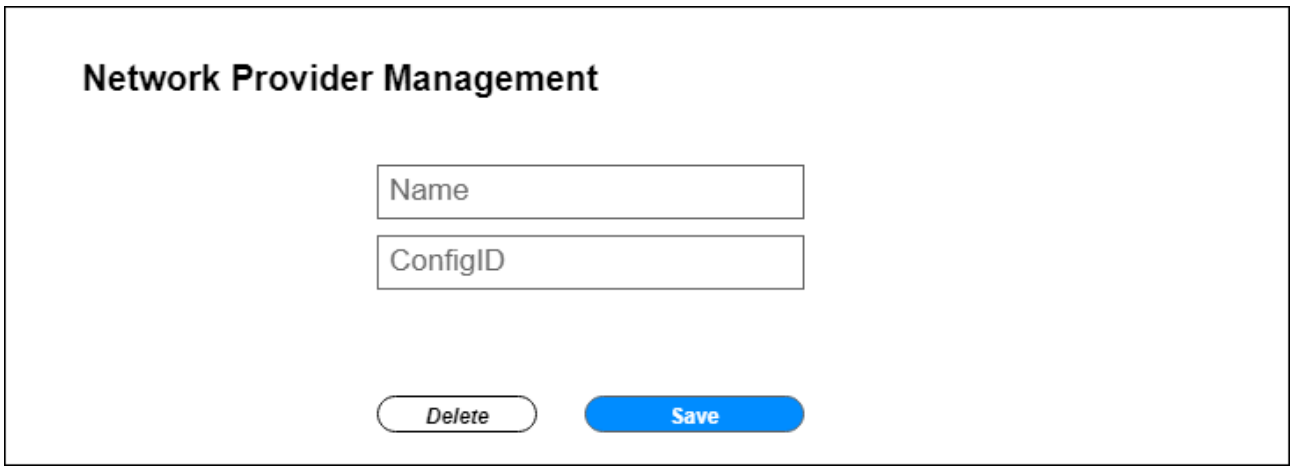


Abb. 20: Mockup CRUD Network Provider

5.9 Useralert Management

Useralert Übersicht

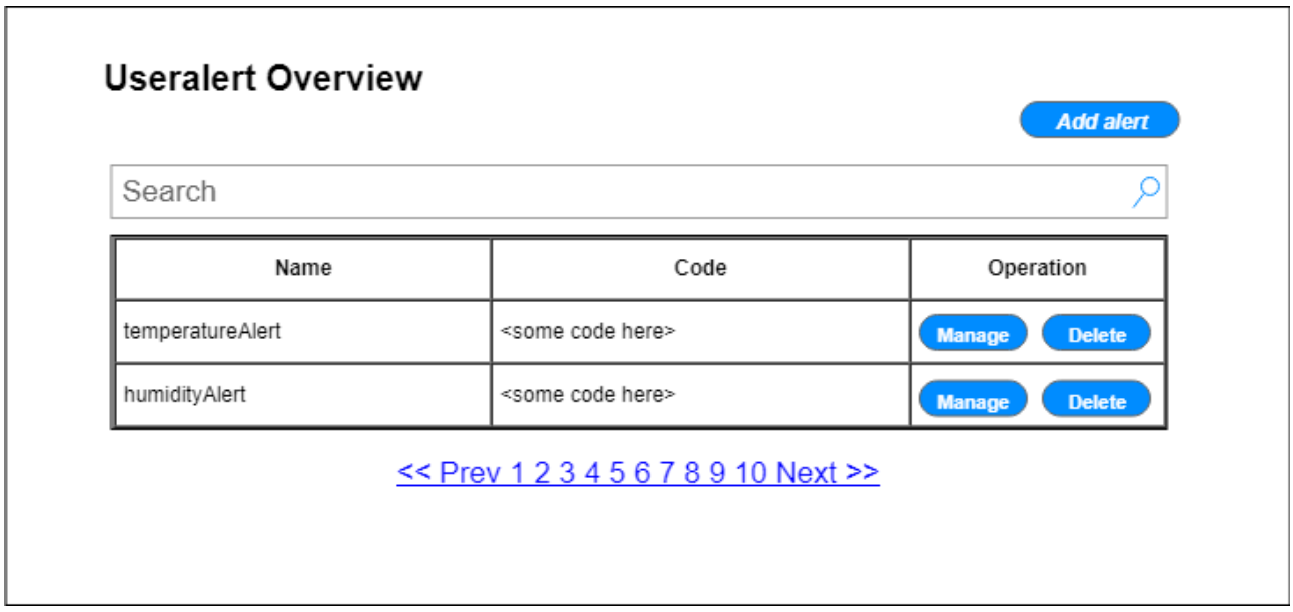


Abb. 21: Mockup Useralert Übersicht

CRUD Useralert

Abb. 22: Mockup CRUD Useralert

6 Konzeption und Design

6.1 Architektur

In diesem Kapitel beschreiben wir die Architektur der Applikation.

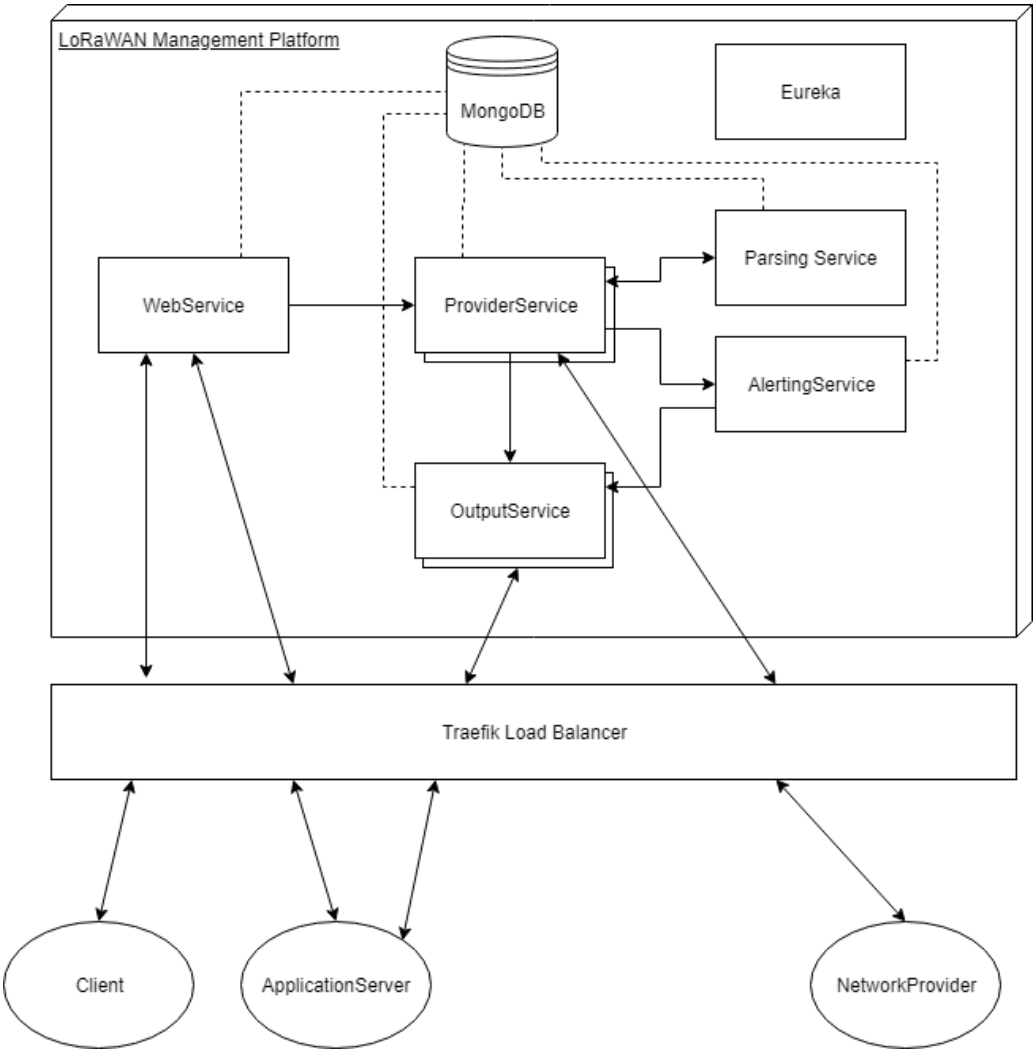


Abb. 23: Architektur

Alle Services bieten eine REST API an, über welche sie miteinander kommunizieren können. Versendet werden somit einfache HTTP-Requests.

6.1.1 Service Beschreibung

Traefik Traefik ist ein Open Source Reverse Proxy und Load Balancer.

Wir setzen Traefik ein, um die externen Zugriffe auf die Service Instanzen zu verteilen.

Muss der Load Balancer skaliert werden, so muss DNS Round Robin eingesetzt werden. Dabei wird bei der DNS Anfrage jeweils bereits der Traffic auf verschiedene IP Adressen verteilt.

Eureka Eureka ist ein Service der von Netflix entwickelt wurde. Mit ihm können Services gefunden werden.

Eureka besteht aus dem Eureka Server, bei welchem sich die Services registrieren müssen. Wenn nun ein Service einen anderen Service sucht, kann er mit dem Eureka Client eine Anfrage an den Eureka Server stellen, um den Service zu finden. Dabei macht der Client auch gleich das load balancing mittels dem simplen round-robin Algorithmus.

Auch der Eureka Server kann skaliert werden, indem weitere Instanzen gestartet werden. [12]

WebService Dieser Service stellt die Schnittstelle zu den Clients dar. Der WebService nimmt also die API Aufrufe entgegen. Die gesamteAPI-Beschreibung ist im Anhang zu finden.

Je nachdem leitet er nun die Anfragen an andere Services weiter, oder übernimmt die Aufgabe selber. Um die Architektur nicht zu verkomplizieren, haben wir uns entschieden, einfache Create/Read/Update/Delete (CRUD) Aufgaben direkt im WebService zu verarbeiten. Dazu braucht dieser Service Zugriff auf die MongoDB Datenbank.

Der WebService nutzt den ProviderService um Downlinks zu senden, und Geräte beim Netzwerk Anbieter zu registrieren.

ProviderService Von diesem Service gibt es eine Version für jeden Netzwerk Anbieter, also zum Beispiel einer für Swisscom und einer für Loriot.

Der ProviderService beinhaltet die gesamte Logik um mit dem entsprechenden Netzwerk Anbieter zu kommunizieren.

Wenn der ProviderService ein Uplink Paket erhält, nutzt er den ParsingService um das Paket zu parsen, und stösst den AlertingService an, um das Paket zu analysieren.

Danach nutzt er den OutputService um dem Applikations Server das Paket zu senden.

Dieser Service braucht Zugriff auf die MongoDB Datenbank um die Pakete zu speichern, und die Geräte zu registrieren.

Schnittstelle:

Method	URL	Beschreibung	Parameter
POST	/downlink	Sendet Downlink Paket an Gerät.	Downlink payload und DEVEui
POST	/device	Gerät registrieren	DEVEui
PUT	/device	Gerätdaten aktualisieren	DEVEui
DELETE	/device	Gerät löschen	DEVEui

OutputService Von diesem Service gibt es eine Version pro Output Channel. Also zum Beispiel einen für HTTP/POST und einen für InfluxDB.

Der OutputService kümmert sich um das Senden von Nachrichten an den Application Server. Dazu beinhaltet er die Logik, um mittels dem spezifischen Protokoll zu kommunizieren.

Der OutputService ist auch dafür zuständig, sich die Nachrichten, die nicht zugestellt werden konnten, zu speichern. Dazu braucht er Zugriff auf die MongoDB.

Schnittstelle:

Method	URL	Beschreibung	Parameter
POST	/message	Sendet Nachricht.	Nachricht (Json-Object), Geräte DEVEui

Das Paket, welches an diesen Endpunkt gesendet werden kann, soll im JSON Format gesendet werden, und folgende Felder enthalten:

Feld	Typ	Beschreibung
deveui	String	Die DevEUI zu welchem die Nachricht gehört
message	Object	Dieses Objekt wird serialisiert und der Client Applikation gesendet.

ParsingService Dieser Service parst den Payload der Uplink Pakete, wenn für den Gerätetyp ein Parser hinterlegt wurde.

Schnittstelle:

Method	URL	Beschreibung	Parameter
POST	/parse	Führt Code aus	Code, Timeout in ms, Variablen die an den Code übergeben werden sollen

AlertingService Der AlertingService ist dafür zuständig, beim Empfang von Datenpaketen diese zu analysieren und gegebenfalls einen Alarm auszulösen.

Dazu nutzt er den ParsingService, um den vom Benutzer definierten Alert-Code auszuführen. Der Output wird dann analysiert, und falls ein Alert ausgegeben wurde, wird dieser in der Datenbank gesichert.

Zu einem Alert gehört nicht nur der vom Benutzer definierte Python-Code, sondern auch ein State. Dieser State wird bei jeder Codeausführung bereitgestellt, und erlaubt es, komplexere Logiken zu implementieren. Zum Beispiel ein Durchschnitt über die Empfangsstärke der letzten N Uplinks. Damit können alle Alerts, welche wir in der Problemanalyse eruiert haben, implementiert werden.

Schnittstelle:

Method	URL	Beschreibung	Parameter
POST	/alert	Uplink Paket prüfen	Uplink ID
POST	/alert/test	Teste Alertcode	Uplink Paket, Alertcode & State

6.1.2 Wichtige Verbindungen im System

Netzwerk Server zu Management Plattform

Wenn ein Sensor einen Uplink sendet, wird dieser vom Netzwerk Server an die Management Plattform weitergeleitet.

Je nach Anbieter, kann dies unterschiedlich geschehen. Deshalb muss hier pro LoRaWAN Netzwerkanbieter ein eigenes Interface bereitgestellt werden. Das muss also in der Architektur der Applikation berücksichtigt werden.

Die Management Plattform soll alle Pakete annehmen, auch wenn sie (noch) keinem Gerät zugewiesen werden können.

Swisscom Die Swisscom benutzt zur Übermittlung der Daten ein HTTP/POST mit wahlweise einem JSON oder XML Payload. [13]

Ein Uplink Paket sieht bei der Swisscom wie folgt aus:


```
{
    "DevEUI_uplink":{
        "MType":"2",
        "FCntUp":"869",
        "MeanPER":"0.553657",
        "mic_hex":"b3eb4c04",
        "ModelCfg":"0",
        "SpFact":"12",
        "InstantPER":"0.545455",
        "LrrRSSI":"-119.000000",
        "Late":"0",
        "Lrrs":{
            "Lrr":[
                {
                "LrrRSSI":"-119.000000",
                "Lrrid":"080E066B",
                "Chain":"0",
                "LrrESP":"-136.085800",
                "LrrSNR":"-17.000000"
                }
            ]
        },
        "Lrcid":"00000401",
        "FPort":"1",
        "DevEUI":"0059AC0000151FE7",
        "Lrrid":"080E066B",
        "CustomerData":{
            "alr":{
                "ver":"1",
```

```
                "pro":"LORA/Generic"
            }
        },
        "LrrLAT":"47.512962",
        "DevLrrCnt":"1",
        "payload_hex":"0101f574c2004896810c10ee00000000",
        "FCntDn":"109",
        "DevAddr":"08028E3E",
        "SubBand":"G0",
        "Channel":"LC5",
        "Time":"2019-04-01T11:27:44.908+02:00",
        "LrrSNR":"-17.000000",
        "NbTrans":1,
        "LrrLON":"8.043602",
        "CustomerID":"100001174",
        "TxPower":14.0
    }
}
```


Bei der Swisscom kann entschieden werden, ob man die Authentifikation ein oder ausschalten möchte.

Loriot Der Netzwerkanbieter Loriot bietet diverse Varianten an, über welche die Daten weitergeleitet werden. Wie bei der Swisscom auch, bietet Loriot die Möglichkeit, die Daten per HTTP/POST auf eine vordefinierte URL zu erhalten. Des Weiteren kann hier noch ein "Authorization" Header Wert definiert werden.

Hier die Beschreibung eines Loriot Uplink Paketes:

```
{
    cmd      : 'rx'; // identifies type of message, rx = uplink message
    EUI      : string;  // device EUI, 16 hex digits (without dashes)
    ts       : number;  // server timestamp as number (miliseconds from Linux epoch)
    ack      : boolean; // NEW! acknowledgement flag as set by device
    fcnt     : number;  // frame counter, a 32-bit number
    port     : number;  // port as sent by the end device

    encdata? : string;  // data payload (APPSKEY encrypted hex string)
                        // only present if APPSKEY is assigned to device

    data?    : string;  // data payload (decrypted ,plaintext hex string)
                        // only present if APPSKEY is not assigned to device

    // extended radio information

    freq     : number;  // radio frequency at which the frame was received, in Hz
    dr       : string;  // radio data rate - spreading factor, bandwidth and coding rate
                        // e.g. SF12 BW125 4/5
```

```
                        // UPDATED: previously the field was called 'sf'
    rssi      : number;  // frame rssi, in dBm, as integer number
    snr       : number;  // frame snr, in dB, one decimal place
}
```

Wobei die "extended radio information" nur gesendet werden, wenn man bei Loriot einen "commercial account" hat.

Management Plattform zu Netzwerk Server

Wenn die Management Plattform mit dem Server der Netzwerkanbieter kommunizieren muss, um Downlink Pakete zu senden, oder neue Sensoren zu registrieren, ist das je nach Anbieter unterschiedlich.

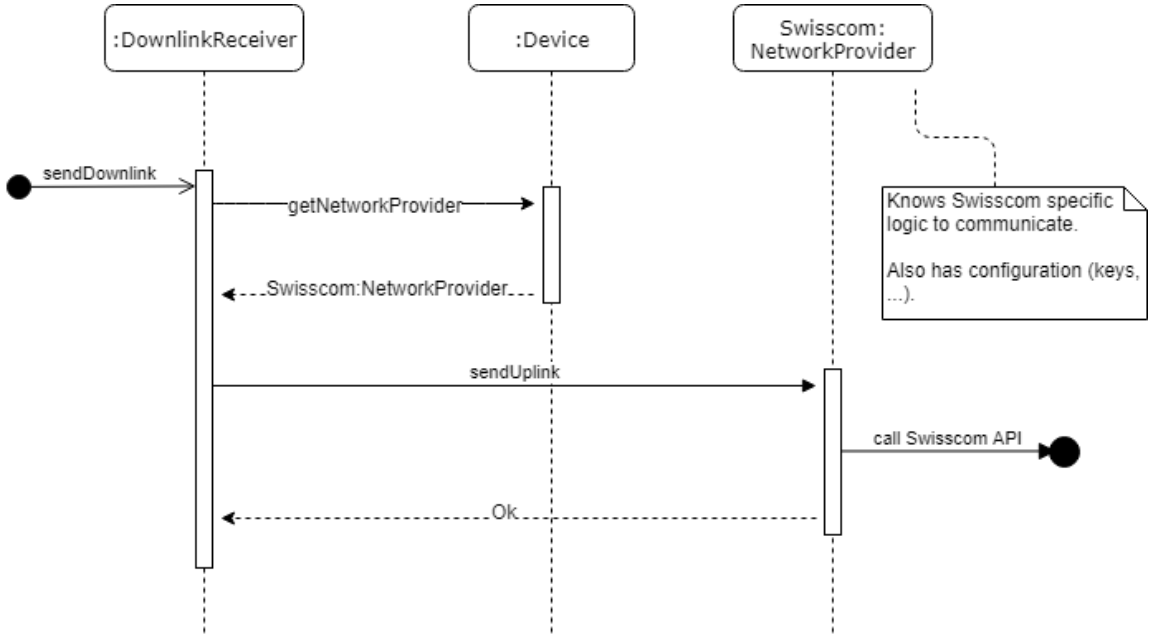


Abb. 24: Auswahl des Netzwerkanbieters am Beispiel "Downlink senden"

Die Swisscom bietet zum Senden von Downlinks eine Schnittstelle an, an welche man eine HTTP/POST Anfrage schicken kann. [13] Um Endgeräte zu verwalten, wird eine andere REST API zur Verfügung gestellt. [14] Diese ist von ThingPark, das Produkt welches von der Swisscom eingesetzt wird, um den LoRaWAN Service anzubieten.

Loriot Auch Loriot bietet eine REST API an, mit der man Downlinks senden kann. [15] Über diese API kann man auch Geräte verwalten, wenn man den dafür nötigen Zugang erworben hat.

Ein Downlink Paket sieht bei Loriot wie folgt aus:

```
{
    cmd         : 'tx';   // must always have the value 'tx'
```

```
    EUI        : string;  // device EUI, 16 hex digits (without dashes)
    port       : number;  // port to be used (1..223)
    confirmed? : boolean; // (optional) request confirmation (ACK) from end-device
    data       : string;  // data payload (to be encrypted by our server)
}
```

Oder falls man selbst die Verschlüsselung der Daten vornimmt:

```
{
    cmd        : 'tx';  // must always have the value 'tx'
    EUI        : string;  // device EUI, 16 hex digits (without dashes)
    port       : number;  // port to be used (1..223)
    confirmed? : boolean; // NEW! (optional) request confirmation (ACK) from end-device
    encdata    : string;  // data payload (already APPSKEY encrypted)
    seqno      : number;  // must correspond to the latest seqdn reported
                          // by the 'txd' message
}
```

Und sobald das Paket an den Gateway zur Auslieferung gesendet wird, wird folgendes Paket an die Management Platform gesendet:

```
{
    cmd      : 'txd'; // always has the value 'txd'
    EUI      : string;  // device EUI, 16 hex digits (without dashes)
    seqdn    : number;  // sequence number used for the downlink
    ts       : number;  // unix timestamp, moment of the transfer to gateway
}
```

Management Plattform zu Application Server

Die Management Plattform sendet die empfangenen Downlink Pakete und Alerts an den Application Server weiter. Hier bietet man dem Application Server eine Auswahl an (Output Channels), wie er die Daten gerne empfangen würde. Ein Application Server kann auch mehrere Output Channel haben. In diesem Fall wird die Nachricht an jeden gesendet.

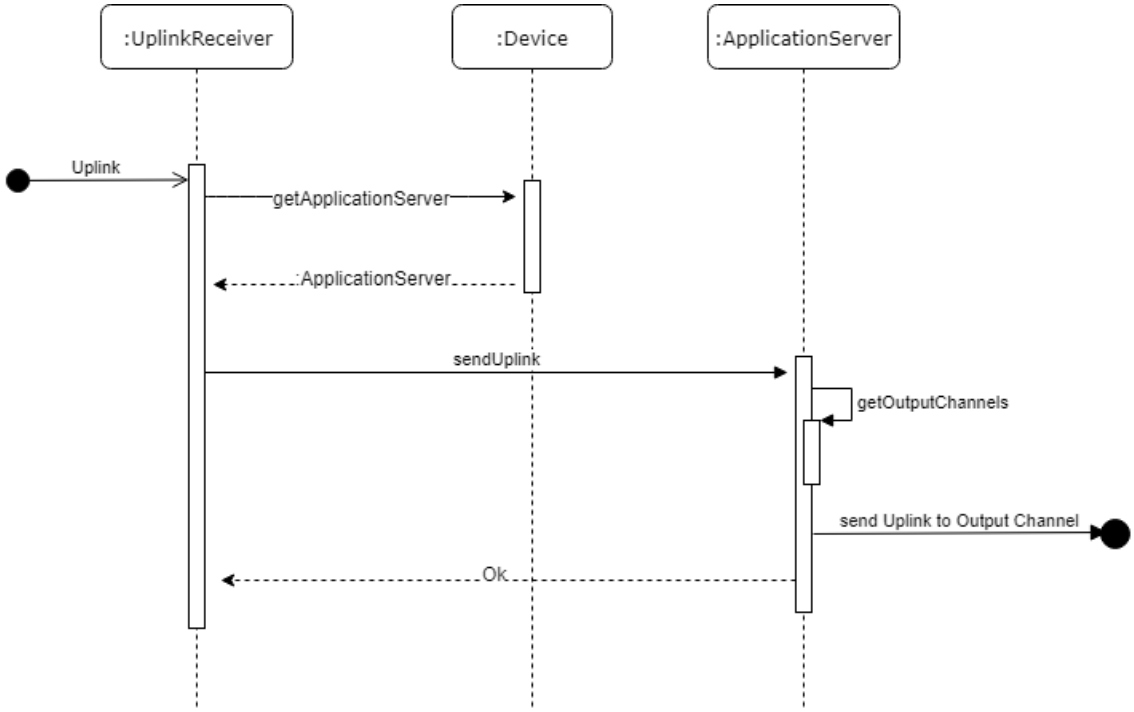


Abb. 25: Auswahl der Output Channels am Beispiel "Uplink senden"

Output Channel spezifische Fehlerfälle müssen pro Output Channel abgehandelt werden.

HTTPS/POST Ein Output Channel ist, die Daten per HTTPS/POST zu senden. Dazu muss eine URL hinterlegt werden, unter der die Daten Empfangen werden können. Auch muss ein "pre shared secret" hinterlegt werden, das zur Authorisierung bei jedem Request mitgesendet wird.

Damit das "pre shared secret" nicht von dritten mitgelesen werden kann, muss die Verbindung zwingend über HTTPS geschehen (nicht HTTP).

Das Senden per HTTPS/POST hat den Vorteil, dass das Empfangen der Daten beim Application Server sehr einfach implementiert werden kann, da mit vielen Frameworks ein solcher HTTPS Endpunkt schnell erstellt ist.

Die Daten werden im Body des Requests als JSON codierte Daten gesendet. Im Header des Requests wird das Feld "Authorization" mit dem hinterlegten "pre shared secret" gesendet.

InfluxDB Auch soll es möglich sein, Daten direkt in eine Influx-Datenbank der Client Applikation zu schreiben. Dies wird auch als Output Channel realisiert.

Uplink Paket Ein Uplink Paket, welches an den Application Server gesendet wird, ist wie folgt definiert. Ein JSON Objekt welches folgende Felder beinhaltet:

```
{
    devEUI      : string;  // Device EUI without separators
```

```
    date        : string;  // datetime
    framecount  : number;  // frame counter, a 32-bit number
    port        : number;  // port as sent by the end device
    data        : string;  // data payload (decrypted ,plaintext hex string)
    sf          : number;  // (optional) the spreading factor
    rssi        : number;  // (optional) frame rssi, in dBm, as integer number
    snr         : number;  // (optional) frame snr, in dB, one decimal place
}
```

Wobei die mit "(optional)" gekennzeichneten Felder je nach Netzwerkanbieter auch leer sein können. Zusätzlich werden noch die Felder welche beim Parsen generiert werden mitgeschickt.

Application Server zu Management Plattform

Die Management Plattform bietet eine REST API an. Diese wurde mit Swagger dokumentiert, was es dem Entwickler des Application Server sehr leicht macht, die API zu entdecken. Die Beschreibung befindet sich im Anhang der Arbeit oder kann interaktiv [1] eingesehen werden.

6.1.3 Up-/Downlink durch die Architektur

Die folgenden Sequenzdiagramme beschreiben, wie die einzelnen Services in der gesamten Architektur miteinander zusammenspielen, um einen Up- bzw. Downlink zu verabeiten.

[1] `https://app.swaggerhub.com/apis-docs/raffivoegeli/LorawanManagementPlattformAPI/1.0.0-oas3`

Uplink

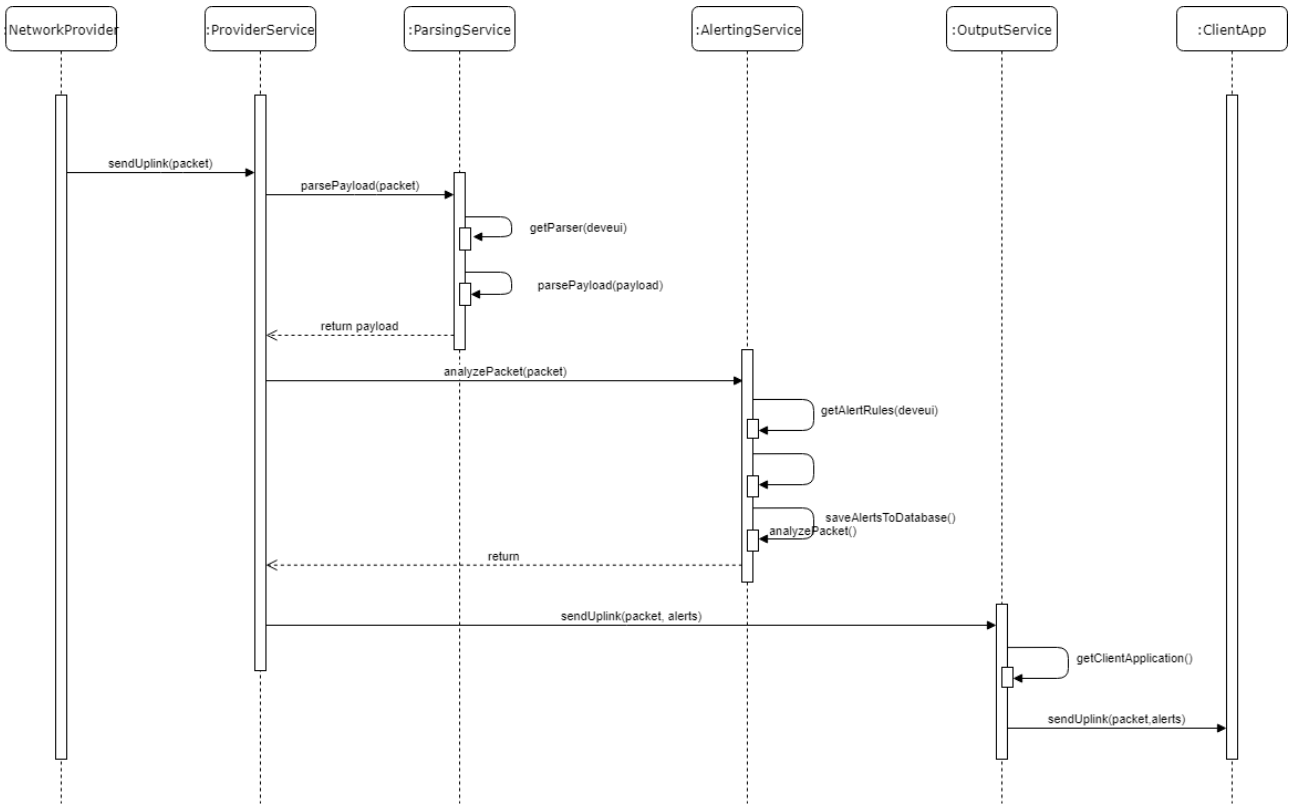


Abb. 26: Uplinkverarbeitung durch Systemarchitektur

Downlink

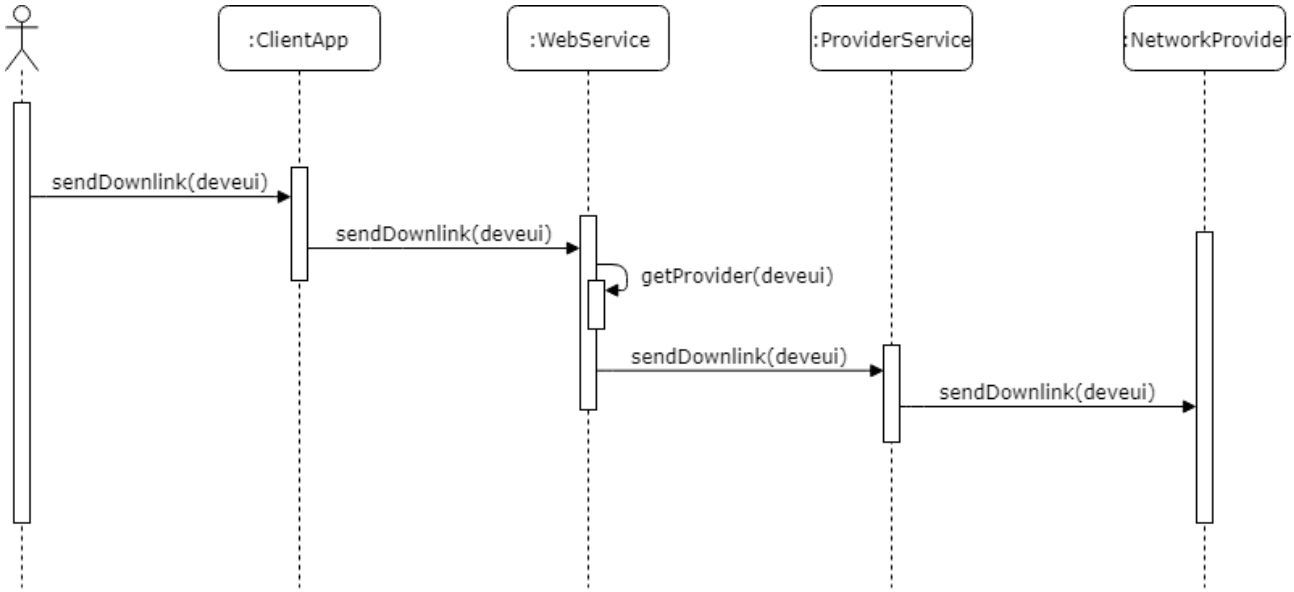


Abb. 27: Downlinkverarbeitung durch Systemarchitektur

6.2 Datenbank

In MongoBD werden sogenannte Dokumente in Collections gespeichert. Diese Dokumente können Referenzen auf andere Dokumente enthalten. Dadurch ergibt sich folgende Datenstruktur:

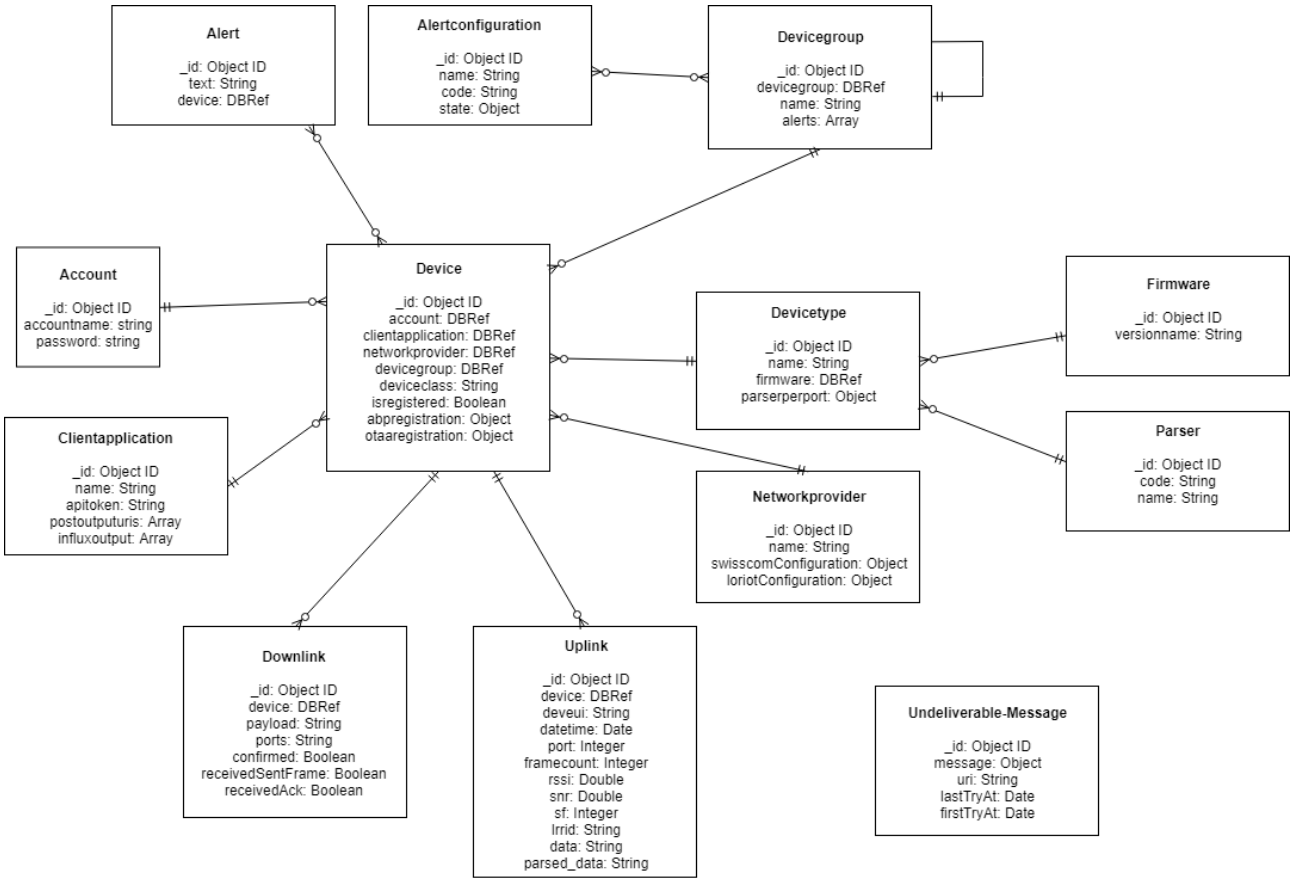


Abb. 28: Datenbankmodell

Die Devicegruppen und Devices werden gemäss dem Composite-Pattern miteinander verbunden.

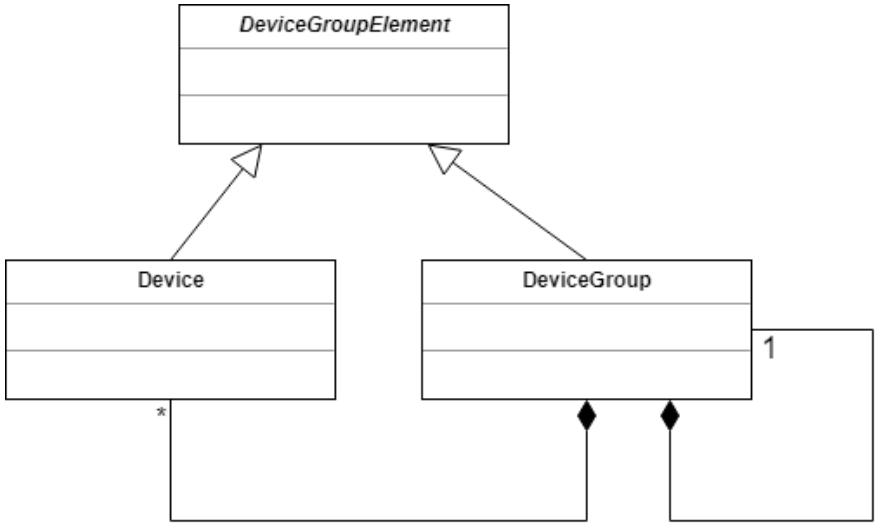


Abb. 29: Composite-Pattern Devicegroup/Devices

6.3 Testkonzept

6.3.1 Unit Testing

Um die Unit Tests nicht auf der original MongoDB-Instanz durchführen zu müssen, werden die Repository-Zugriffe gemockt.

Hierfür kann für das Unit Testing in der Unit-Testklasse in Spring Boot das Mockito-Framework verwendet werden. Mockito erlaubt es, Repositories zu mocken sowie Services mit als Bean-Instanz Mock zur Verfügung zu stellen.

Die Tests können lokal, sowie auch vom Build-Server ausgeführt werden.

6.3.2 System Testing

Um das System auf seine geforderten Funktionalitäten, welche als Use Cases definiert wurden, testen zu können, wird ein Testprotokoll mit System Tests definiert. Dieses Testprotokoll besteht aus Vorbedingungen, einem erwarteten Ergebnis sowie der Beschreibung des eingetretenen Resultats. Die System Tests sowie eine Ausführung der Tests befinden sich im Anhang der Arbeit.

Neben den Funktionalität-Tests sollen auch noch Security-Tests durchgeführt werden, in welchem das Verhalten der Plattform bezüglich fehlender oder fehlerhaften Bearer Tokens geprüft wird. Diese werden ebenfalls mit Hilfe von System Tests durchgeführt.

6.3.3 Load Testing

Um die Skalierbarkeit der Applikation realitätsnah testen zu können, werden Lasttests für die API Anfragen an den jeweiligen Service durchgeführt. Wie im Kapitel Nicht-Funktionale Anforderungen

definiert, muss das System in der Lage sein, 500 Up- bzw. Downlinks pro Minute zu verarbeiten. Dies entspricht 8.33 Requests pro Sekunde.

Aufbau der Tests

Verwendet wird Locust[2], ein Open Source Load Testing Tool, mit welchem man einfache HTTP-Requests an die REST-API schicken kann. Dabei kann die Anzahl Benutzer, welche gleichzeitig einen Request absetzen, simuliert werden. Getestet wird einerseits der ProviderService, welcher die Uplinks entgegen nehmen muss und andererseits der Webservice, welcher die von Benutzer initiierten Downlinks weiterleitet. Die einzelnen Microservices können dabei in der Anzahl der Instanzen variiert werden.

6.4 Monitoring- & Loggingkonzept

In den folgenden Kapiteln möchten wir einen Vorschlag machen, wie man Monitoring und Logging in Zukunft einsetzen müsste.

6.4.1 Monitoring

Im Betrieb müssen die einzelnen Services überwacht werden. Das Spring Boot Framework hat bereits Monitoring integriert. Der sogenante Actuator ist ein HTTP Endpunkt, den man auf dem Service ansprechen kann, und welcher Informationen über die Gesundheit des Services bereitstellt.

Ein Projekt, welches genau dies macht, ist Spring Boot Admin[16]. Wir empfehlen dieses Tool, da es den Gesundheitsstatus jeder Serviceinstanz anzeigt, und Alarme versenden kann, wenn ein Service ausfällt.

Ein weiterer Vorteil ist, dass man die einzelnen Services nicht von Hand bei Spring Boot Admin registrieren muss, sondern dass dieser automatisch die Services über Eureka ausfindig machen kann[3].

6.4.2 Logging

Um die Loggingdaten brauchbar zu machen, sollte jeder Request mit einer Trace-ID versehen werden. Dies, da ein Request oft Requests an andere Services auslöst, und man den Zusammenhang dieser sehen möchte.

Um dies zu erreichen empfehlen wir das Spring Projekt "Spring Cloud Sleuth" [4]. Dieses kann sehr einfach eingebunden werden, und fügt von da an sogenannte Span-IDs und Trace-IDs den einzelnen Request hinzu, und führt diese in den Loggingdateien auf.

Des Weiteren sollten dann die Loggingdaten an einen Zentralen Loggingservice gesendet werden, auf welchem man Auswertungen machen kann. Einer dieser Services ist Papertrail[17].

[2] `https://locust.io/`
[3] `https://codecentric.github.io/spring-boot-admin/2.1.4`
[4] `https://spring.io/projects/spring-cloud-sleuth`

Papertrail ist Cloud basiert, kann aber gratis ausprobiert werden. Der Vorteil ist, dass man die Logs sehr einfach an Papertrail senden kann, vor allem wenn man Docker verwendet um die Microservices zu betreiben. Hier eine Übersichtsgrafik aus dem Buch "Spring Microservices in Action"[18]:

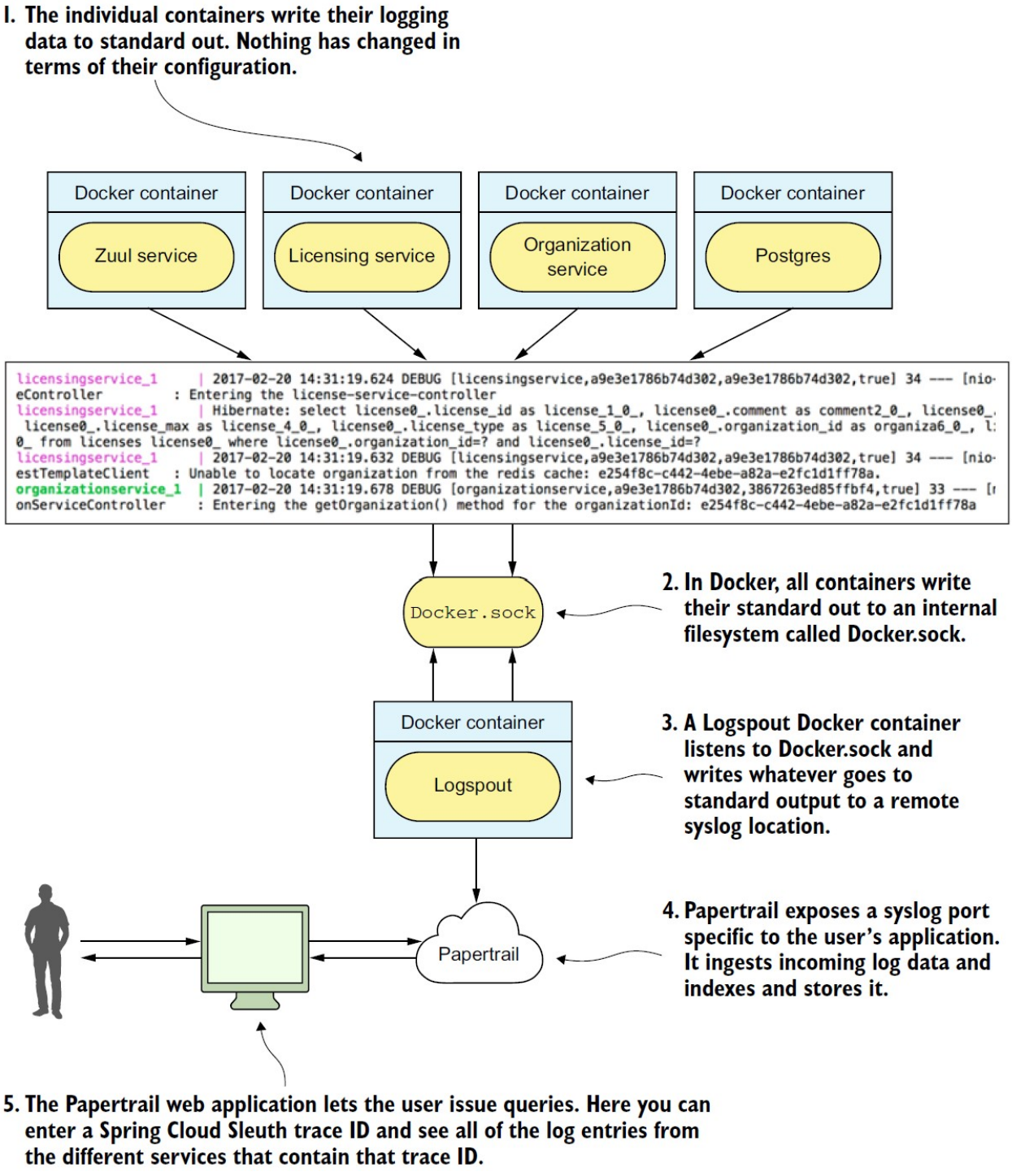


Abb. 30: Wie man Docker, Logspot und Papertrail verwendet um einfach eine Logging-Architektur zu erstellen

Um über Fehler informiert zu werden, empfehlen wir Sentry[19]. Sentry ist sehr einfach einzubinden[5]. Danach fängt es Exceptions ab und leitet diese weiter an den Sentry Service. Dieser erfasst den Fehler inklusive Stack Trace, und bietet eine Vielzahl an Möglichkeiten über das Auftreten des Fehlers informiert zu werden.

6.5 Sicherheitskonzept

6.5.1 Benutzer der Plattform

Die Management Plattform unterscheidet zwischen zwei verschiedenen Benutzertypen, bei welchen die Authentifizierung unterschiedlich gehandhabt wird:

API-Benutzer (Client Applikation) Ein API-Benutzer, sprich eine Client Applikation, authentifiziert und autorisiert sich bei der Plattform und kommuniziert danach via API mit der Plattform.

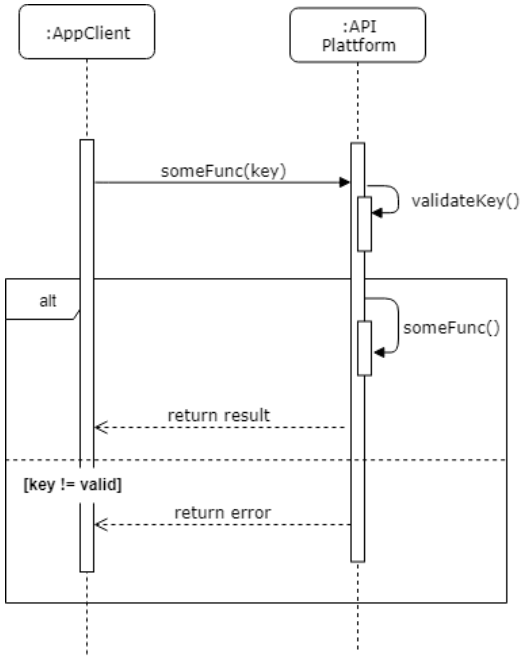


Abb. 31: Login API-Benutzer

Die Authentifizierung erfolgt über ein Pre-Shared Secret, das als Bearer Token mitgesendet wird. Beim Erstellen einer Client Application auf der Management Plattform wird es generiert und bei jedem Request neu geprüft. Dazu muss im Header der Anfrage das Feld "Authorization" wie folgt gesetzt werden, wobei "<token>" mit dem tatsächlichen Token ausgetauscht werden muss:

```
Authorization: Bearer <token>
```

Da das Token im Header mitgesendet wird, muss sichergestellt werden, dass nur per HTTPS mit der API kommuniziert werden kann. Bei jedem Request wird die Validierung des Tokens durchgeführt.

[5] `https://sentry.io/for/spring/`

Frontend-Benutzer Der Frontend-Benutzer repräsentiert einen Kunden oder Techniker, welcher die Plattform über das zur Verfügung gestellte, grafische Interface bedient.

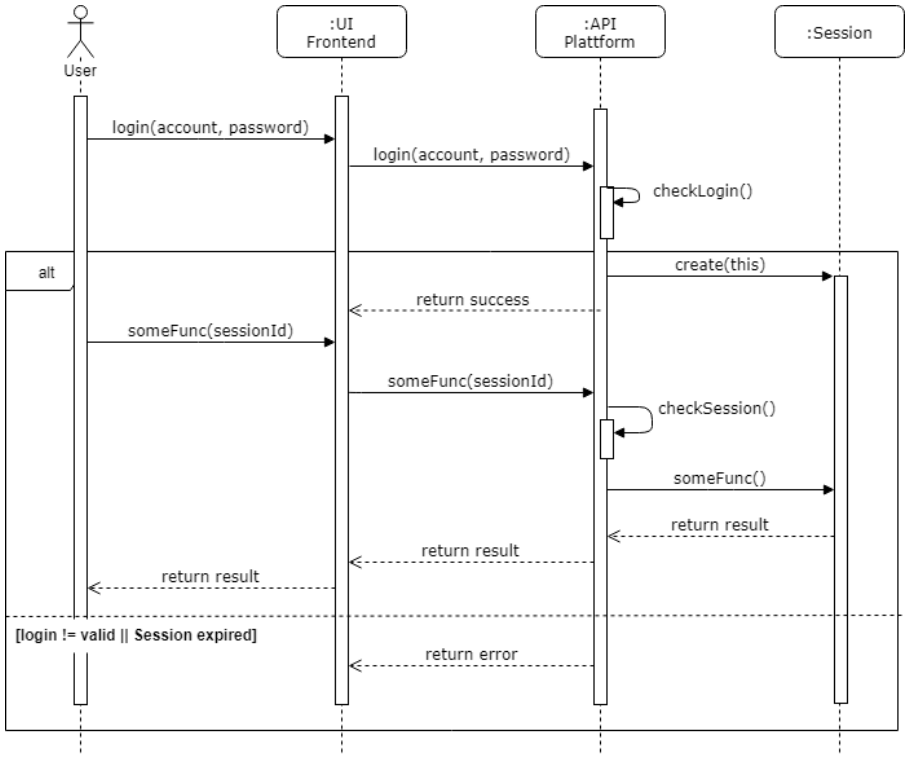


Abb. 32: Login Frontend-Benutzer

Die Authentifizierung erfolgt über ein Login mit Accountname und Passwort, welches durch den Superuser erstellt werden kann. Danach erhält er ein Token, welches für eine gewisse Zeit gültig ist. Wenn die Gültigkeit des Tokens abläuft, ist es dem Benutzer verwehrt, weitere Requests auszuführen. Er wird im Frontend zudem automatisch ausgeloggt.

6.5.2 Microserviceskommunikation

Die Services sollen untereinander authentifiziert kommunizieren. Das bedeutet, dass bei jedem Aufruf ein Token im Header der Anfrage mitgesendet werden und vom Gegenüber geprüft werden muss. Die Tokens sind im Vorhinein bereits abgemacht und gelten somit als API-Key, wobei jeder Service die benötigten Keys für die anderen Services besitzt.

7 Resultate und Erkenntnisse

7.1 Entwicklungsprozess

Die einzelnen Projekte werden in jeweils eigenen Git Repositories entwickelt. Wir nutzen GitLab.com, was den Vorteil hat, dass wir private Repositories haben, und das GitLab CI nutzen können.

Wir nutzen das CI um die Services jeweils als Docker Container zu erstellen, und, in der ebenfalls in GitLab integrierten, Registry zu speichern. Dies hat den Vorteil dass man immer einen Docker Container mit der aktuellsten Version des Services zur Verfügung hat, was das Updaten und Installieren der Services sehr einfach macht.

Das CI führt auch jeweils die Tests des Services aus, und bricht den Build ab, falls diese nicht erfolgreich waren.

7.2 Microservices allgemein

Die Microservices sind, ausgenommen vom Parsing Service, als Java Spring Boot Projekt implementiert.

7.2.1 Architektur

Um eine Vereinheitlichung aller Spring-Boot Projekte sicherstellen zu können, besitzen alle dieselbe Grundstruktur ihrer Packages:

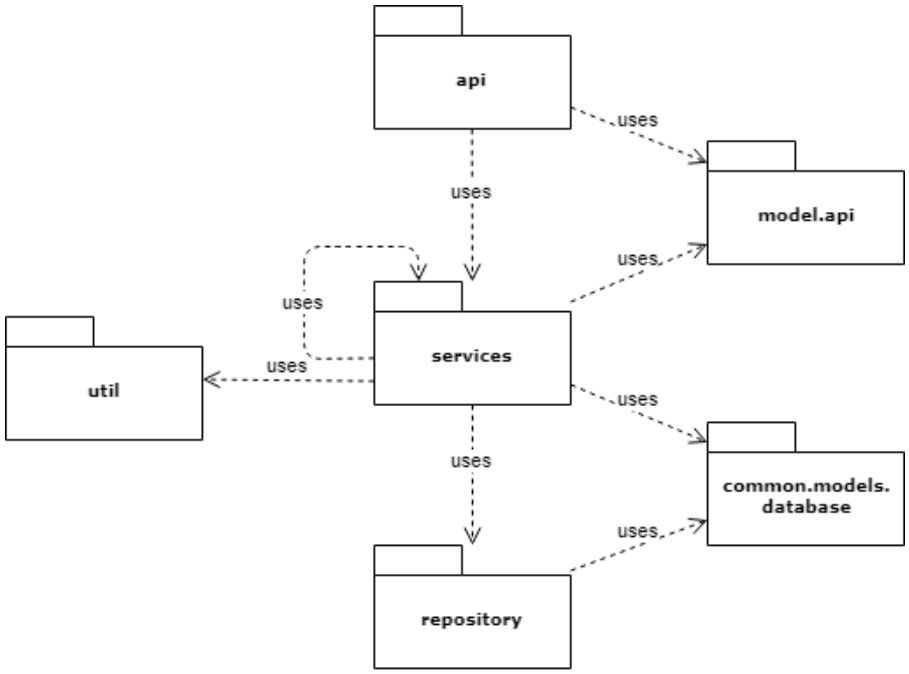


Abb. 33: Packagediagramm: Grundstruktur Services

Im api-Package liegen die Controller, welche die Endpunkte der API zur Verfügung stellen. Sie empfangen Daten mit Hilfe eines vordefinierten Data Transfer Objects (DTO), welches im model.api-Package hinterlegt ist und rufen die entsprechenden Service-Methoden auf.

Im services-Package ist die Businesslogik hinterlegt. Diese benutzt Hilfsfunktionen, welche sich im util-Package befinden. Manche Services rufen sich gegenseitig auf. Der Service kommuniziert ausserdem mit dem Repository, um eine Verbindung zur Datenbank herstellen zu können.

Wir erreichen dadurch eine Abstraktion zwischen den Controllern und der Businesslogik, sowie zwischen der Businesslogik und der Datenbank (Repositories). Dies erleichtert in Zukunft das Auswechseln von Komponenten, und macht das Testing einfacher.

7.2.2 Kommunikation

Damit die einzelnen Services miteinander kommunizieren können, stellt das Spring Cloud Framework einen deklarativen REST-Client namens Feign[6] zur Verfügung. Dabei werden die Service-Endpunkte als sogenanntes Feign-Interface deklariert und sollten der Signatur des entsprechenden Controllers entsprechen.

Listing 1: Feign-Client Interface

```
1   @Service
2   @FeignClient(name = "swisscom-provider-service", configuration =
        SwisscomProviderService.SwisscomFeignClientConfiguration.class)
3   public interface SwisscomProviderService {
4       class SwisscomFeignClientConfiguration {
5           @Bean
6           public BearerTokenRequestInterceptor
                bearerHeaderAuthRequestInterceptor
                (@Value("${lmp.swisscomproviderservice.apikey}") String
                apiKey) {
7               return new BearerTokenRequestInterceptor(apiKey);
8           }
9       }

11      @RequestMapping(
12              method= RequestMethod.POST,
13              value="/downlink",
14              consumes="application/json")
15      void postDownlink(DownlinkDto downlinkDto);
```

Als Konfiguration mit der Anntotation wird ausserdem bei jedem Request an diesen Microservice ein BearerTokenRequestInterceptor instanziert, welcher den API-Key als Request Header anfügt.

Die Applikation muss dabei noch mit der entsprechenden Annotiation versehen werden, damit Feign unterstützt wird. Da die Service-Interface Definitionen im Common-Projekt liegen, muss dem Projekt noch mitgeteilt werden, in welchen basePackages er nach den Konfigurationen suchen muss.

[6] https://cloud.spring.io/spring-cloud-openfeign/multi/multi_spring-cloud-feign.html

Listing 2: Enable Feign Client Usage

```
1  @EnableFeignClients(basePackages = "ch.hsr.common.services")
2  public class SwisscomProviderServiceApplication {
3      public static void main(String[] args) {
4          SpringApplication.run(SwisscomProviderServiceApplication.class,
               args);
5      }
6  }
```

7.2.3 Gemeinsame Dateien

Da jeder Service ein eigenes Projekt ist, jedoch alle Services gemeinsame Dateien teilen, benötigt es ein globales Projekt, welches diese Dateien hortet und für alle Services verfügbar macht.

Um dieses Problem zu beheben, existiert ein Git-Projekt, in dem alle gemeinsamen Dateien gespeichert sind. Dieses wird als Git-Submodule[20] in die jeweiligen Service-Projekte eingebunden.

Darin befinden sich einerseits die Models für die Datenbankobjekte. Andererseits werden die Feign-Client-Interfaces in den gemeinsamen Dateien verwaltet. Diese können mittels Dependency Injection im Projekt verwendet werden.

Für das Error-Handling der einzelnen Feign-Requests ist ein FeignErrorDecoder im Einsatz. Er nimmt die FeignRequests unter die Lupe. Er prüft, ob die Response einen Error beinhaltet und führt das Errorhandling aus.

Ebenfalls ausgelagert ist ein RequestInterceptor, welcher definiert, wie der Header bei jedem Request der von einem Microservice zu einem anderen Microservice geschickt wird, auszusehen hat.

7.2.4 Abhängigkeiten

Buildtime Als Build-Management-Tool und zur Auflösung vom Abhängigkeiten zur Buildtime wird Maven verwendet.

Runtime Um zur Laufzeit die Abhängigkeiten zu anderen Services und Komponenten verfügbar zu machen, stellt das Spring Framework eine interne Dependency Injection Lösung an. Repositories, Services und Components werden bei Bedarf mit der @autowired Annotation versehen, und können sogleich verwendet werden. Ausserdem vereinfacht diese Dependency Injection das Testen einzelner Services, weil sie sehr leicht gemockt oder gefaked werden können.

7.2.5 Microservices Authentifizierung

Zwischen den einzelnen Microservices wird eine Pre-Shared Secret Authentifizierung verwendet, um sicher zu stellen, dass keine unauthentifizierten Requests von den einzelnen Services verarbeitet werden.

Umgesetzt wurde dies mit einer Eigenimplementierung des WebSecurityConfigurerAdapter von Spring Security. Bei jedem Request wird mit Hilfe eines ApiKeyAuthFilters, welcher die abstrakte Klasse

AbstractPreAuthenticatedProcessingFilter implentiert, geprüft, welcher Api-Key im Request Header mitgesendet wurde und ob er mit dem eigenen Api-Key übereinstimmt. Sollte dies nicht der Fall sein, erhält der Caller eine Response mit den Http-Status Code 403 Forbidden.

Als Key-Header wird "**API-KEY**" verwendet, als Value das jeweilige Secret des Microservices, an den geschickt wird. Das Secret ist ein zufällig generierter SHA-256 String mit einer Länge von 64 Byte.

Die benötigten Keys liegen bei den einzelnen Projekten in der application.properties Datei, wo sie beliebig konfiguriert werden können.

7.3 Webservice

Der Webservice ist die zentrale Schnittstelle für den Benutzer der API resp. des User Interfaces auf die Plattform. Es verarbeitet alle Anfragen, führt einfache CRUD-Requests direkt aus und koordiniert entsprechende Aufgaben an die weiteren Services.

7.3.1 Authentifizierung

Um Zugriff auf den Webservice zu erhalten, muss sich ein Benutzer des User Interface mit Accountname und Password, welches von einem Administrator erstellt wurde, einloggen. Als Benutzer des APIs wird pro Client Application, welche erstellt wird, ein JSON Web Token (JWT) vorgeneriert.

Dieses JWT Token setzt sich wie folgt zusammen:

eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbiIsImlzcyI6IkxvcmF3YW5NYW5hZ2VtZW50UGxhdGZvcm0iLCJleHAiOjE1NjA0MTY5OTQsImlhdCI6MTU1OTgxMjE5NH0.ymTy-2GhYKOU4TgjPkn1irzAZUUFAxJWtEiOGT19qJ8qxGwZ2r7CoDKHNAuq2gdGyPT3hKrbOOX1Ybe4i1D3qQ

Braun: Algorithmus (HMAC using SHA-512)
Blau:

- Sub (Subject): Account oder Client Application Name
- Iss (Issuer): LorawanManagementPlattform
- Exp (Expiration Date): In application.properties gesetzt
- Iat (Issued At)

Grün:

- Signatur (HMAC using SHA-512)
  - Header (Base64)
  - Payload (Base64)
  - Secret: In application.properties gesetzt

Der Webservice nutzt denn OncePerRequestFilter von Spring Web. Dieser prüft das JWT Token bei jedem Request auf die Gültigkeit bezüglich Ablaufdatum und ob der Subject sich in den Account resp. Client Application Tabellen befindet.

7.3.2 API

Die API beinhaltet 45 Endpunkte und ist mit Swagger dokumentiert. Die Beschreibung zum Webservice-API ist im Anhang oder Online zu finden.

7.4 Swisscom Provider Service

Der Swisscom Provider Service kommuniziert mit Hilfe des Thingpark APIs mit dem Swisscom LPN. Auf das API wird mit Hilfe von HTTPS-Requests kommuniziert.

Die URL-Basis aller Requests auf die API lautet wie folgt:

https://dx-api.thingpark.com/core/latest/api

In diesem Kapitel wird erklärt, welche Schnittstellen der Provider Service vom Thingpark API verwendet und wie die Requests aussehen müssen, resp. welche Antworten zu erwarten sind.

7.4.1 Authentifizierung

Die API besteht darauf, dass ein Bearer Token mitgesendet werden muss. Dieses Token erhält man mit einer Kombination aus Benutzername (Client ID) und Password (Client Secret). Diese Eigenschaften müssen beim Netzwerk Provider auf der Plattform hinterlegt werden. Der Endpunkt auf der API befindet sich unter

https://dx-api.thingpark.com/core/latest/api/oauth/token

Ein Request sieht dabei wie folgt aus:

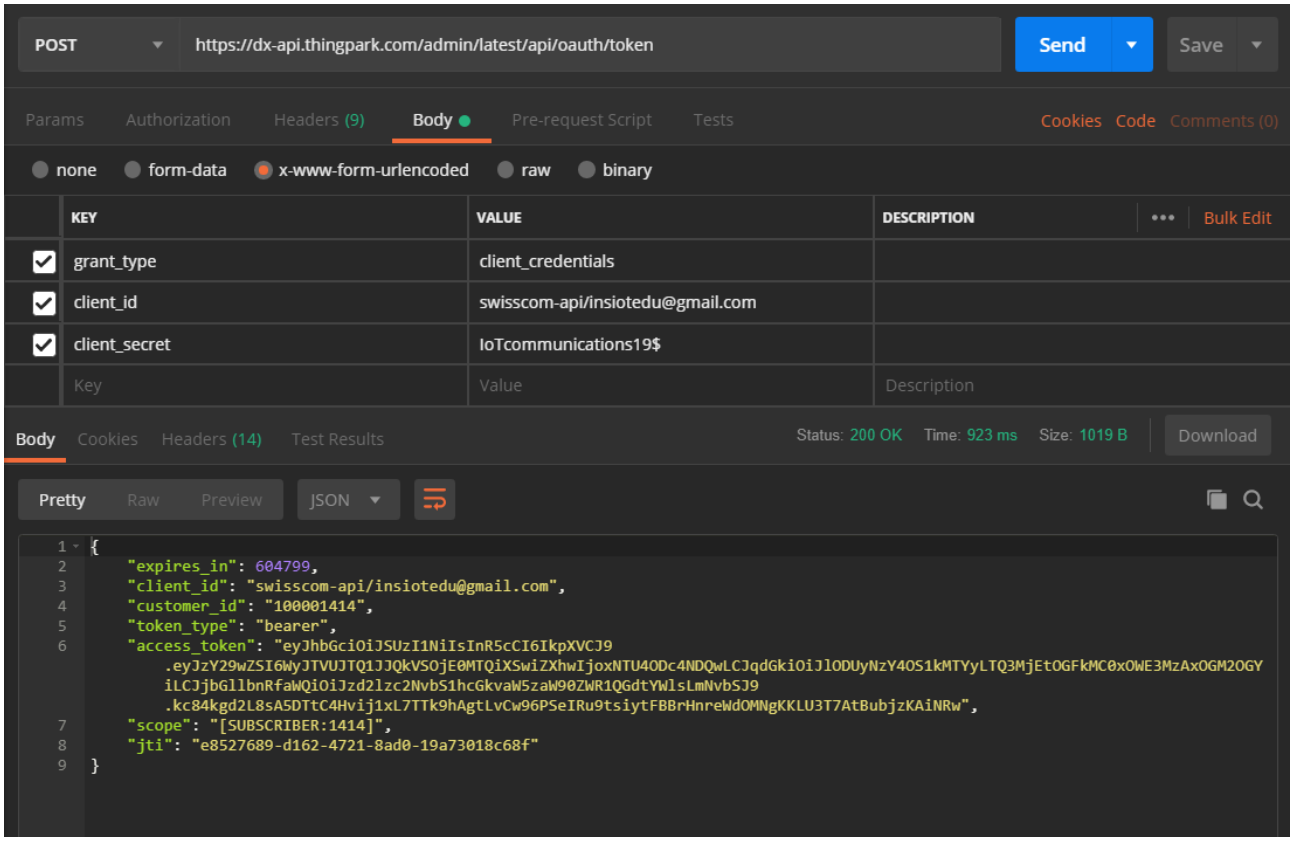


Abb. 34: POST Request Bearer Token

Bei erfolgreicher Authentifizierung erhält man eine Response mit dem Status Code 200 und dem Bearer Token, welches sich im Body unter dem Attribut "access_token" befindet.

Damit nicht bei jeder neuen Verbindung mit der API ein zusätzlicher Request für das Bearer Token gemacht werden muss, wird das Token auf dem Service zwischengespeichert. So kann es wiederverwendet werden.

Das Token wird dann bei jedem Request als Header Parameter mitgesendet:

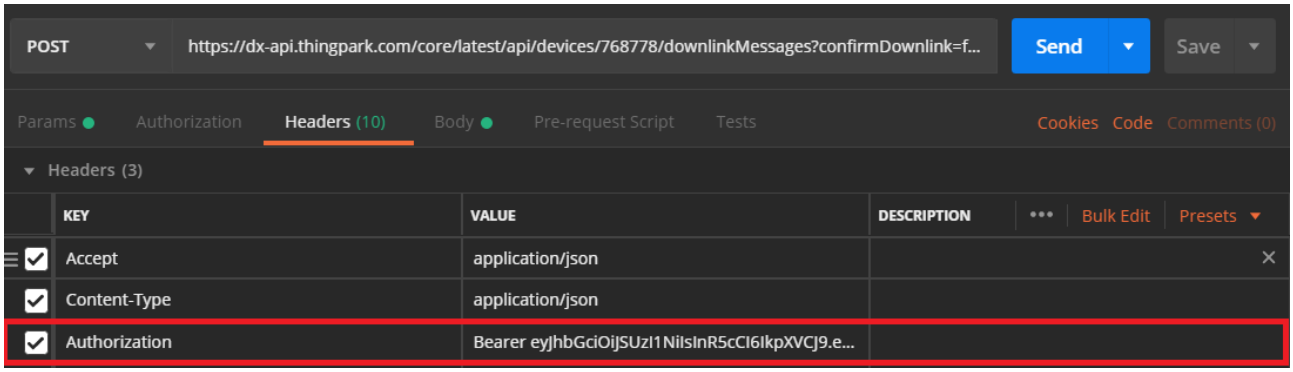


Abb. 35: Header Parameter Authorization

7.4.2 Schnittstellen

Der Swisscom Provider Service ist in der Lage Geräte zu registrieren, Geräte zu löschen und Downlinks zu senden. Die umgesetzten Requests auf die Thingpark API sehen dann jeweils wie folgt aus:

Gerät registrieren (POST Request)

Ein Gerät kann wahlweise via Over The Air Activiation (OTAA) oder Activiation By Personalization (ABP) bei der Swisscom registriert werden. Je nach Auswahl werden unterschiedliche Informationen des Geräts benötigt. Diese Eigenschaften werden vom Benutzer der Plattform beim Erstellen eines Geräts hinterlegt. Es wird ein POST-Request getätigt.

URL: **https://dx-api.thingpark.com/core/latest/api/devices/**

Dabei müssen folgende Geräteeigenschaften mitgesendet werden:

Eigenschaft	**Beschreibung**	**Bemerkungen**
name	Gerätename	
devEUI	Device EUI	Hexadezimaler Wert
activationType	Registrierungstyp	OTAA oder ABP
connectivityPlanId	Abonnementtyp	
deviceProfileId	Device Klassen (A oder C)	Wird mit den Swisscom-spezifischen Codes gemappt (A -> LORA/GenericA.1; C -> LORA/GenericC.1)
routingProfileId	Routing Profil des Geräts	Wird von Benutzer hinterlegt
applicationEUI	Application EUI	**NUR OTAA**
applicationKey	Application Key	**NUR OTAA**
networkAddress	Netzwerkgeräteadresse	**NUR ABP** - Hexadezimaler Wert
networkSessionKey	Netzwerksessionkey	**NUR ABP**
applicationSessionKey	Application Session Key	**NUR ABP**
sourcePorts	Ports, welche den Application Session Key verwenden	**NUR ABP** - Default "*"

Tab. 3: Swisscom Registrierung Geräteeigenschaften

Bei der Swisscom gibt es unterschiedliche Abonnement-Typen. Ein Abonnement (von der Swisscom auch Connectivity Plan genannt) definiert, wie viele Nachrichten (Up- bzw. Downlinks) pro Tag gesendet werden können. Diese Abonnements werden in Absprache mit dem Industriepartner für den Rahmen dieser Arbeit fix bei der Plattform hinterlegt.

Ingesamt sehen die jeweiligen Requests (OTAA- sowie ABP Registration) wie folgt aus:

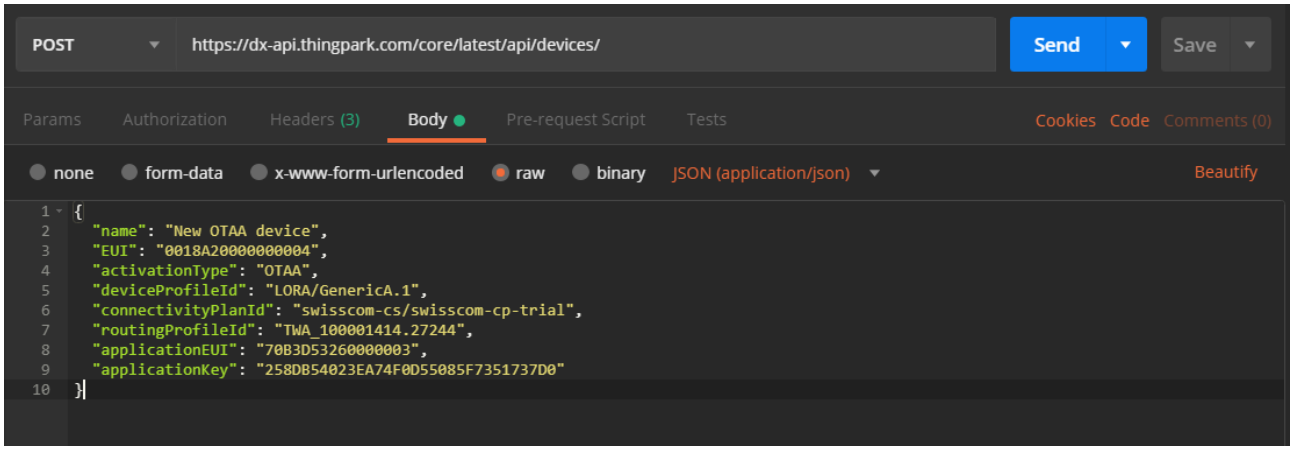


Abb. 36: Swisscom POST Request: OTAA Registration

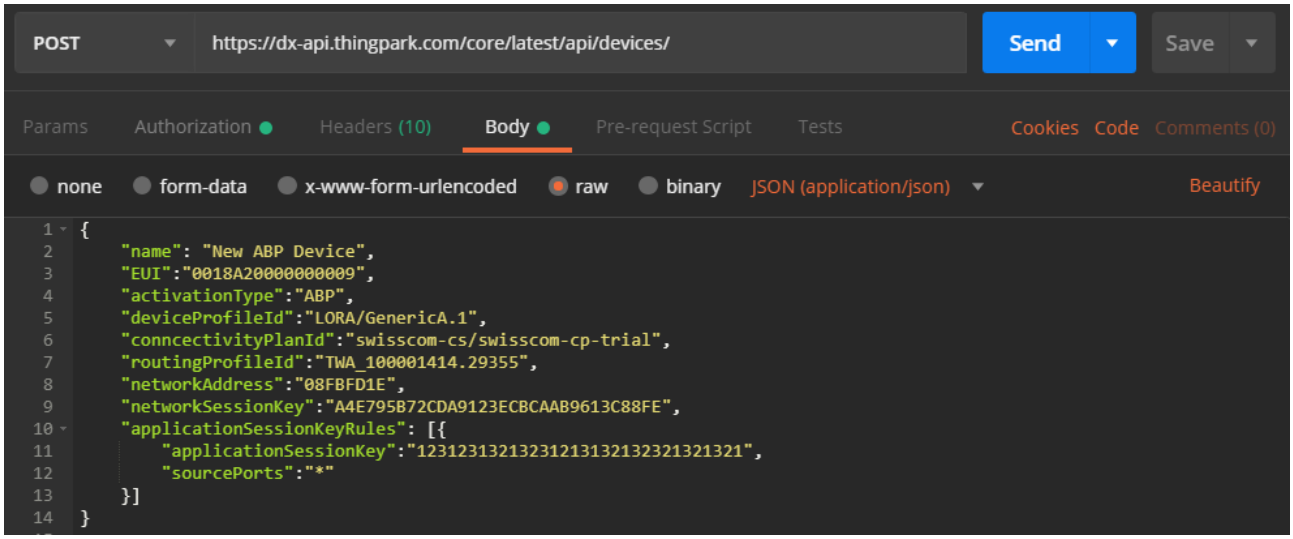


Abb. 37: Swisscom POST Request: ABP Registration

Bei einer erfolgreichen Registration erhält man eine Response mit dem Status Code 201 und der internen Swisscom Referenznummer für dieses Gerät. Diese Referenznummer muss bei der Plattform hinterlegt werden, weil das Senden der Downlinks bzw. das Löschen eines Geräts nur mit Hilfe dieser Referenznummer erfolgen kann.

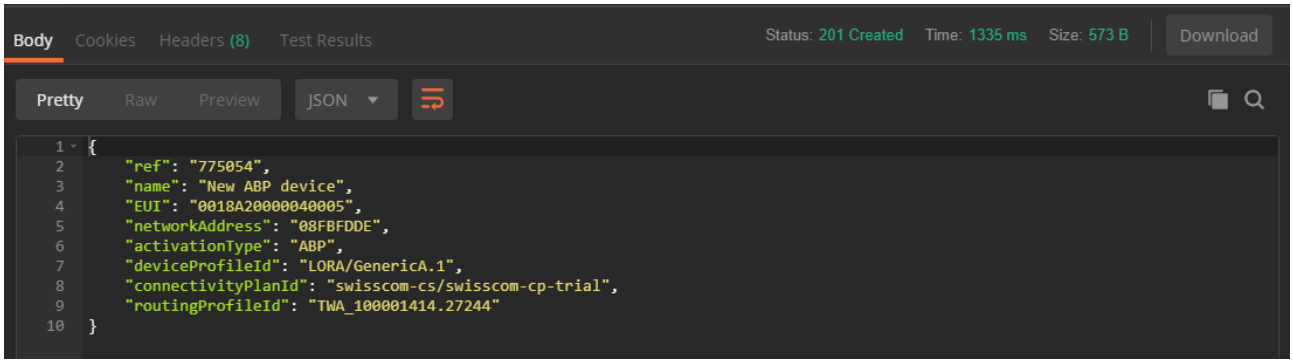


Abb. 38: Swisscom Registration: Success Response

Bei einem Fehlerfall schickt die Thingpark API immer einen Status Code 400 und unter dem Attribut "message" erhält man die Fehlerinformation.

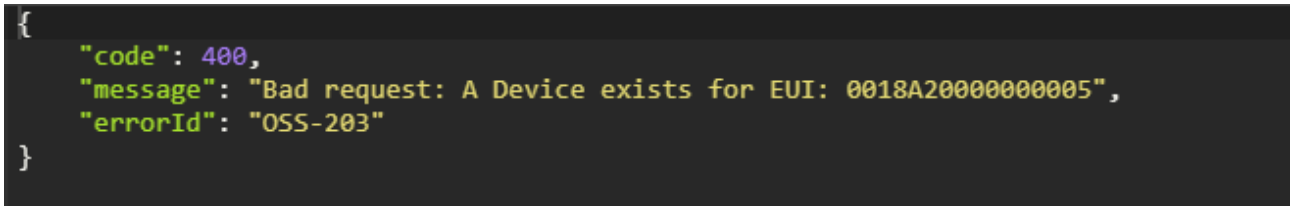


Abb. 39: Swisscom Registration: Error Response

Gerät löschen/deregistrieren (DELETE Request)

Mit Hilfe der Swisscom Referenznummer, welche man bei der Registration eines Geräts erhalten hat, kann ein Gerät mit einem DELETE Request auf den Endpunkt **https://dx-api.thingpark.com/core/latest/api/devices/{device_ref}** wieder gelöscht werden.

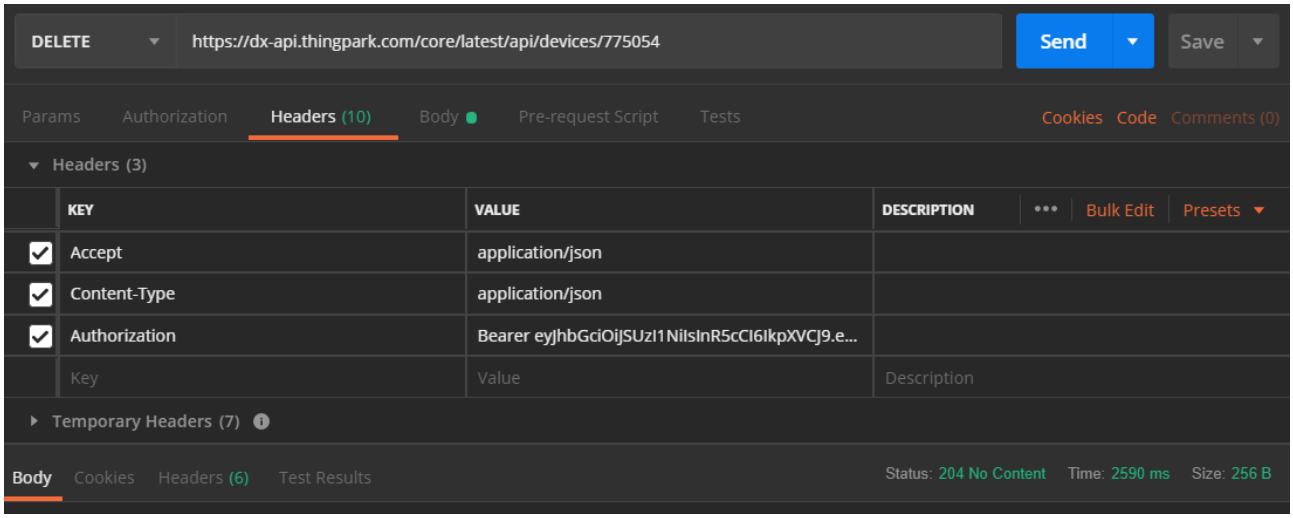


Abb. 40: Gerät löschen: Success Response

Ein erfolgreicher Request wird mit einem Response Status Code 204 gekennzeichnet. Andererseits wird bei einem Fehlerfall ein Status Code 400 und die entsprechende Fehlermeldung wie bei der Registration im Body mitgeschickt.

Downlink senden (POST Request)

Für einen Downlink wird wiederum die Swisscom Referenznummer benötigt, um die URL des Endpunktes zu generieren: **https://dx-api.thingpark.com/core/latest/api/devices/{device_ref}**

Es wird ein POST Request mit folgenden Query-Parametern gemacht:

Eigenschaft	Beschreibung	Bemerkungen
confirmDownlink	Confirmation des Downlinks	Wird vom Plattformbenutzer angegeben
flushDownlinkQueue	Initiiert einen Flush der Queue bevor der Downlink hinzugefügt wird	Default false wird verwendet

Tab. 4: Swisscom Downlink Request: Queryparameter

Als Body erwartet die Schnittstelle folgende Eigenschaften:

Eigenschaft	Beschreibung	Bemerkungen
payloadHex	Payload des Downlinks	Hexadezimaler Wert
targetPorts	Ports an die der Downlink gesendet werden soll	Kommaseparierter String

Tab. 5: Swisscom Downlink Request: Body

Hier ein Beispielrequest:

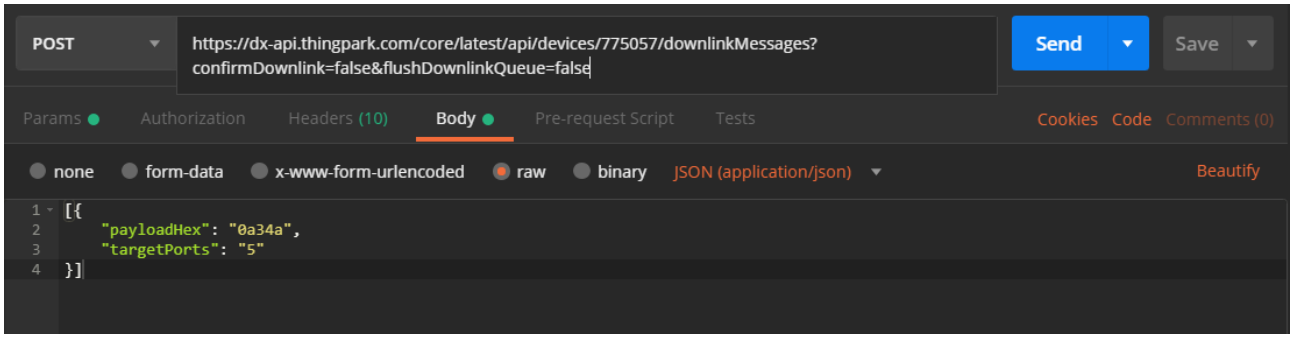


Abb. 41: Swisscom Downlink Request

Als Response sendet die Thingpark API jeweils immer den Status Code 201. Um herauszufinden, ob ein Downlink erfolgreich an das Gerät übermittelt werden kann, findet sich im Response Body versteckt der Status des gesendeten Downlinks. Der Status ist dabei einer der folgenden Werte:

Status	**Beschreibung**	**Bemerkungen**
QUEUED	Downlink erfolgreich gesendet und in Queue eingeordnet	
ERROR_MSG	Fehler beim Senden	Das Feld error_description beinhaltet die Fehlermeldung

Tab. 6: Swisscom Downlink Response: Body

Bei einem Fehlerfall schickt die Thingpark API immer einen Status Code 400 und unter dem Attribut "message" erhält man die Fehlerinformation.

Uplinks (Von Swisscom zu LMP)

Um Uplinks von der Swisscom an unsere Plattform erhalten zu können, mussten folgende Einstellungen auf dem ThingPark Wireless Management Tool getätigt werden:

1.) Application Server hinterlegen Unsere Plattform muss bei ThingPark erst als aktive Applikation hinterlegt werden. Wichtig ist, dass als Content Type der Daten JSON hinterlegt ist, weil nur dieser Typ unterstützt wird. Zudem wird bei den Routes angegeben, welche URL auf der Seite des Application Server den HTTP-Post der Uplinks empfangen kann.

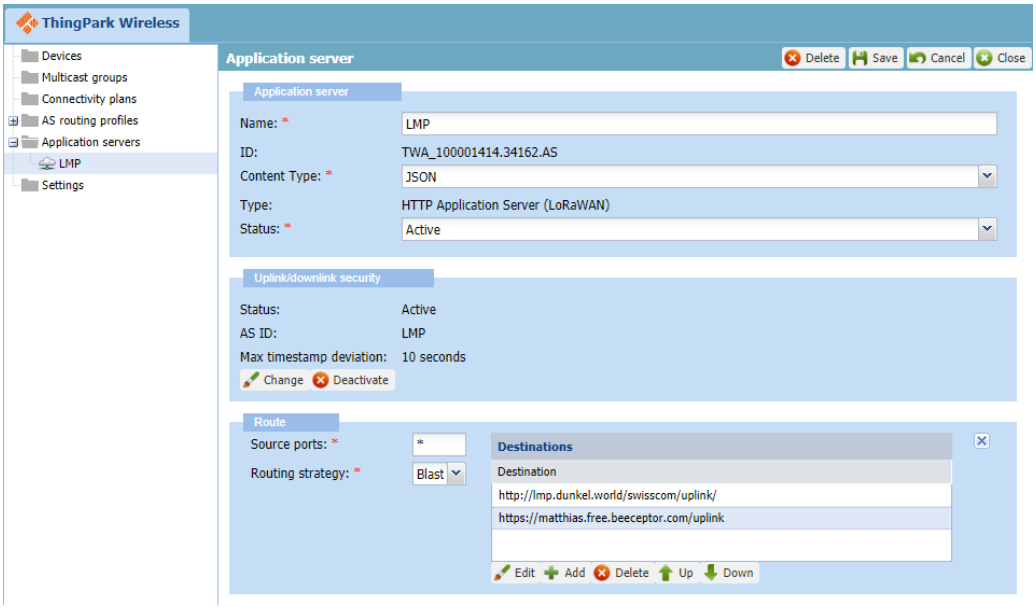


Abb. 42: ThingPark: Application Server hinterlegen

2.) Application Service Routing Profile erstellen Damit einem Device nun ein Application Server zugewiesen werden kann, benötigt es ein Application Service Routing Profile. Dieses Profil hinterlegt man bei der Registrierung eines Geräts. Als Destination wird der zuvor erstellte Application Server ausgewählt.

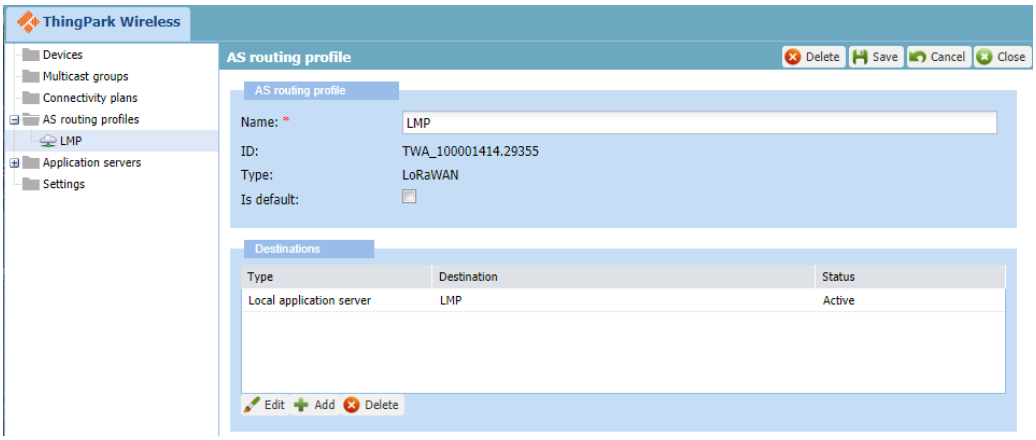


Abb. 43: ThingPark: AS Routing Profile hinterlegen

Nun werden Uplinks an den definierten Endpunkt unserer Plattform gesendet.

Authentifizierung

Die Gewährleistung der Authentifizierung von Swisscom Paketen zu unserer Plattform erweist sich ein wenig komplexer. Sie wird von der Swisscom bereits vorgegeben und geschieht in folgenden Schritten:

1.) Sicherheit bei ThingPark aktivieren Um die Authentifizierung bei der Swisscom hinterlegen zu können, benötigt man für die Applikation (in unserem Fall LMP) ein selbstgewähltes Secret (128bit hexadezimal), welches AS-Key genannt wird, und eine Applikation Server-ID (AS ID). Das Secret muss für die Tokenvalidierung bei der Plattform hinterlegt werden. Ausserdem kann mit der *Max. time deviation* definiert werden, wie viel Zeit zwischen Tokengenerierung auf dem Netzwerkserver der Swisscom und der Validation auf der Client Applikation (in unserem Fall die Plattform) maximal vergehen darf.

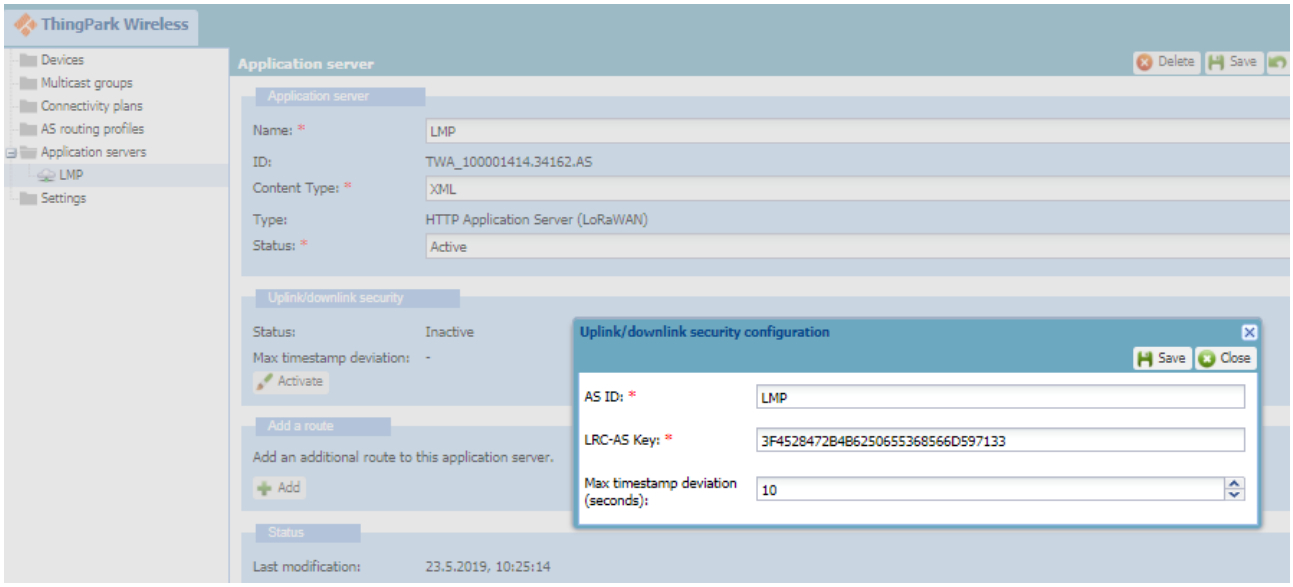


Abb. 44: Thingpark: Aktivierung der Security für Uplinks

2.) Uplink Query-Parameter zusammensetzen Als Query-Parameter erhält die Plattform vom Uplink der Swisscom diverse Eigenschaften, wobei die folgenden zu einem String mit einem Ampersand (&) als Separator zusammengesetzt werden müssen:

Eigenschaft	**Beschreibung**	**Bemerkungen**
AS_ID	Application Service ID	Nicht gesetzt, wenn Punkt 1) nicht ausgeführt wurde und somit keine Authentifizierung eingerichtet worden ist.
LrnDevEui	Device EUI	Hexadezimaler Wert
LrnFPort	Portnummer	
LrnInfos	Service Profile Name	
Time	ISO Zeitstempel	Nicht gesetzt, wenn Punkt 1. nicht ausgeführt wurde und somit keine Authentifizierung eingerichtet worden ist.
Token	Token generiert von Swisscom	Nicht gesetzt, wenn Punkt 1. nicht ausgeführt wurde und somit keine Authentifizierung eingerichtet worden ist.

Tab. 7: Swisscom Uplink: Query Parameter

Beispiel: *LrnDevEui=000000000F1D8693&LrnFPort=108&LrnInfos=LMP&AS_ID=LMP&Time=2018-05-25T12:00:00.000+02:00*

3.) Uplink Body-Request zusammensetzen Im Request-Body enthält der Uplink diverse Elemente, wobei folgende wiederum zusammengesetzt werden müssen (ohne Separator):

Eigenschaft	**Beschreibung**	**Bemerkungen**
CustomerID	Kundennummer vom Provisioning	
DevEUI	Device EUI	Hexadezimaler Wert
FPort	Portnummer	
FCntUp	Uplink Counter	
payload_hex	Uplink Payload Daten	

Tab. 8: Swisscom Uplink: Request-Body Elemente

Beispiel: *100000507000000000F1D869310870110027bd00*

4.) Hashing der Daten Um das Token generieren zu können, müssen die Daten mit dem SHA-256 Algorithmus gehashed werden. Dabei werden die zusammengesetzten Strings der Query-Parameter und Body-Elemente aus den vorherigen beiden Schritten, mit dem hinterlegten AS-Key aus Punkt 1) zu einem String verbunden und prozessiert:

SHA-256 (<query_parameter><body_elements><AS-Key>)

Beispiel: *SHA-256(LrnDevEui=000000000F1D8693&LrnFPort=108&LrnInfos=LMP&AS_ID=LMP&Time=2018-05-25T12:00:00.000+02:00100000507000000000F1D869310870110027bd00 3F4528472B4B6250655368566D597133)*

=> 37DB43C3CB2579E08790F99CE0239C2336C9653173F465CFCB01705A8FDDA2A5

5.) Vergleich der Tokens In den Query-Parametern des Uplink Pakets befindet sich der Token, welcher von der Swisscom berechnet worden ist. Im Gegenzug haben wir unseren eigenen Token berechnet. Sind diese beiden Token identisch, ist der Uplink erfolgreich authentifiziert worden.

7.5 Loriot Provider Service

Der Loriot Provider Service kommuniziert mit dem von Loriot zur Verfügung gestellten API und wickelt das Gerätemanagement und die Down- bzw. Uplinkverarbeitung ab. Bei der Loriot-API gibt es, je nach Region in welcher man sich befindet, unterschiedliche Server-URLs. Da wir uns in Europa befinden, wird ein europäischer Server bevorzugt. Die Basis-URL ist in den Network Provider Einstellungen hinterlegt und kann bei Bedarf auch verändert werden.

Basis-URL: https://eu1.loriot.io/

In diesem Kapitel wird erklärt, welche Schnittstellen der Provider Service von der Loriot-API verwendet und wie die Requests aussehen müssen, resp. welche Antworten zu erwarten sind.

7.5.1 Authentifizierung

Um die benötigten Funktionalitäten der Loriot-API verwenden zu können, müssen zwei verschiedene Tokens generiert werden. Einerseits gibt es einen **API-Key**, welchen man für das Management der Geräte verwenden muss. Dieser kann auf dem Benutzerprofil im Frontend von Loriot erstellt werden:

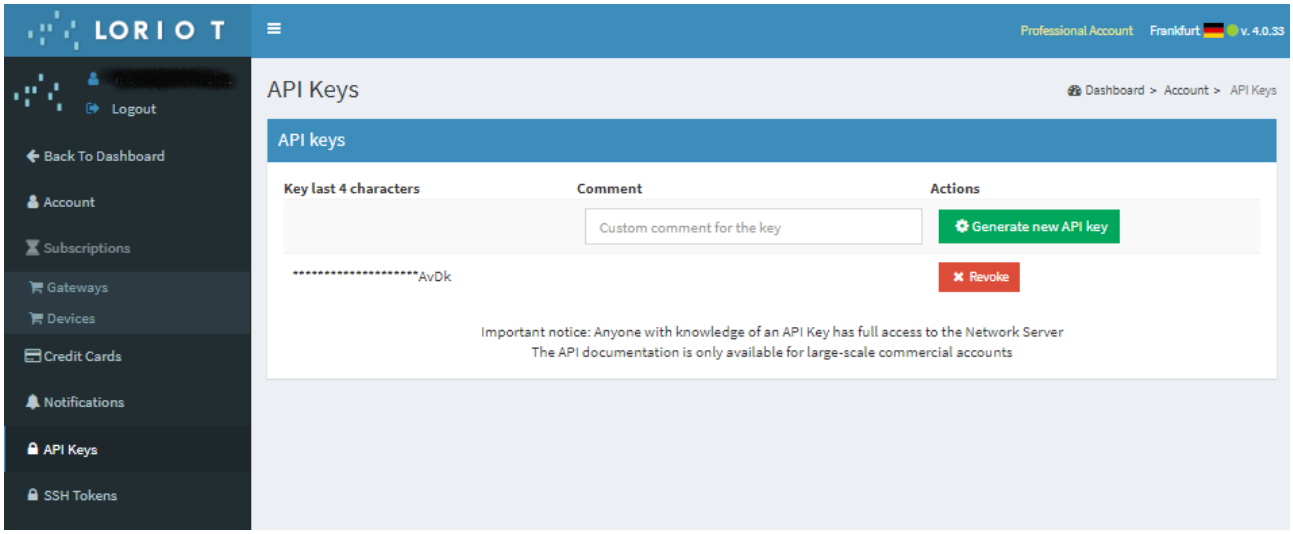


Abb. 45: Loriot UI: API Key erstellen

Zusätzlich benötigt man pro registrierter Applikation, die man bei Loriot hinterlegt hat, ein **Application Token**, um Downlinks an die Geräte senden zu können. Diese können unter der registrierten Applikation im Loriot-UI generiert werden:

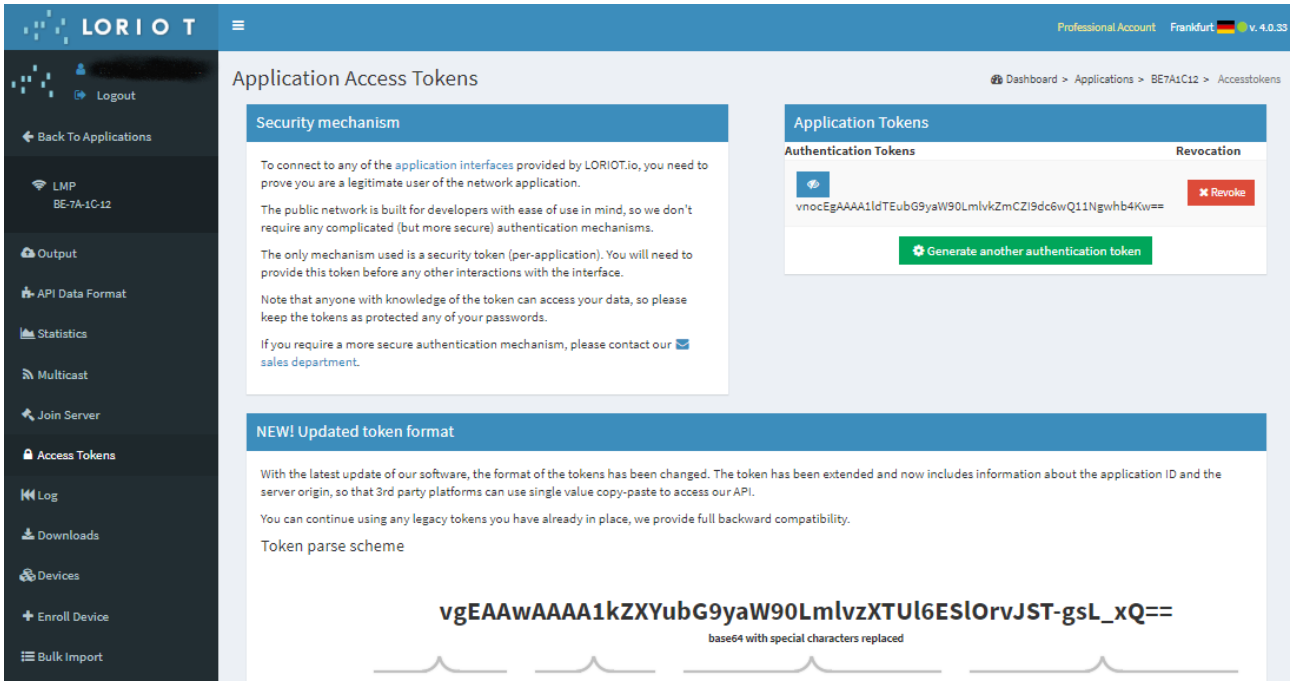


Abb. 46: Loriot UI: API Key erstellen

Beide Tokens müssen auf der Plattform beim Loriot Network-Provider hinterlegt werden.

7.5.2 Schnittstellen

Der Loriot Provider Service ist gleich wie der Swisscom Provider Service in der Lage, Geräte zu registrieren, Geräte zu löschen und Downlinks zu senden. Die folgenden Schnittstellen des Loriot-API werden dabei wie folgt benutzt:

Device registrieren

Um ein Device registrieren zu können, muss beim Loriot-Management Tool die entsprechende Applikation hinzugefügt werden. Diese Applikation erhält dann eine AppId, welche auf der LoRaWAN Management Plattform auf dem Network Provider hinterlegt werden muss.

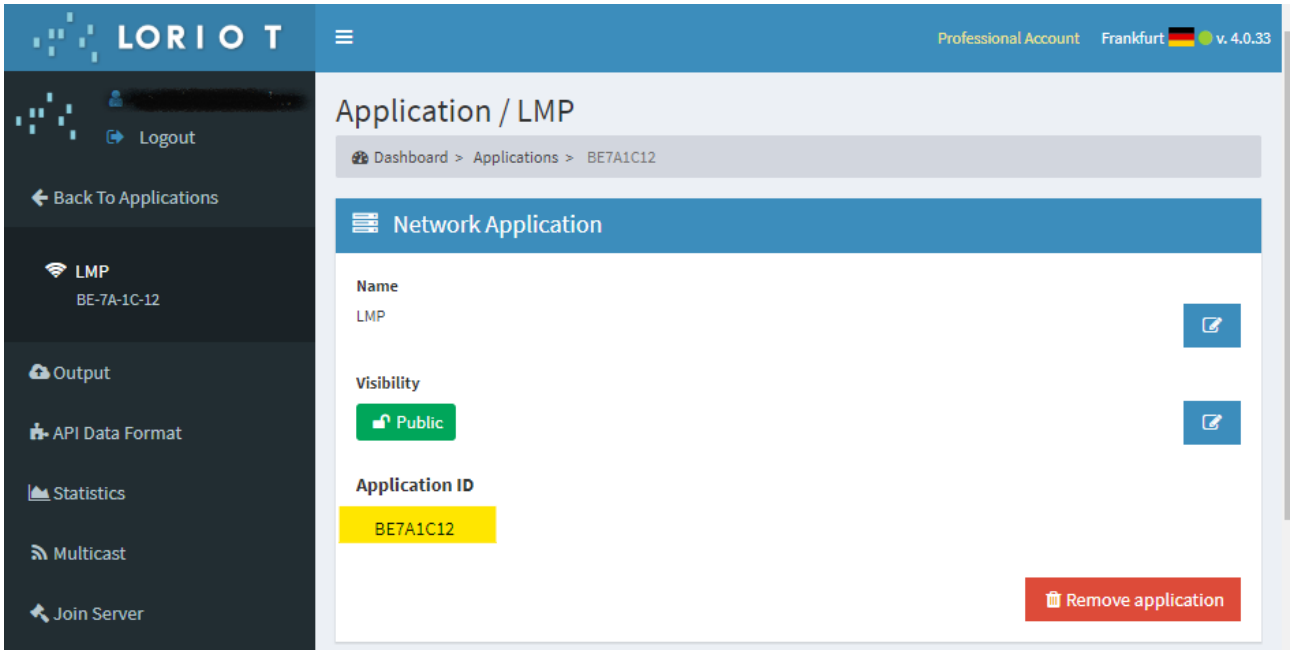


Abb. 47: Loriot UI: Application ID (App ID)

Mit Hilfe dieser App ID kann der Pfad zum Endpunkt erstellt werden, an welchen die Request gesendet werden können:

https://eu1.loriot.io/1/nwk/app/{app_id}/devices

Der Request wird per POST-Request durchgeführt, welcher als Header Parameter den API-Key als Authorization Bearer Token mitsendet.

Die Body Parameter sehen wie folgt aus:

Eigenschaft	Beschreibung	Bemerkungen
title	Gerätename	
deveui	Device EUI	Hexadezimaler Wert
devclass	Device Klassen (A oder C)	
appeui	Application EUI	**NUR OTAA**
appkey	Application Key	**NUR OTAA**
devaddr	Netzwerkgeräteadresse	**NUR ABP** - Hexadezimaler Wert
nwkskey	Netzwerksessionkey	**NUR ABP**
seqno	Initiale Uplink Sequenz Nummer	**NUR ABP** - Default 0
seqdn	Initiale Downlink Sequenz Nummer	**NUR ABP** - Default 0

Tab. 9: Loriot Registrierung Geräteeigenschaften

Somit erhalten wir zwei unterschiedlich aussehende POST-Requests für OTAA sowie ABP Registration:

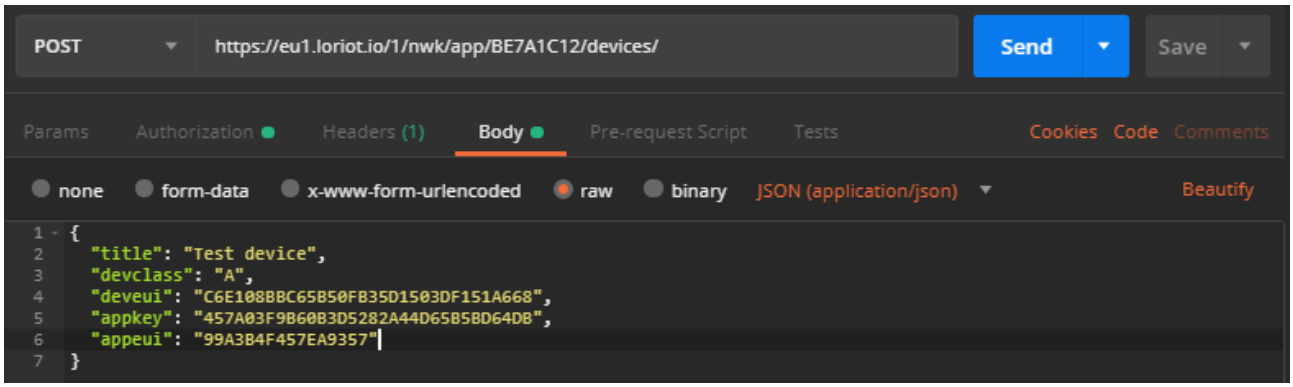


Abb. 48: Loriot API: OTAA Registration

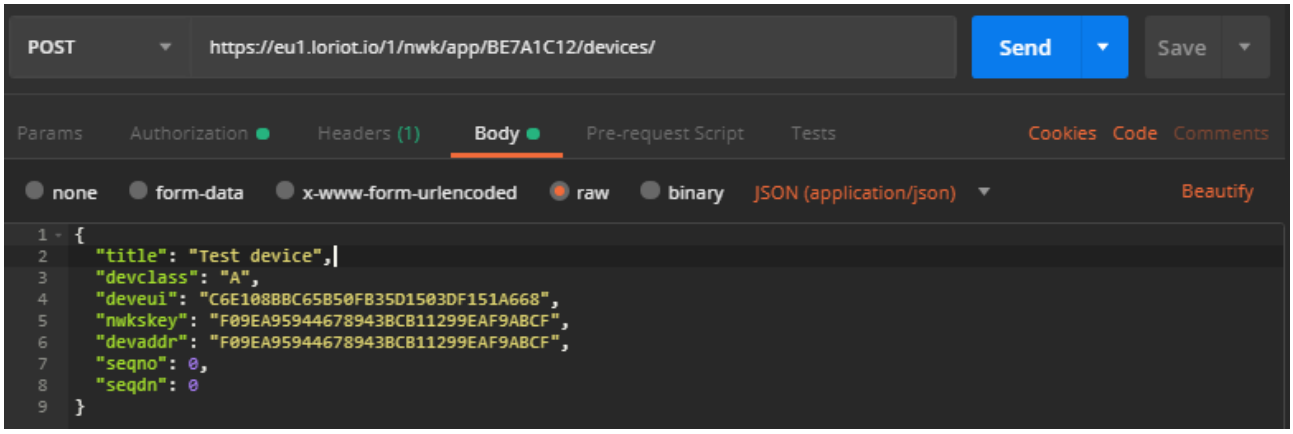


Abb. 49: Loriot API: ABP Registration

Die Responsestatus des Service sind folgende:

Status Code	**Beschreibung**	**Response Body**
200	Success	Erstelltes Gerät als Objekt
400	Bad Request	Error Message
401	Unauthorized	
403	Forbidden	

Tab. 10: Loriot-API Registrierung Responses

Device löschen (DELETE Request)

Um ein Device deregistrieren zu können, muss es gelöscht werden. Dies wird wiederum auf dem Device Management Endpunkt, mit einem **DELETE** Request, gemacht:

https://eu1.loriot.io/1/nwk/app/{app_id}/device/{deveui}

Wiederum wird im Header der API Key als Bearer Token mitgegeben.

Die Responsestatus des Service sind folgende:

Status Code	Beschreibung	Response Body
200	Success	
400	Bad Request	Error Message
401	Unauthorized	
403	Forbidden	

Tab. 11: Loriot-API Registrierung Responses

Downlink senden

Um einen Downlink zu senden wird im Gegensatz zum Devicemanagement ein unterschiedliches Interface benötigt:

https://eu1.loriot.io/1/rest

Auch die Authentifizierung wird unterschiedlich gemanaged. Anstatt eines API-Keys wird eine generiertes Application Token mitgesendet. Der Request wird als POST-Request getätigt.

Folgende Parameter werden im Body mitgesendet:

Eigenschaft	Beschreibung	Bemerkungen
cmd	Message-Typ	tx für Downlink, rx für Uplink
appid	Application ID	
EUI	Device EUI	
port	Einzelner Port	Integer
confirmed	Confirmation des Downlink	Boolean (true, false)
data	Daten des Downlinks	Hexadezimaler Wert

Tab. 12: Loriot Registrierung Geräteeigenschaften

Der Beispielrequest sieht wie folgt aus:

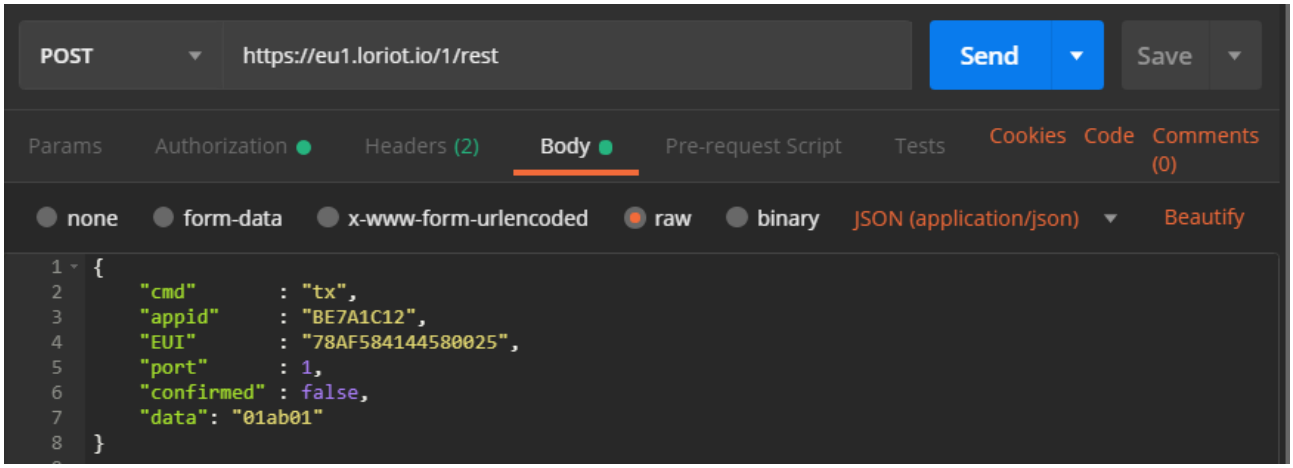


Abb. 50: Loriot Downlink Request

Als Responsestatus können folgende Status vorkommen:

Status Code	**Beschreibung**	**Response Body**
200	Success	Daten enqueued
400	Bad Request	Error Message
401	Unauthorized	
403	Forbidden	

Tab. 13: Loriot-API Registrierung Responses

Bei einer Success-Response sieht der Body wie folgt aus:

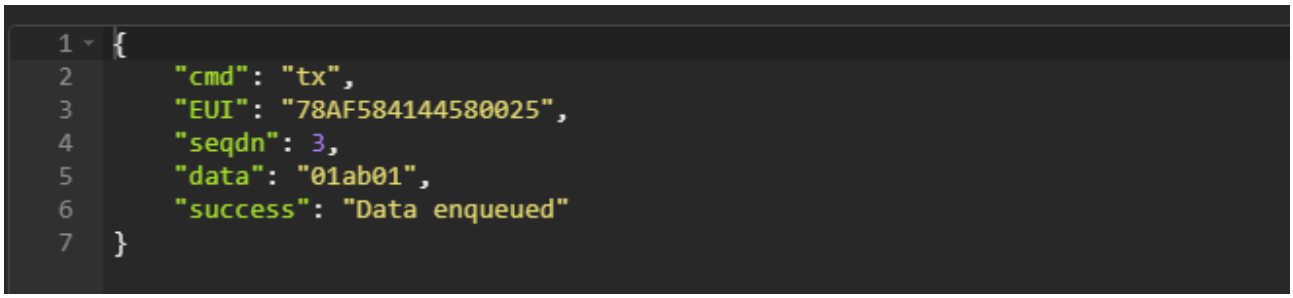


Abb. 51: Loriot Downlink Success

Ein Bad Request-Response enthält folgenden Body:

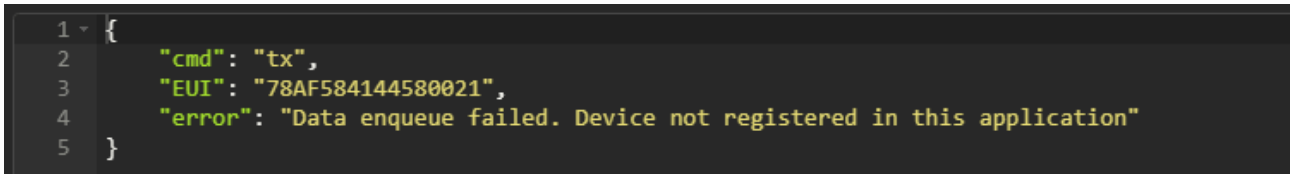


Abb. 52: Loriot Downlink Success

7.6 Parsing Service

Der Parsing Service ist dahingehend speziell, da er nicht mit dem Spring Boot Framework entwickelt wurde, noch nicht mal mit Java, sondern mit Node.js[7]. Dies, da wir das Open Source Projekt "Docker Python Sandbox"[8] übernehmen und unseren Bedürfnissen anpassen konnten.

Der Service nutzt das Express Framework und stellt damit einen Endpunkt bereit, über den man Python Code senden kann, welcher dann ausgeführt wird. Als Antwort erhält man den Output.

Welche Python Version verwendet wird, ist per Environmentvariable bestimmbar. Der Service überwacht den Thread und bricht ihn ab, falls er die vorgegebene Ausführungszeit überschreitet.

7.7 Alerting Service

Der Alerting Service erhält die Uplink-ID, anhand welcher das dazugehörige Gerät ermittelt wird. Dann wird rekursiv jede Gruppe des Gerätes überprüft. Hat diese Gruppe eine Alert-Konfiguration zugewiesen, wird dieser ausgeführt.

Dies geschieht, indem der Code sowie der State und die Uplink Daten an den Parsing service geschickt werden, um den Alert-Code auszuführen.

Zuvor wird der Code, welcher vom Benutzer eingegeben wurde, mit weiterem Code angereichert. Dies, damit der Benutzer in seinem Code vorgefertigte Funktionen benutzen kann, was das Schreiben der Alerts deutich einfacher macht.

Falls das Resulat einen Alert ausgibt, wird dieser in die Datenbank geschrieben.

7.8 Post Output Service

Dieser Service verschickt HTTP/POST Nachrichten an die vom Benutzer definierte URL. Diese Nachrichten werden JSON codiert.

Grundsätzlich ist dieser Service in der Lage jedes beliebige Datenpaket zu senden. Praktisch wird er aber bis jetzt nur verwendet, um Uplinks zu versenden. Die Daten eines solchen Datenpaketes sehen wiefolgt aus:

```
{
```

[7] https://nodejs.org/
[8] https://github.com/christophetd/docker-python-sandbox

```
    "port": 108,
    "framecount": 12,
    "sf": 12,
    "rssi": -116.45,
    "snr": -0.5,
    "data": "40040e01f60f",
    "devEUI": "78AF584144580029",
    "date": 1560078177447,
    "secure": true,
    "temperature": 270,
    "battery": 4086,
    "distance": 1088
}
```


Hier die Beschreibung dieser Felder:

Feld	**Typ**	**Beschreibung**
port	int	Der LoRaWAN Port
framecount	int	Der Framecount der LoRaWAN Paketes
sf	int	Der Spreading Factor
rssi	float	Der RSSI
snr	float	Der SNR
data	String	Die ungeparsten Daten
devEUI	String	Die DevEUI des Absender Gerätes
date	int	Absendezeit als Unix Timestamp
secure	boolean	Gibt an, ob der Absender dem System bekannt ist, und validiert werden konnte
...		Es folgen beliebig viele Felder, die vom hinterlegten Parser definiert werden.

Tab. 14: Beschreibung der JSON-Felder

7.9 Influx Output Service

Dieser Service ist dafür zuständig, Daten in einen Influx Datenbank zu schreiben. Dazu müssen vom Benutzer die Verbindungsdaten sowie Benutzername und Passwort hinterlegt werden. Die Daten werden in die, auch vom Benutzer, spezifizierte Tabelle geschrieben.

Auch dieser Service ist in der Lage jedes beliebige Objekt an die Datenbank zu senden. Allerdings muss das Objekt eine flache Struktur haben.

Zum jetzigen Stand werden auch hier nur die Uplinks versendet. Hier eine Beschreibung dieser Felder:

Feld	**Typ**	**Beschreibung**
port	int	Der LoRaWAN Port
framecount	int	Der Framecount der LoRaWAN Paketes
sf	int	Der Spreading Factor
rssi	float	Der RSSI
snr	float	Der SNR
data	String	Die ungeparsten Daten
devEUI	String	Die DevEUI des Absender Gerätes
date	int	Absendezeit als Unix Timestamp
secure	boolean	Gibt an, ob der Absender dem System bekannt ist, und validiert werden konnte
...		Es folgen beliebig viele Felder, die vom hinterlegten Parser definiert werden.

Tab. 15: Beschreibung der Felder, welche in die Influx Datenbank geschrieben werden

7.10 Frontend

Das Frontend wurde als Single Page Application entwickelt. Dazu wurde das Framework VueJS eingesetzt.

Es wurde ein komponentenbasierter Entwicklungsansatz gewählt. Dies bedeutet, dass jedes Element der Webseite eine eigene Komponente ist. Eine solche Komponente ist in sich geschlossen und hat ihr eigenes HTML, CSS und JS Code. Dies ermöglicht eine hohe Wiederverwendbarkeit im Projekt. Eine solche Komponente kann aber auch einfach in einem anderen Projekt wiederverwendet werden.

Das Frontend benutzt einen Router, der es ermöglicht, die "Seite zu wechseln" ohne neu zu laden. Dieser Router übernimmt auch die Aufgabe zu überprüfen, ob der Benutzer eingeloggt ist. Falls nicht, wird er automatisch auf die Log-In Seite weitergeleitet.

Im Bild 53 wurden die einzelnen Komponenten rot hervorgehoben. Man sieht, dass zum einen die einzelnen Seiten der Webseite als Komponenten implementiert wurden. Zum anderen wurden komplexere Elemente wie Buttons oder die formatierte JSON-Ausgabe als eigene Komponente implementiert. Dabei konnten auch diverse schon bestehende Komponenten aus früheren Projekten oder aus Open Source Projekten verwendet werden.

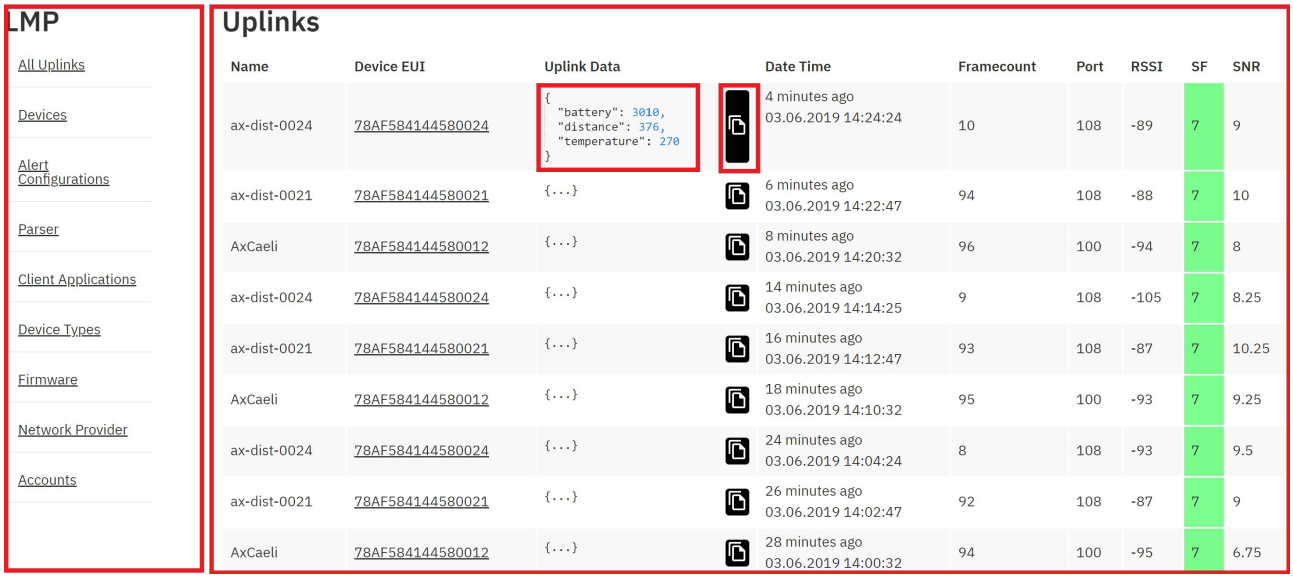


Abb. 53: Die Übersicht über eingegangene Uplinks. In Rot wurden die VueJS Komponenten hervorgehoben.

Im Bild 53 sieht man die Übersicht über eingegangene Uplinks. Wenn die Datenpakete geparsed wurden, kann man sich diese als JSON-Objekt anzeigen lassen. Mit dem Button daneben können die Daten in die Zwischenablage kopiert werden. Des Weiteren besteht das Frontend aus vielen Formularen und Listen um die CRUD Use Cases abzudecken. Hier ein Auszug davon:

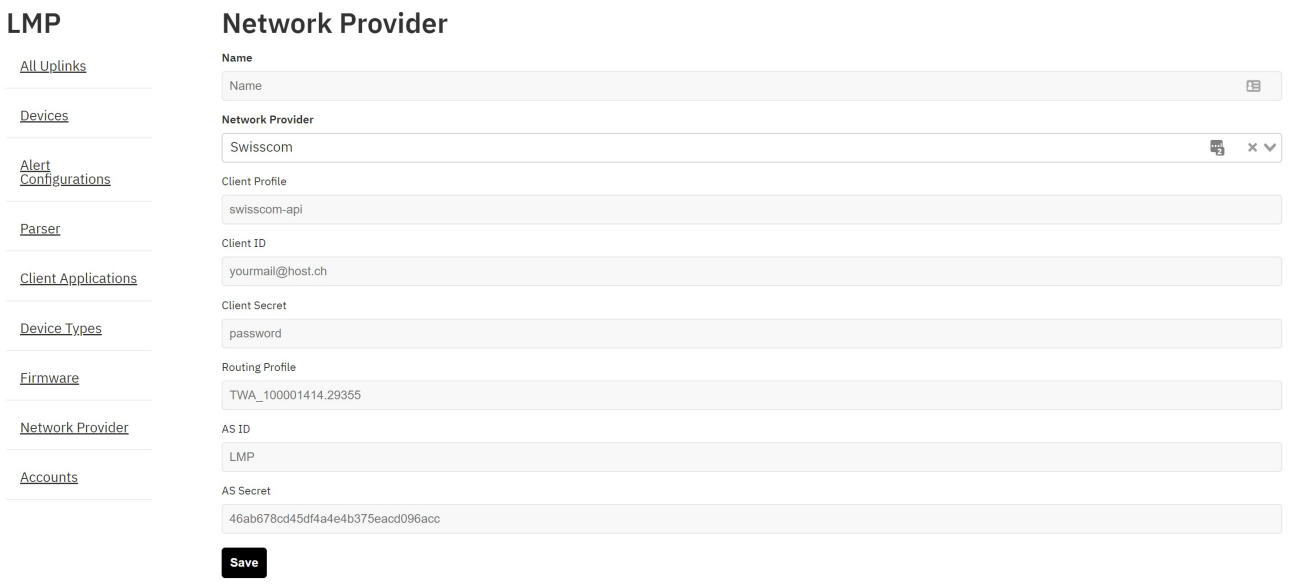


Abb. 54: Das Formular um einen Netzwerkanbieter zu erstellen

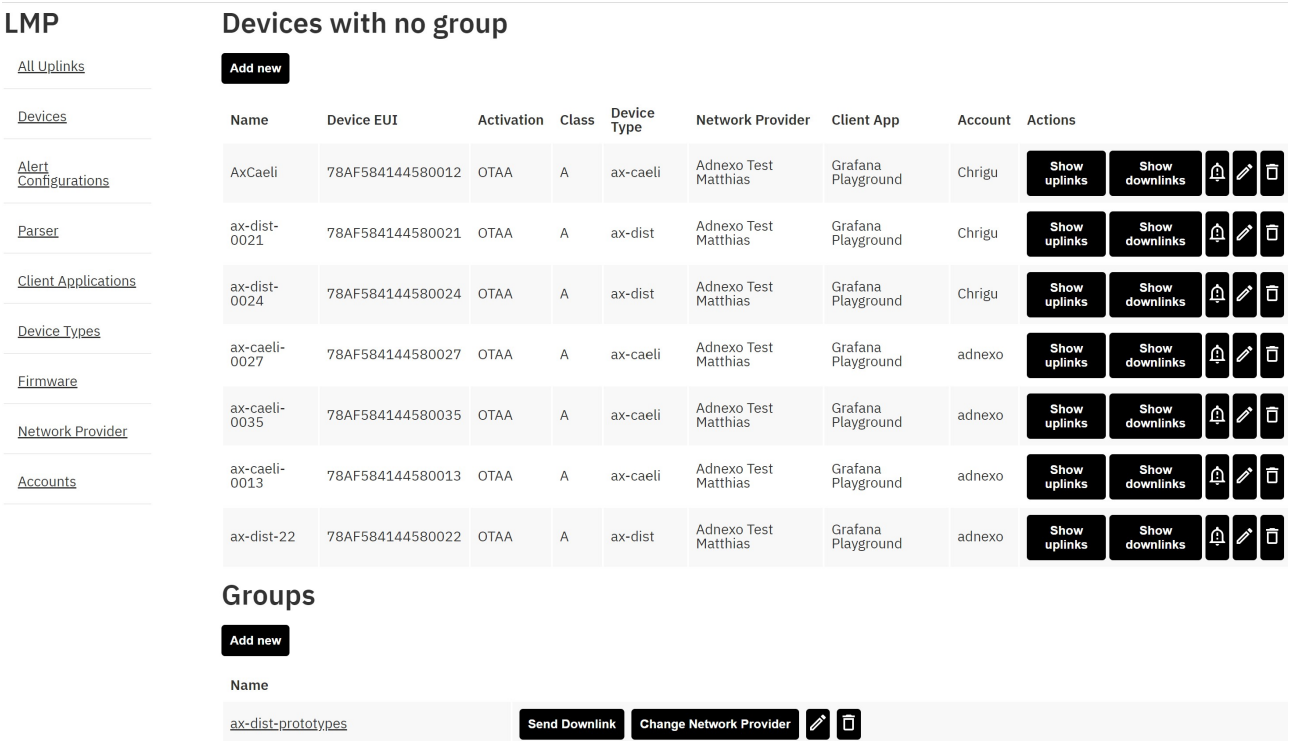


Abb. 55: Die Liste der erfassten Geräte und der vorhandenen Gruppen

Hier das Formular zum Erstellen eines Parsers. Um dem Benutzer die Code-Eingabe zu vereinfachen wurde Code Highlighting eingefügt. Darunter gibt es ein Feld mit welchem man einen Uplink simulieren und so den Code testen kann.

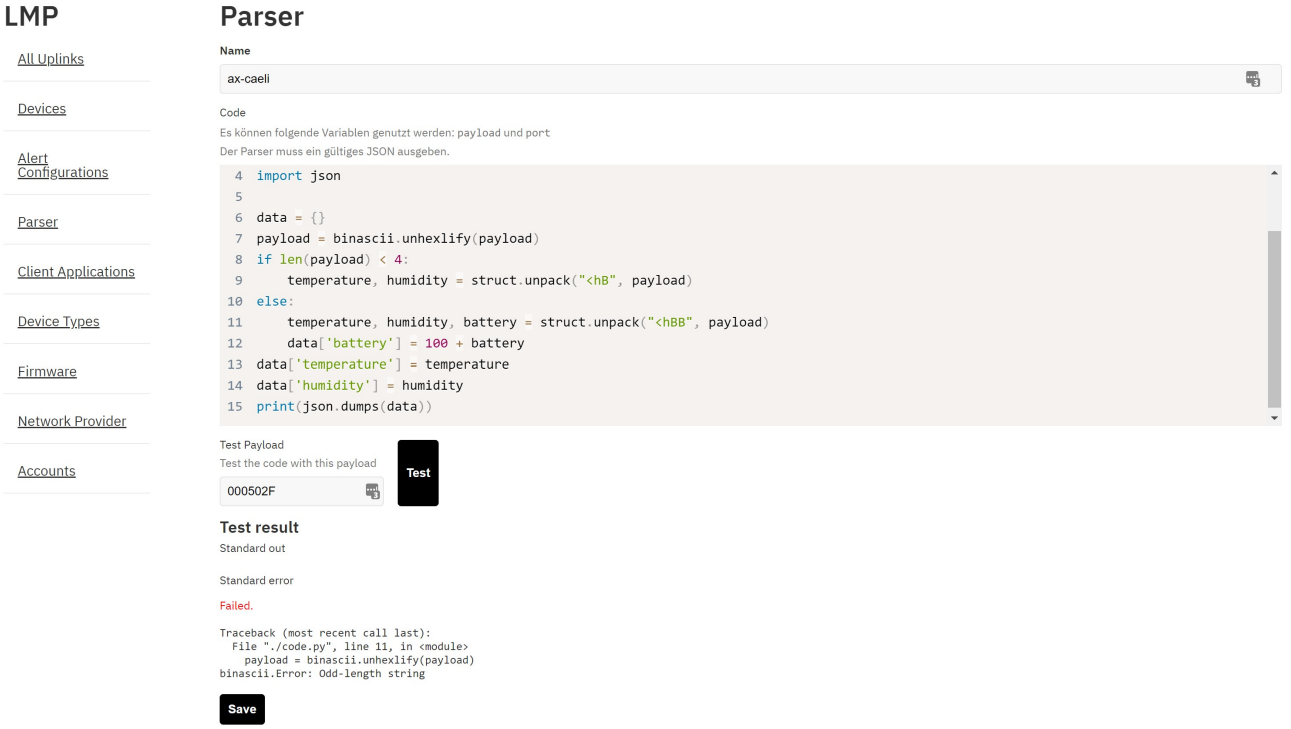


Abb. 56: Das Formular zum Erstellen eines Parsers

Nachdem wir das Frontend dem Industriepartner zugänglich machten, erhielten wir wertvolle Rückmeldungen. Basierend darauf haben wir das Frontend angepasst. Zum Beispiel ist es nun möglich, die DevEUI beim Erstellen eines Devices in diversen Formaten einzufügen. Das Eingabefeld wandelt es dann automatisch in das von der Plattform verlangte Format.

7.11 Testresultate

7.11.1 Systemtest

Um die Funktionalität zu testen, wurden System Tests definiert. Für das Resultat der Arbeit wurde ein solcher Testlauf durchgeführt. Die Spezifikation sowie das Resultat der Tests können im Anhang eingesehen werden.

7.11.2 Frontendtesting

Das Frontend wurde regelmässig vom Industriepartner getestet und anhand von Feedbacks konnten während der Construction Phase bereits die nötigen Änderungen und Verbesserungen an der Usability der Webseite getätigt werden.

7.11.3 Lasttest

Wir hatten an der HSR einen virtuellen Server, mit folgender Spezifikation: 2 vCPU max. 2.2 GHz, 2 GB RAM. Darauf haben wir alle Services laufen gelassen, und einen Lasttest durchgeführt.

Uplinks Es wurde analysiert, wie viele Uplinks die Applikation verarbeiten kann. Dazu wurden API-Benutzer simuliert, welche jede Sekunde einen Uplink versenden, sobald der vorhergehende abgeschlossen wurde. Die Simulierten API-Benutzer entsprechen hiermit nicht dem, was unsere Applikation verwalten kann, sondern wie viele API-Benutzer gleichzeitig unsere API nutzen (und einen Uplink senden).

Es wurde nacheinander mit 10, 20, 50 und 80 simulierten API-Benutzer getestet. Dabei wurde der Lasttest durchgeführt bis zirka 1000 Requests gemacht wurden.

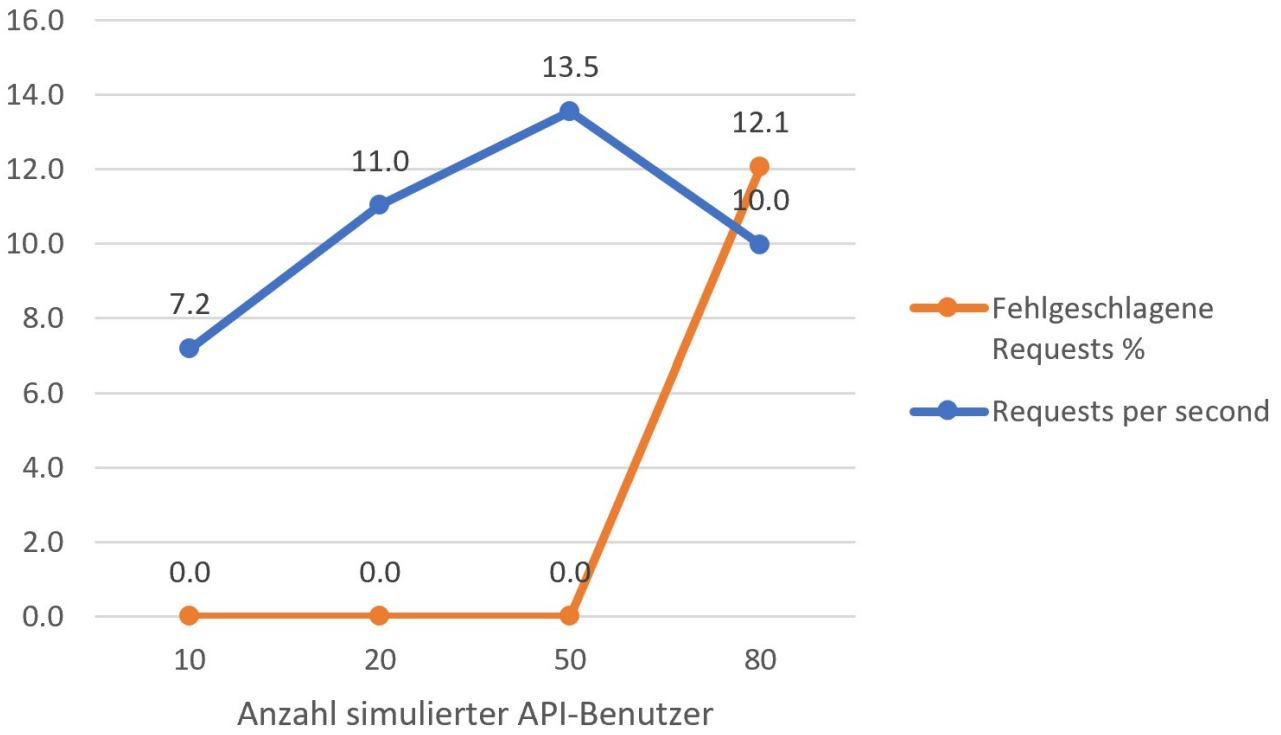


Abb. 57: Uplinks pro Sekunde sowie die fehlgeschlagenen Requests

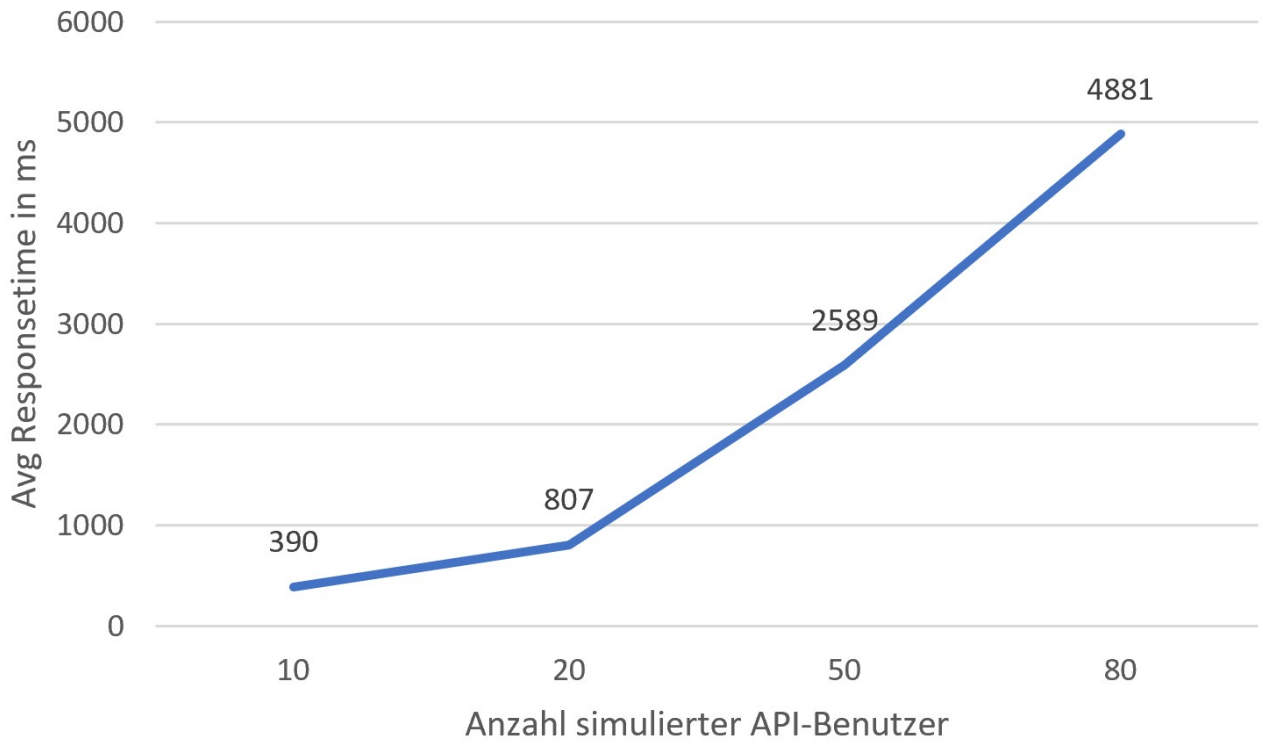


Abb. 58: Durchschnittliche Antwortzeit in Millisekunden

Natürlich steigen die möglichen Requests per Second (RPS) mit steigender Anzahl API-Benutzer, das

Maximum fanden wir bei 13.5 RPS. Bei mehr RPS kam der Server nicht mehr mit der Verarbeitung nach. Dabei stellte sich heraus, dass der Parsing-Service nicht mehr alle Anfragen annehmen konnte. Dieser ist zuständig für die Ausführung des Python-Codes, welcher der Benutzer definiert für das Parsen des Payloads, sowie für die Alerts. Da pro Uplink ein Parser, sowie zwei Alerts hinterlegt waren, wurde der Parsing-Service dreimal pro Anfrage aufgerufen.

Selbst auf diesem eher schwachen Server, auf dem alle Services gleichzeitig liefen, konnte unsere nichtfunktionale Anforderung an die Skalierbarkeit erfüllt werden.

Wir liessen die Applikation auch auf einem Laptop laufen, mit folgenden Spezifikation: 4 x 2.6 GHz, 16 GB RAM. Darauf erreichten wir 25.2 RPS, bei Skalierung des Parsing-Services auf zwei (wobei die zweite Instanz auf einem anderen Laptop lief) erreichten wir sogar 30 RPS.

Unsere Nachforschungen haben ergeben, dass der Parsing-Service im Verhältnis sehr langsam ist. Er braucht ca. 130-150 ms um eine einzelne Anfrage zu beantworten.

Downlinks Beim Lasttest der Downlinks stellten wir fest, dass es grosse Unterschiede zwischen Loriot und Swisscom gab. Wenn wir einen Lasttest durchführten, und Downlinks an Swisscom sendeten, kamen wir nur auf 2.7 RPS.

Es stellt sich heraus, dass die Swisscom zur Verarbeitung eines Downlinks im Durchschnitt 1382ms Zeit braucht. Zum anderen scheinen sie das Senden von Downlinks stark zu drosseln. Nach ca. 10 Aufrufen pro Sekunde, antwortet ihre API mit einem HTTP Statuscode 503 "Temporary unavailable".

Im krassen Gegensatz dazu, erreichen wir bei Loriot 49.19 RPS.

Beim Senden von Downlinks an Loriot, konnten unsere nichtfunktionalen Anforderungen locker erreicht werden. Bei der Swisscom sind wir weit davon entfernt, haben aber leider keinen Einfluss darauf, wie schnell die Swisscom reagiert.

Schlussfolgerung Sollte die Last der Uplinks stark steigen, müssen mehr Parsing-Services gestartet werden. Diese brauchen sehr lange um Anfragen zu beantworten, und sind das schwache Glied in der Kette. Speziell da dieser Service pro erhaltenem Uplink mehrmals aufgerufen werden kann, wenn Alerts für diesen Uplink definiert wurden.

Bei den Downlinks muss man auf die Limite der Swisscom achten.

7.12 Metriken

Um Codemetriken analysieren zu können, wurde das IntelliJ-Plugin "MetricsReloaded" [9] von JetBrains verwendet.

[9] `https://plugins.jetbrains.com/plugin/93-metricsreloaded`

Projekt	**Non-Commented Lines of Code (NCLOC)**
ch.hsr.common	1'554
ch.hsr.webservice	2'567
ch.hsr.swisscomproviderservice	1'018
ch.hsr.loriotproviderservice	747
ch.hsr.alertingservice	251
ch.hsr.parsingservice	118
ch.hsr.postoutputservice	162
ch.hsr.influxoutputservice	141
Gesamt	**6'558**

Tab. 16: Metrik: Non-Commented Lines of Code

8 Schlussfolgerung

8.1 Ergebnisse

Es konnten so gut wie alle Use Cases abgedeckt werden und alle in der Aufgabenstellung verlangten Ziele konnten erreicht werden:

Use Case	**Priorität**	**Umgesetzt**
UC01: Uplink senden	MUSS	Ja
UC02: Uplink erhalten	MUSS	Ja
UC03: Dekodierten Uplink erhalten	MUSS	Ja
UC04: CRUD IoT Gerät	MUSS	Ja
UC05: IoT Gerät bei Netzwerk registrieren	MUSS	Ja
UC06: CRUD IoT Gerätegruppe	MUSS	Ja
UC07: Downlink senden	MUSS	Ja
UC08: Geräteprobleme erkennen und anzeigen	MUSS	Ja
UC09: Benachrichtigung bei Geräteproblemen erhalten	MUSS	Nein
UC10: Pakete analysieren	MUSS	Ja
UC11: Archivierte Paketdaten einsehen	MUSS	(Ja)
UC12: CRUD Application Server	MUSS	Ja
UC13: CRUD Gerätetyp	MUSS	Ja
UC14: Parser hinterlegen	MUSS	Ja
UC15: Netzwerkanbieterendpunkt konfigurieren	MUSS	Ja
UC16: CRUD Kundenaccount	MUSS	Ja
UC17: Rechtevergabe Kundenaccount	KANN	Nein

Tab. 17: Loriot Registrierung Geräteeigenschaften

Der als optional gekennzeichnete UC17 "Rechtevergabe Kundenaccount" konnte aus Zeitmangel nicht

mehr umgesetzt werden. Ebenso nicht umgesetzt wurde UC09 "Benachrichtigung bei Geräteproblemen erhalten", bei welchem wir mit dem Industriepartner in einem Meeting besprochen haben, dass das Anzeigen der Alerts im Frontend genügt. Die Filterfunktion bei UC11 konnte auf Grund fehlender Zeit nicht mehr umgesetzt werden.

Zusätzlich wurden allerdings noch weitere Funktionalitäten implementiert, die erst im Verlaufe der Arbeit, von uns oder von der Adnexo GmbH, erkannt wurden. Dazu gehört zum Beispiel, dass man den Parser Code direkt beim Erfassen mit einem Beispiel Payload testen kann, oder dass gewisse Aktionen direkt auf der Geräte Gruppe für alle Geräte ausgeführt werden können. Sowie diverse Features im Frontend, welche die Benutzung für den Benutzer vereinfachen (Copy to Clipboard, automatisches Parsen der DevEUI bei Copy&Paste, ...). Zusätzlich haben wir auch noch einen zweiten Outputservice entwickelt, welcher die Datenpakete in eine Influx Datenbank schreiben kann. Dies ermöglicht dem Industriepartner, Auswertungen über die Zeit zu machen.

8.2 Weiteres Vorgehen

Obwohl wir viel erreicht haben, bleiben trotzdem gewisse Punkte offen, die man noch verbessern könnte.

Wir haben während der Entwicklung offene Punkte und Bugs im Issue Tracker erfasst, welche bei einer Weiterführung der Arbeit abgearbeitet werden können.

Damit die Plattform auch Aussenstehenden zugänglich gemacht werden könnte, müsste noch der Use Case UC17 "Rechtevergabe Kundenaccount" umgesetzt werden.

Etwas, was der Plattform sicher einen erheblichen Mehrwert bringen könnte, ist, dass sie die Ver- und Entschlüsselung der Datenpakete selbst vornehmen kann. Bis jetzt überlässt die LoRaWAN-Management-Plattform (LMP) diese Arbeit dem Netzwerkbetreiber. Erreichen könnte man dies, indem man bei der Aktivierung des Gerätes über Activation by Personalization (ABP) dem Netzwerkanbieter den sogenannten "AppSKey" nicht mitteilt. Mit diesem Key wird das Gerät seine Payload verschlüsseln, und erst die LMP kann diese dann wieder entschlüsseln.

Natürlich könnte man die Plattform in Zukunft um weitere Netzwerkanbieter ergänzen. Auch haben wir vorgesehen, dass die LMP um weitere Output-Kanäle ergänzt werden kann. Hier würde sich die Erweiterung um Websockets anbieten, um "live" neue Datenpakete anzeigen zu können. Um dies zu erreichen, müssten sogenannte "sticky sessions" auf dem Load Balancer aktiviert werden, damit der Client immer auf den selben Websocket-Outputservice verbindet.

Beim Lasttest haben wir auch festgestellt, dass die Swisscom sehr langsam beim Verarbeiten der Downlinks ist. Damit die Client Applikation nicht auf die Swisscom warten muss, könnte man hier noch eine Warteschlange einbauen, welche die Anfragen der Client Applikation annimmt, und dann asynchron abarbeitet.

Weiter in die Zukunft geschaut, wäre die LMP der perfekte Ort um einen Digital Twin[10] der IoT Geräte zu implementieren.

[10] `https://en.wikipedia.org/wiki/Digital_twin`

8.3 Bewertung

Unser Resultat erfüllt die Aufgabe, welche an uns gestellt wurde, und konnte sogar noch um zusätzliche Funktionalitäten erweitert werden.

Es reichte sogar für ein schönes UI. Natürlich ist dieses noch ausbaufähig, aber gut zu gebrauchen.

Dank den wöchentlichen Sitzungen mit dem Betreuer, und den zweiwöchentlichen Sprint Meetings mit dem Industriepartner, konnten Unklarheiten schnell erkannt und behoben werden. Zusätzlich konnten wir auch während der Entwicklungsphase auf die Bedürfnisse und Wünsche des Industriepartners eingehen.

Dank der klaren Struktur und dem Microservice-Ansatz, wird der Industriepartner in der Lage sein, der Applikation einfach weitere Features hinzuzufügen. Auch ist es denkbar, dass er einzelne Services aus anderen Applikationen heraus anspricht, um diese nicht neu entwickeln zu müssen.

Aus der Sicht der Projektplanung konnten die Meilensteine, welche in der Evaluation-Phase bestimmt wurden, termingerecht erreicht und umgesetzt werden. Auch das Zeitmanagement wurde durchdacht und zielführend geplant.

Insgesamt haben wir ein Produkt entwickelt, welches vom Industriepartner eingesetzt wird und ihm einiges an Arbeit abnehmen kann. Es hilft ihm nicht nur beim Verwalten von IoT Geräten, sondern bildet auch eine Grundlage bei der Neuentwicklung von Applikationen, welche mit einem LoRaWAN Netzwerkanbieter kommunizieren müssen.

8.4 Erkenntnisse

Das Entwickeln von Microservices hat diverse Vorteile, ist aber, vor allem für ein Zweierteam, auch ein Mehraufwand. Während der Entwicklung muss man oft diverse Services gleichzeitig starten, was die Komplexität deutlich erhöht.

Ein weiterer Punkt, den wir das nächstes Mal besser machen würden, ist das Teilen von gemeinsamen Dateien zwischen den Microservices. Wir haben dies in diesem Projekt mit einem Git Submodule gelöst, welches gemeinsame Dateien enthielt, und bei jedem Service eingebunden wurde.

Das hat funktioniert, hat aber immer wieder für Verwirrungen gesorgt, weil man sichergehen musste, dass man den Service auch mit der neusten Version der gemeinsamen Dateien testet.

Das nächste Mal würden wir ein Maven Repository nutzen, und darauf ein Projekt publizieren, welches die gemeinsamen Dateien enthält. Dieses Package könnte dann bei jedem Projekt eingebunden werden und würde automatisch von Maven verwaltet.

Codeauszüge

Abbildungsverzeichnis

Tabellenverzeichnis

References

[1] Wikipedia. Low power wide area network. https://de.wikipedia.org/wiki/Low_Power_Wide_Area_Network. Abgerufen am: 11.06.2019.

[2] LoRa Alliance. What is lorawan? https://lora-alliance.org/sites/default/files/2018-04/what-is-lorawan.pdf. Abgerufen am: 11.06.2019.

[3] LoRa Alliance. Was ist die lorawan-spezifikation? https://lora-alliance.org/about-lorawan. Abgerufen am: 04.03.2019.

[4] Wikipedia. Endgerät. https://de.wikipedia.org/wiki/Endger%C3%A4t. Abgerufen am: 04.03.2019.

[5] Wikipedia. Gateway (informatik). https://de.wikipedia.org/wiki/Gateway_(Informatik). Abgerufen am: 11.06.2019.

[6] 3GLTEinfo. Lorawan frequency bands. http://www.3glteinfo.com/lora/lorawan-frequency-bands/. Abgerufen am: 11.06.2019.

[7] Xiaofen Zhang Jing Sun. Study of zigbee wireless mesh networks. *Ninth International Conference on Hybrid Intelligent Systems*, 2009.

[8] ZigBee Alliance. Zigbee 3.0. https://www.zigbee.org/zigbee-for-developers/zigbee-3-0/. Abgerufen am: 11.06.2019.

[9] ZigBee Alliance. Zigbee: Securing the wireless iot. *ZigBee Security Whitepaper*, 2017.

[10] LinkLabs. An nb-iot architecture breakdown for iot architects. https://www.link-labs.com/blog/nb-iot-architecture. Abgerufen am: 11.06.2019.

[11] PostgreSQL Global Development Group. Postgresql - different replica solutions. https://www.postgresql.org/docs/9.2/different-replication-solutions.html. Abgerufen am: 11.06.2019.

[12] Spring cloud: Peer awareness. https://cloud.spring.io/spring-cloud-static/spring-cloud.html#_peer_awareness. Abgerufen am: 11.06.2019.

[13] Swisscom (Schweiz) AG. *Swisscom LPN-CMP Application Development Guide*, 2016.

[14] Actility. Dx core api - reference documentation. https://dx-api.thingpark.com/core/latest/doc/index.html#device-operations. Abgerufen am: 11.06.2019.

[15] Loriot. Api data format | loriot. https://eu1.loriot.io/documentation/applications/apidataformat. Abgerufen am: 11.06.2019.

[16] codecentric/spring-boot-admin: Admin ui for administration of spring boot applications. https://github.com/codecentric/spring-boot-admin. Abgerufen am: 04.06.2019.

[17] Papertrail - cloud-hosted log management, live in seconds. https://papertrailapp.com/. Abgerufen am: 04.06.2019.

[18] John Carnell. *Spring Microservices in Action*. Manning Publications, 2017.

[19] Sentry | error tracking software — javascript, python, php, ruby, more. https://sentry.io/. Abgerufen am: 04.06.2019.

[20] Git - submodules. https://git-scm.com/book/en/v2/Git-Tools-Submodules/. Abgerufen am: 04.06.2019.

Acronyms

ABP Activation by Personalization. 91

API Application Programming Interface. 30

CI continuous integration. 30

CRUD Create/Read/Update/Delete. 44

GUI Graphical User Interface. 14, 30

IoT Internet of Things. 8, 24, 25, 91, 92, 103

LMP LoRaWAN-Management-Plattform. 91

LoRaWAN long range wide-area network. 22, 24, 25, 27

RPS Requests per Second. 87, 88

RSSI Received signal strength indication. 28

SF Spreading Factor. 27

SNR Signal Noise Ratio. 28

UI User Interface. 30, 92

WPAN Wireless Personal Area Network. 18

Glossary

Application Server Als Application Server bezeichnen wir diejenige Applikation, welche unsere Plattform als Abstraktionsschicht zu den Netwerk Anbietern nutzt. Diese Applikation ist interessiert an den Datenpaketen der IoT Geräte, und verarbeitet diese.. 12, 24

Bearer Token Ein Bearer Token ist ein HTTP Authentifizierungs Schema, bei dem dem Halter eines Token Zugriff auf etwas gewährt wird. Dieses Token wird jeweils im "Authorization" Header mitgesendet.. 57

Client Applikation Client Applikation wird synonym für Application Server verwendet. 24

Downlink Ist ein Daten-Paket, welches von einer Applikation an ein IoT Gerät gesendet wird. 13, 24, 25, 44, 48, 88, 91

Uplink Ist ein Daten-Paket, welches von einem IoT Gerät versendet wird. 13, 24, 25, 27, 44–47, 81, 83, 86, 88

Part V

Attachments

A Projektplan

Revision History

Revision	Date	Author(s)	Description
0.1	18.02.2019	Raffael Vögeli	Initialer Draft
0.2	25.02.2019	Raffael Vögeli	Finalisierung
0.3	06.03.2019	Matthias Dunkel	Anpassungen gemäss Besprechung
0.4	11.03.2019	Raffael Vögeli	Anpassungen Verlängerung Elaboration
0.5	24.05.2019	Raffael Vögeli	Zusammenlegeung Construction 5 & Construction 6 zu einem Sprint (Outcome aus Scrum Meeting)

A.1 Einführung

A.1.1 Zweck

Der Projektplan ist ein Instrument zur Planung, Steuerung und Kontrolle durch das Projektmanagement. Dieses Dokument dient zusätzlich als Beschreibung der Bachelorarbeit, in welchem das Projekt "LoRaWAN-Management Plattform" in einer ersten Version umgesetzt wird.

A.1.2 Gültigkeitsbereich

Der Gültigkeitsbereich beschränkt sich auf die Gesamtdauer des Projektes "LoRaWAN-Management Plattform" im Modul Bachelorarbeit des Frühjahrsemesters 2019. Änderungen werden fortlaufend ergänzt und in der Änderungsgeschichte notiert.

A.2 Projektübersicht

A.2.1 Ziel und Zweck

In der heutigen Welt sind IoT-Geräte und die Vernetzung analoger Geräte allgegenwärtig und immer mehr im Aufschwung. Das Netzwerkprotokoll LoRaWAN definiert die Schnittstellen zwischen den IoT-Geräten über die Gateway zu den Netzwerkservern der verschiedenen Anbietern. Bei der Kommunikation zwischen den Netzwerkservern und den Client-Applikationen kommen allerdings verschiedenste Protokolle (HTTPS, MQTT, WebSockets etc.) zum Einsatz.

Ziel dieser Arbeit ist es nun, eine standardisierte Schnittstelle zu erschaffen, über welche die verschiedenen Netzwerkserver mit den unterschiedlichsten Client-Applikationen kommunizieren können.

Auch soll auf der neuen Applikation das Gerätemanagement der IoT-Geräte möglich sein.

A.2.2 Vorgehen

Einarbeitung

In einem ersten Schritt werden ein paar der gängigsten IoT-Technologien analysiert. Dabei geht es darum einen Überblick über die Technologien zu erhalten. Es ist aber bereits klar, dass die Applikation für LoRaWAN umgesetzt werden soll.

Planung & Implementation

Als zweiter Schritt steht die Planung und Entwicklung der System-Architektur sowie des Front- und Backends der Applikation im Vordergrund. Daraus ergeben sich folgende Resultate:

- API für die Anbindung der Client-Applikationen
- User Interface für Management Plattform

A.2.3 Lieferumfang

Als Abgabe werden folgende Objekte angesehen:

- Dokumentation der Analyse
- funktionstüchtige Applikation gemäss Anforderungen
- API Spezifikation
- Software Dokumentation

A.2.4 Annahmen und Einschränkungen

Dieses Projekt findet in Form eines Modules mit 2 x 12 ECTS Punkten statt. Daher soll das Projekt mit 2 beteiligten Personen im Rahmen von 720 Arbeitsstunden durchgeführt werden. Das Projekt endet dabei aber spätestens am letzten Tag der Moduldurchführung - dem 14.06.2019.

A.3 Projektorganisation

Das Projekt "LoRaWAN-Management Plattform" wird im Rahmen der Bachelorarbeit FS2019 umgesetzt. Das Projekt-Team besteht aus zwei Studierenden, zusätzlich einem Auftraggeber, und wird von einem Dozenten betreut.

A.3.1 Organigramm

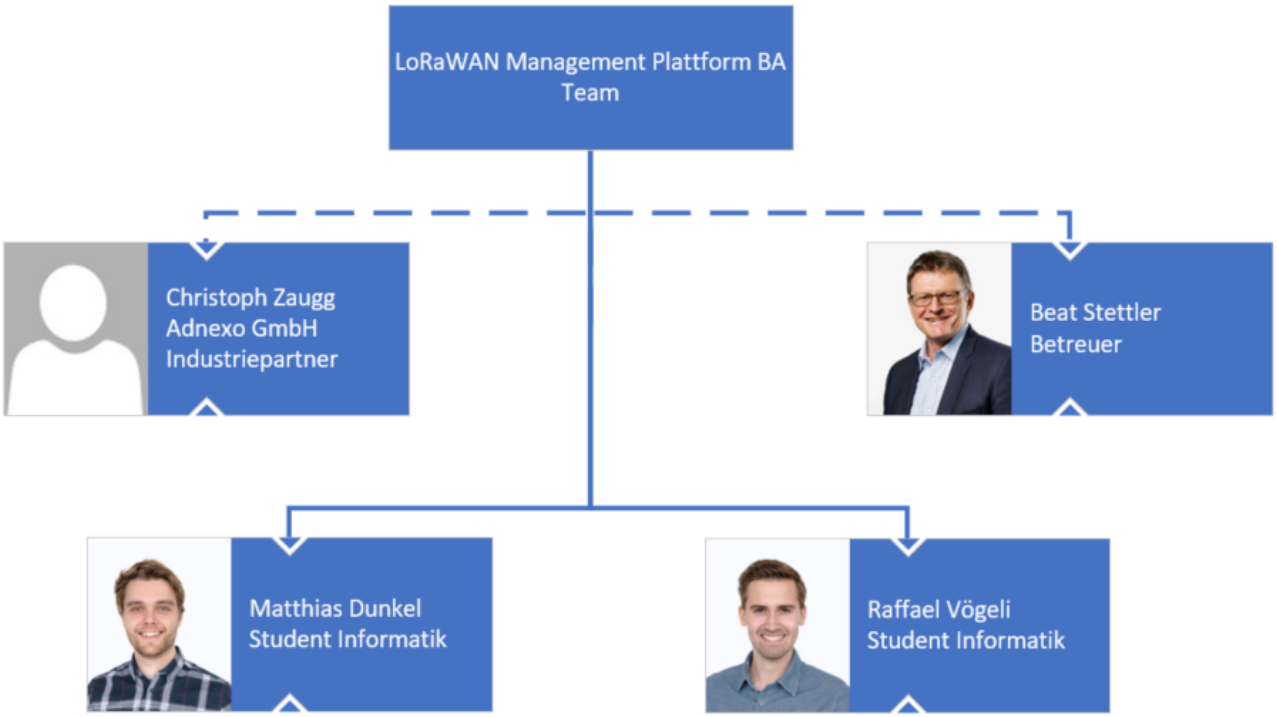


Abb. 59: Organigramm

Teammitglieder

Nachfolgend sind alle Teammitglieder mit ihren Zuständigkeiten und Kontaktangaben aufgelistet.

Matthias Dunkel

Email	mdunkel@hsr.ch
Zuständigkeiten	Projektplanung, Dokumentation, Entwicklung

Tab. 19: Angaben Matthias Dunkel

Raffael Vögeli

Email	rvoegeli@hsr.ch
Zuständigkeiten	Projektplanung, Dokumentation, Entwicklung

Tab. 20: Angaben Raffael Vögeli

A.3.2 Externe Schnittstellen

Folgende Personen werden als Ansprechpartner und Betreuer der Projektarbeit benötigt.

Betreuer

Beat Stettler

Email	beat.stettler@ins.hsr.ch
Zuständigkeiten	Betreuer der Bachelorarbeit

Tab. 21: Angaben Beat Stettler

Industriepartner

Die Arbeit wird zusammen mit der Adnexo GmbH durchgeführt. Folgende Personen dienen als Ansprechpartner des Industriepartners:

Christian Fässler

Email	christian.faessler@adnexo.ch
Zuständigkeiten	Geschäftsführer adnexo GmbH

Tab. 22: Angaben Christian Fässler

Martin Haas

Email	martin.haas@adnexo.ch
Zuständigkeiten	Geschäftsführer adnexo GmbH

Tab. 23: Angaben Martin Haas

Christoph Zaugg

Email	christoph.zaugg@adnexo.ch
Zuständigkeiten	IoT Hardware Engineer adnexo GmbH

Tab. 24: Angaben Christoph Zaugg

A.4 Managementabläufe

A.4.1 Zeitplanung

Als Projektvorgehensmodell wird Scrum+ (inkl. RUP) verwendet. Dabei wird das Projekt in vier Phasen namens Inception, Elaboration, Construction und Transition aufgeteilt. Scrum wird mit Iterationslängen von 2 Wochen durchgeführt.

Phasen

Phase	Beschreibung	Dauer
Inception	Kickoff Meeting, Aufgabenstellung, Projektplan	1 Woche
Elaboration	Technologieanalyse, Anforderungen, Research, Architektur	3 Wochen
Construction	Realisierung, Testing, Dokumentation	11 Wochen
Transition	Abschluss Dokumentation, Projektreflexion, Poster	2 Wochen

Tab. 25: Projektphasen und deren Inhalte

Meilensteinübersicht

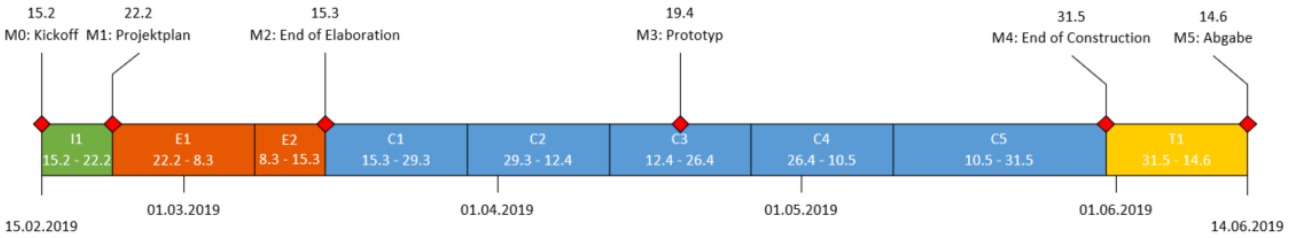


Abb. 60: Meilensteinübersicht

Meilensteine

Name	Datum	Work Products
M0 – Kickoff Meeting	15.02.2019	• Kennenlernen • Besprechung der Aufgabenstellung
M1 – Aufgabenstellung, Projektplan	22.02.2019	• Projektplan erstellt • Aufgabenstellung definiert und unterschrieben
M2 – End of Elaboration	15.03.2019	• IoT-Technologien evaluiert • Systemgrenzen definiert • Architektur definiert • Use Cases definiert (Funktionale sowie Nicht-Funktionale)
M3 – Erster Prototyp mit wichtigsten Funktionalitäten	19.04.2019	• Anbindung der Plattform an Client-Applikationen und Netzwerkserver möglich • Durchstich erreicht • Wichtigste Funktionalitäten abgedeckt
M4 – End of Construction	31.05.2019	• Anforderungen umgesetzt • Testing der Komponenten durchgeführt und protokolliert
M5 – Abgabe	14.06.2019	• Abgabe aller Dokumente • Abgabe des Codes

Tab. 26: Meilensteine

A.4.2 Meetings

Projektbesprechung Die Projektbesprechungen mit dem Betreuer und den zusätzlichen Projektpartnern werden wenn möglich einmal pro Woche durchgeführt. Diese Meetings werden protokolliert.

Teambesprechung Die Teambesprechungen werden an den jeweiligen Arbeitstagen (dreimal pro Woche) in einem Zeitrahmen von 15-30 Minuten durchgeführt. Dabei werden die durchgeführten Arbeiten, allfällige Probleme, und das weitere Vorgehen besprochen und definiert

SCRUM Sprint Planning In der Construction Phase wird jeweils am Anfang eines Sprints ein Sprint Planning Meeting mit dem Industriepartner gehalten, in dem die Arbeitspakete für den nächsten

Sprint festgelegt werden. Diese Meetings werden protokolliert.

A.4.3 Kostenvoranschlag

Die Umsetzung des Projekt erfolgt im Frühjahrssemester 2019, welches vom 18.02.2019 bis zum 14.06.2019 dauert. Der Zeitaufwand pro Woche unter dem regulären Semester, welches 15 Wochen lang dauert, wird mit 40h[11] pro Woche berechnet. Zusätzlich stehen 2 weitere Wochen nach dem Semester bis zur Abgabe zur Verfügung, in welchen ein Aufwand von 40h pro Person geleistet werden kann. Ausserdem werden 2h[12] pro Woche für Meetings aufgewendet.

[11] 20 Stunden pro Person
[12] Je 1h für Besprechung mit Betreuer und im Team

A.5 Risikomanagement

Nr	Titel	Beschreibung	max. Schaden [h]	Eintrittswahrscheinlichkeit	Gewichteter Schaden	Vorbeugung	Verhalten beim Eintreten
R1	Ausfall JIRA	JIRA ist nicht mehr erreichbar	1	8%	0.08	Projektbeteiligte haben genügend Informationen um ohne JIRA zu arbeiten	Melden des Problems an JIRA
R2	Datenverlust JIRA	Daten von JIRA beschädigt oder verschwunden	2	5%	0.1	Backup der JIRA Daten	Backup einspielen
R3	Code Datenverlust	Code auf git geht verloren	2	5%	0.1	Lokale Arbeitskopien erstellen	Lokale Arbeitskopien der Teammitglieder werden zusammengeführt und ins Git Repository hochgeladen
R4	Dokumentenverlust	Dokumente auf GitLab gehen verloren	2	5%	0.1	Lokale Sicherungen erstellen	Einspielen der lokalen Sicherung
R5	Ungenügende Kommunikation	Kommunikation mit Kunde / Betreuer ist ungenügend oder schwierig	4	5%	0.2	Regelmässige Planung von Sitzungen und Kenntnisnahme über Verständnis	Kontakt mit Kunde / Betreuer aufnehmen, Sitzungsplanung überdenken
R6	Kommunikation mit Netzwerkanbieter	Der Netzwerkanbieter ist nicht erreichbar resp. erreicht der Netzwerkanbieter die Applikation nicht	8	5%	0.4	Erreichbarkeit im Internet sicherstellen	Analyse und Behebung des Problems
R7	Downlinks erreichen nicht das IoT Endgerät	Das IoT Endgerät empfängt den Downlink nicht.	1	50%	0.5	Applikation muss auf fehlgeschlagenes Senden reagieren.	Downlink nach einer gewissen Zeit nochmals senden.
R8	Applikations-API ändert sich	API Endpunkte der Applikation ändern sich. Bestehende Consumer der API sollten nicht angepasst werde müssen.	4	30%	1.2	Versionierung der API, damit geänderte Funktionen auf anderen API Endpunkten angeboten werden können.	Consumer müssen benachrichtigt werden.
R9	Performance	Die Applikation ist langsam	24	10%	2.4	In Architekturplanung einfliessen lassen	Architektur überdenken, Problem evaluieren
R10	Netzwerkanbieter ändert API-Spezifikation	Die Applikation kann mit der neuen API-Spezifikation nicht mehr kommunizieren	30	25%	7.5	Entkopplung der Kommunikation zum Netzwerkanbieter innerhalb der Applikation	Anpassung der Kommunikationskomponente zum Netzwerkanbieter
R11	Unterschätzte Komplexität	Teammitglied braucht viel mehr Zeit als geplant für ein Arbeitspaket	40	40%	16	Zeitreserven einplanen, Regelmässige Kontrolle des Backlogs	Arbeitspakete in kleinere Teilpakete aufteilen oder schlimmstenfalls auf Feature verzichten
R12	Einarbeitung in neue Technologien und Frameworks	Gewisse Technologien und Framework sind uns unbekannt und wir benötigen mehr Zeit um unse einzuarbeiten	80	40%	32	Genügend Zeit einplanen.	Mehr Zeit investieren und gegebenfalls Projektumfang reduzieren.
Summe			198		60.58		

Abb. 61: Risikoanalyse

A.6 Arbeitspakete

Die Arbeitspakete werden mit Hilfe von JIRA geplant. Die Zeiterfassung erfolgt mit Clockify. Mit Hilfe des folgenden Links gelangt man auf die Projektseite:

- Jira: `https://adnexo.atlassian.net/jira/software/projects/LMP/boards/17`
- GitLab - Dokumentation: `https://gitlab.com/lorawan-management-platform/ba-documentation`

A.6.1 Aufwand und Schätzung

Der Aufwand für einzelne Arbeitspakete wird in Stunden angegeben, wobei diese Werte auf Schätzungen beruhen. Bei einer schnelleren Umsetzung resp. aufkommenden unerwarteten Problemen können diese Zeiten abweichen.

Story Point Ein Story Point repräsentiert zu Beginn eine Arbeitsstunde. Der Aufwand eines Storypoints kann nach einem Sprintreview aber angepasst werden.

A.6.2 Priorisierung

Die Arbeitspakete werden wie folgt priorisiert:

Hoch Umsetzung zwingend notwendig, um einen erfolgreichen Abschluss des Projekts zu erreichen.

Normal Umsetzung notwendig, da andere Features davon abhängig sein können.

Niedrig Implementierung ist nicht unbedingt notwendig. Erfolgreicher Projektabschluss unabhängig von diesen Arbeitspaketen.

Um das Projekt erfolgreich beenden zu können, wird der Fokus auf die Arbeitspakete mit den Prioritäten Hoch und Normal gesetzt. Die Arbeitspakete mit der Priorität Niedrig werden bei übrig bleibender Zeit zusätzlich umgesetzt.

A.7 Qualitätsmassnahmen

In diesem Kapitel wird definiert, mit welchen Massnahmen die Qualität des Projekts so hoch wie möglich gehalten wird.

A.7.1 Dokumentation

Die von uns erstellten Dokumente befinden sich auf dem Git-Server und werden somit versioniert abgespeichert. Dadurch kann auf alte Zustände der Dokumente zugegriffen werden, um alte Ideen oder Beschlüsse noch einmal hervorzuholen.

Mit Hilfe eines Corporate Design stellen wir sicher, dass alle Dokumente einheitlich aussehen. Dieses Dokument-Template wird für alle abzugebenden Spezifikationen und Dokumentationen verwendet.

A.7.2 Projektmanagement

Als Projektmanagement-Tool im Einsatz steht JIRA, welches sich in der Jira Cloud von Adnexo GmbH befindet.

A.7.3 Definition of Done

Arbeitspakete werden als abgeschlossen betrachtet, sofern sie folgende Kriterien erfüllen:

- Funktionalität gemäss Beschreibung realisiert
- Sicherstellung das Funktionalität die Anforderungen abdeckt
- Feature wurde nachgeführt und geschlossen
- Wichtige Dokumentationen wurden angepasst
- Nachführung von Informationen und Zeiterfassung erledigt
- Issue wurde geschlossen
- Bei Implementationspaketen
  - Tests wurden implementiert
  - Fehlerfreie Unit-Tests (bestehende wie auch neue)

A.7.4 Entwicklung

Der Source-Code befindet sich in einem Git-Repository von GitLab, und kann dank Versionierung getrackt werden. Die Qualität wird durch regelmässige Code Reviews und durch das Verwenden einer Code Style Guideline sichergestellt. Alle erstellten Entwicklungs-Projekte, welche für die Arbeit benötigt werden, werden in einer GitLab-Gruppe gesammelt, um sie an einem einheitlichen Ort sammeln zu können.

`https://gitlab.com/lorawan-management-platform`

Unit Testing

Für alle wichtigen Klassen und Komponenten werden systematisch und regelmässig Unit Tests geschrieben. Diese Tests sollen garantieren, dass der Code fehlerfrei läuft.

Coding Guidelines

Das Projekt wird ahnhand der Coding Guidelines von Java[13] und JavaScript[14] implementiert.

Code Reviews

Die Teammitglieder reflektieren den Code des Anderen in regelmässigen Abständen und geben Verbesserungsvorschläge, welche im Team ausdiskutiert und umgesetzt werden.

Automatisierung der Builds

Für dieses Projekt werden wir die Builds auf einen Server auslagern und automatisieren.

[13] https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
[14] https://www.w3schools.com/js/js_conventions.asp

B Systemtests

B.1 Spezifikation

ST01 - Uplink von Netzwerkanbieter empfangen

Use Case	UC01
Testziel	Der Netzwerkanbieter kann ein Uplinkpaket eines Geräts an die Plattform senden.
Vorbedingungen	• Plattform ist als HTTP-Post Endpunkt beim Provider hinterlegt
Ausführung	1. Uplink von einem Endgerät aus initiieren
Erwartetes Ergebnis	Uplink wird in Uplink-Übersicht angezeigt

Tab. 27: ST01 - Uplink von Netzwerkanbieter empfangen

ST02 - Uplink eines bekannten Geräts an Client Application senden

Use Case	UC02
Testziel	Der erhaltene Uplink wird an die Client Application weitergeleitet
Vorbedingungen	• Plattform ist als HTTP-Post Endpunkt beim Provider hinterlegt • Der Client Application, bei welcher das Gerät hinterlegt wurde, besitzt einen gültigen Eintrag in den Post-Output Einstellungen.
Ausführung	1. Uplink von einem Endgerät aus initiieren
Erwartetes Ergebnis	Uplink wird in Client Application angezeigt

Tab. 28: ST02 - Uplink eines bekannten Geräts an Client Application senden

ST03 - Uplink eines unbekannten Geräts empfangen

Use Case	UC02
Testziel	Der erhaltene Uplink eines unbekannten Geräts wird aufgenommen.
Vorbedingungen	• Plattform ist als HTTP-Post Endpunkt beim Provider hinterlegt • Gerät ist nicht in Plattform hinterlegt
Ausführung	1. Uplink von einem Endgerät aus initiieren
Erwartetes Ergebnis	Uplink wird in Uplink-Übersicht angezeigt

Tab. 29: ST03 - Uplink eines unbekannten Geräts empfangen

ST04 - Dekodierten Uplink eines bekannten Geräts an Client Application senden

Use Case	UC03
Testziel	Der erhaltene dekodierte Uplink wird an die Client Application weitergeleitet
Vorbedingungen	• Plattform ist als HTTP-Post Endpunkt beim Provider hinterlegt • Der Client Application, bei welcher das Gerät hinterlegt wurde, besitzt einen gültigen Eintrag in den Post-Output Einstellungen.
Ausführung	1. Dekodierten Uplink von einem Endgerät aus initiieren
Erwartetes Ergebnis	Uplink wird von Plattform dekodiert und in Client Application angezeigt

Tab. 30: ST04 - Dekodierten Uplink eines bekannten Geräts an Client Application senden

ST05 - CRUD Device

Use Case	UC04
Testziel	Ein Gerät wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Gerät erstellen 2. Gerät in Geräteübersicht eingesehen 3. Gerät editieren und speichern 4. Änderungen in Geräteübersicht eingesehen 5. Gerät löschen
Erwartetes Ergebnis	Gerät wird erstellt, kann eingesehen, aktualisiert und gelöscht werden.

Tab. 31: ST05 - CRUD Device

ST06 - Gerät bei Netzwerkanbieter mit OTAA registrieren

Use Case	UC05
Testziel	Ein Gerät wird beim Netzwerkanbieter registriert
Vorbedingungen	• Benutzer ist eingeloggt • Netzwerkanbieter und benötigte Konfigurationen sind hinterlegt
Ausführung	1. Gerät erstellen 2. Netzwerkanbieter auswählen 3. Als Registrationsart OTAA wählen und entsprechende Informationen angebeben
Erwartetes Ergebnis	Deviceübersichtsseite wird angezeigt und Gerät wurde registriert. Entsprechenden Eintrag findet man beim Netzwerkanbieter.

Tab. 32: ST06 - Gerät bei Netzwerkanbieter mit OTAA registrieren

ST07 - Gerät bei Netzwerkanbieter mit ABP registrieren

Use Case	UC05
Testziel	Ein Gerät wird beim Netzwerkanbieter registriert
Vorbedingungen	• Benutzer ist eingeloggt • Netzwerkanbieter und benötigte Konfigurationen sind hinterlegt
Ausführung	1. Gerät erstellen 2. Netzwerkanbieter auswählen 3. Als Registrationsart ABP wählen und entsprechende Informationen angebeben 4. Speichern
Erwartetes Ergebnis	Deviceübersichtsseite wird angezeigt und Gerät wurde registriert. Entsprechenden Eintrag findet man beim Netzwerkanbieter.

Tab. 33: ST07 - Gerät bei Netzwerkanbieter mit ABP registrieren

ST08 - CRUD Devicegruppe

Use Case	UC06
Testziel	Ein Gerätgruppe wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Gerätgruppe erstellen 2. Gerätgruppe in Geräteübersicht eingesehen 3. Gerätgruppe editieren und speichern 4. Änderungen in Geräteübersicht eingesehen 5. Gerätgruppe löschen
Erwartetes Ergebnis	Gerätgruppe wird erstellt, kann eingesehen, aktualisiert und gelöscht werden.

Tab. 34: ST08 - CRUD Devicegruppe

ST09 - Gerät einer Gerätegruppe hinzufügen

Use Case	UC06
Testziel	Ein Gerät kann einer Gerätegruppe hinzugefügt werden
Vorbedingungen	• Benutzer ist eingeloggt • Gerätegruppe erstellt
Ausführung	1. Gerätgruppenübersicht öffnen 2. Neues Gerät erstellen
Erwartetes Ergebnis	Neues Gerät befindet sich in Gerätegruppe.

Tab. 35: ST09 - Gerät einer Gerätegruppe hinzufügen

ST10 - Downlink senden

Use Case	UC07
Testziel	Einem Gerät wird ein Downlink mit entsprechenden Daten geschickt
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert
Ausführung	1. Geräteübersicht öffnen 2. Neuen Downlink senden 3. Entsprechende Daten angeben
Erwartetes Ergebnis	Gerät erhält Downlink und kann beim Netzwerkanbieter eingesehen werden

Tab. 36: ST10 - Downlink senden

ST11 - Geräteprobleme definieren

Use Case	UC08
Testziel	Alarmfunktionen können vom Benutzer in Form von Pythoncode einer Devicegruppe hinterlegt werden, welche bei jedem Uplink eines Gerätes geprüft werden
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Alert-Configuration Übersicht öffnen 2. Neuen Alert erstellen
Erwartetes Ergebnis	Alert ist in Übersicht vorhanden und kann bei einer Devicegruppe ausgewählt werden

Tab. 37: ST11 - Geräteprobleme definieren

ST12 - Geräteprobleme erkennen und anzeigen

Use Case	UC08
Testziel	Uplinks werden mit hinterlegten Alarmfunktionen geprüft und allfällige Fehlermeldungen rapportiert
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert • Alert-Configuration, welche sicher eine Fehlermeldung ausgibt, ist bei Gerätgruppe, in der sich das Gerät befindet, hinterlegt
Ausführung	1. Uplink von einem Endgerät aus initiieren 2. Geräteübersicht öffnen 3. Bei gewähltem Gerät Alertmeldungen öffnen
Erwartetes Ergebnis	Alert ist in Alertmeldungen vorhanden

Tab. 38: ST12 - Geräteprobleme erkennen und anzeigen

ST13 - Einzelne Paketinformationen einsehen

Use Case	UC10
Testziel	Ein ankommendes Uplinkpaket und dessen geparster Payload wird angezeigt
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert • Parser wurde dem Gerät hinterlegt • Uplink wurde für das Gerät erhalten
Ausführung	1. Uplinkübersicht öffnen
Erwartetes Ergebnis	Die Uplinkdaten sind dargestellt und der Payload wird geparst angezeigt

Tab. 39: ST13 - Einzelne Paketinformationen einsehen

ST14 - Gesamte Paketeübersicht einsehen

Use Case	UC11
Testziel	Die ankommenden Uplinkpakete werden in einer Übersicht dargestellt
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert • Uplinks wurden für das Gerät erhalten
Ausführung	1. Uplinkübersicht öffnen
Erwartetes Ergebnis	Eine Liste aller Uplinks von allen Geräten ist einsehbar

Tab. 40: ST14 - Gesamte Paketeübersicht einsehen

ST15 - Paketeübersicht pro Device einsehen

Use Case	UC11
Testziel	Die ankommenden Uplinkpakete werden in einer Übersicht für ein einzelnes Gerät dargestellt
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert • Uplinks wurden für das Gerät erhalten
Ausführung	1. Uplinkübersicht öffnen 2. DevEUI anklicken
Erwartetes Ergebnis	Eine Liste aller Uplinks vom ausgewählten Geräte ist einsehbar

Tab. 41: ST15 - Paketeübersicht pro Device einsehen

ST16 - Paketeübersicht pro Device einsehen

Use Case	UC11
Testziel	Die ankommenden Uplinkpakete werden in einer Übersicht für ein einzelnes Gerät dargestellt
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert • Uplinks wurden für das Gerät erhalten
Ausführung	1. Uplinkübersicht öffnen 2. DevEUI anklicken
Erwartetes Ergebnis	Eine Liste aller Uplinks vom ausgewählten Geräte ist einsehbar

Tab. 42: ST16 - Paketeübersicht pro Device einsehen

ST17 - Paketeübersicht filtern

Use Case	UC11
Testziel	Die Uplinkübersicht kann mit einem Filter reduziert werden
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert • Uplinks wurden für das Gerät erhalten
Ausführung	1. Uplinkübersicht öffnen
Erwartetes Ergebnis	Die Uplinkübersicht lässt sich mit dem Filter einschränken

Tab. 43: ST17 - Paketeübersicht filtern

ST18 - CRUD Application Server

Use Case	UC12
Testziel	Ein Application Server wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Application Server erstellen 2. Application Server in Application Server übersicht eingesehen 3. Application Server editieren und speichern 4. Änderungen in Application Server Übersicht einsehen 5. Application Server löschen
Erwartetes Ergebnis	Application Server wird erstellt, kann eingesehen, aktualisiert und gelöscht werden.

Tab. 44: ST18 - CRUD Application Server

ST19 - HTTP-Post Output bei Application Server hinterlegen

Use Case	UC12
Testziel	Ein Gerätgruppe wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert • Uplinks wurden für das Gerät erhalten
Ausführung	1. Application Server erstellen 2. Neuen Post Output mit entsprechender URL und Secret definieren 3. Gerät mit Application Server hinterlegen
Erwartetes Ergebnis	Die definierte URL erhält die Uplinkpakete als JSON von der Plattform

Tab. 45: ST19 - HTTP-Post Output bei Application Server hinterlegen

ST20 - Influx-DB Output bei Application Server hinterlegen

Use Case	UC12
Testziel	Ein Gerätgruppe wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt • Gerät wurde erstellt und ist beim Netzwerkanbieter registriert • Uplinks wurden für das Gerät erhalten
Ausführung	1. Application Server erstellen 2. Neuen Influx-DB Output mit entsprechender URL und Secret definieren 3. Gerät mit Application Server hinterlegen
Erwartetes Ergebnis	Die definierte URL erhält die Uplinkpakete als spezielles Influx-DB JSON von der Plattform.

Tab. 46: ST20 - Influx-DB Output bei Application Server hinterlegen

ST21 - API Zugriff mit Application Server Token

Use Case	UC12
Testziel	Request auf API inkl. Token ist erfolgreich
Vorbedingungen	• Application Server ist erstellt • Generiertes Bearer Token bekannt
Ausführung	1. Request inkl. Token im Request Header an beliebigen Endpunkt senden
Erwartetes Ergebnis	Erfolgreicher Request inkl. Erhalt der definierten Response in API Beschreibung

Tab. 47: ST21 - API Zugriff mit Application Server Token

ST22 - CRUD Firmware

Use Case	UC13
Testziel	Ein Firmware wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Firmware erstellen 2. Firmware in Devicetyp-Übersicht eingesehen 3. Firmware editieren und speichern 4. Änderungen in Firmware-Übersicht einsehen 5. Firmware löschen
Erwartetes Ergebnis	Firmware wird erstellt, kann eingesehen, aktualisiert und gelöscht werden.

Tab. 48: ST22 - CRUD Firmware

ST23 - CRUD Devicetyp

Use Case	UC13
Testziel	Ein Devicetyp wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Devicetyp erstellen inkl. Firmware 2. Devicetyp in Devicetyp-Übersicht eingesehen 3. Devicetyp editieren und speichern 4. Änderungen in Devicetyp-Übersicht einsehen 5. Devicetyp löschen
Erwartetes Ergebnis	Devicetyp wird erstellt, kann eingesehen, aktualisiert und gelöscht werden.

Tab. 49: ST23 - CRUD Devicetyp

ST24 - CRUD Parser

Use Case	UC14
Testziel	Ein Parser wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Parser erstellen 2. Parser in Parser-Übersicht eingesehen 3. Parser editieren und speichern 4. Änderungen in Parser-Übersicht einsehen 5. Parser löschen
Erwartetes Ergebnis	Parser wird erstellt, kann eingesehen, aktualisiert und gelöscht werden.

Tab. 50: ST24 - CRUD Parser

ST25 - Parser pro Port einem Devicetyp hinterlegen

Use Case	UC14
Testziel	Einem Devicetyp kann pro Port ein Parser hinterlegt werden
Vorbedingungen	• Benutzer ist eingeloggt • Parser erstellt • Devicetyp erstellt • Gerät wurde erstellt inkl. Devicetyp und ist beim Netzwerkanbieter registriert • Uplinks wurden für das Gerät erhalten
Ausführung	1. Devicetyp-Übersicht öffnen 2. Devicetyp editieren 3. Neuen Parser inkl. Port hinzufügen 4. Uplinkübersicht darstellen
Erwartetes Ergebnis	Daten werden in der Uplinkübersicht mit dem hinterlegten Parsercode encodiert dargestellt

Tab. 51: ST25 - Parser pro Port einem Devicetyp hinterlegen

ST26 - CRUD Netzwerkanbieter

Use Case	UC14
Testziel	Ein Netzwerkanbieter wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Netzwerkanbieter erstellen 2. Netzwerkanbieter in Netzwerkanbieter-Übersicht eingesehen 3. Netzwerkanbieter editieren und speichern 4. Änderungen in Netzwerkanbieter-Übersicht einsehen 5. Netzwerkanbieter löschen
Erwartetes Ergebnis	Netzwerkanbieter wird erstellt, kann eingesehen, aktualisiert und gelöscht werden.

Tab. 52: ST26 - CRUD Netzwerkanbieter

ST27 - Netzwerkanbieterkommunikation steht

Use Case	UC15
Testziel	Die hinterlegten Konfigurationen des Netzwerkproviders funktionieren
Vorbedingungen	• Benutzer ist eingeloggt • Netzwerkanbieter erstellt
Ausführung	1. Neues Gerät erstellen
Erwartetes Ergebnis	Gerät wird beim Netzwerkprovider registriert und bei ihm angezeigt

Tab. 53: ST27 - Netzwerkanbieterkommunikation steht

ST28 - CRUD Account

Use Case	UC16
Testziel	Ein Account wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist eingeloggt
Ausführung	1. Account erstellen 2. Account in Netzwerkanbieter-Übersicht eingesehen 3. Account editieren und speichern 4. Änderungen in Account-Übersicht einsehen 5. Account löschen
Erwartetes Ergebnis	Account wird erstellt, kann eingesehen, aktualisiert und gelöscht werden.

Tab. 54: ST28 - CRUD Account

ST29 - Login mit Account

Use Case	UC16
Testziel	Ein Account wird erstellt, angezeigt, aktualisiert und gelöscht
Vorbedingungen	• Benutzer ist erstellt
Ausführung	1. Loginformular öffnen 2. Daten eingeben
Erwartetes Ergebnis	Login erfolgreich, Benutzer erhält Token und wird zu Uplinkübersicht geleitet

Tab. 55: ST29 - Login mit Account

ST30 - API-Request ohne Token

Use Case	NFR: Security
Testziel	Nicht authentifizierter Request wird verweigert
Vorbedingungen	• -
Ausführung	1. GET-Request um alle Devices zu erhalten, ausführen
Erwartetes Ergebnis	Request wird verweigert, Benutzer erhält Fehlercode 401

Tab. 56: ST30 - API-Request ohne Token

ST31 - API-Request mit Token

Use Case	NFR: Security
Testziel	Authentifizierter Request wird ausgeführt
Vorbedingungen	• Account ist verfügbar
Ausführung	1. POST-Request mit Accountdaten auf /auth ausführen 2. Erhaltenes Token bei neuem Request als Bearer Token hinterlegen 3. GET-Request um alle Devices zu erhalten, ausführen
Erwartetes Ergebnis	Token wird bei erstem Request erhalten. Der nachfolgende Request wird ausgeführt, Benutzer erhält Deviceübersicht als JSON

Tab. 57: ST31 - API-Request mit Token

B.2 Testprotokoll

Datum	06.06.2019
Testumgebung	Windows 10, Alles neu aufgesetzt, Testdaten hinterlegt
Testperson	Raffael Vögeli

Systemtest	**Erfüllt**	**Systemtest**	**Erfüllt**
ST01	Ja	ST17	Nein
ST02	Ja	ST18	Ja
ST03	Ja	ST19	Ja
ST04	Ja	ST20	Ja
ST05	Ja	ST21	Ja
ST06	Ja	ST22	Ja
ST07	Ja	ST23	Ja
ST08	Ja	ST24	Ja
ST09	Ja	ST25	Ja
ST10	Ja	ST26	Ja
ST11	Ja	ST27	Ja
ST12	Ja	ST28	Ja
ST13	Ja	ST29	Ja
ST14	Ja	ST30	Ja
ST15	Ja	ST31	Ja
ST16	Ja		

Tab. 58: System-Testprotokoll: 06.06.2019

B.2.1 Bemerkungen zu den Tests

ST17 schlug fehl, da das Feature gegenüber anderen zusätzlichen Funktionen niedriger priorisiert und somit nicht umgesetzt wurde. Das Problem ist bereits erfasst und dem Industriepartner auch bekannt.

C Installation

C.1 Per Docker

Die einzelnen Services werden als Docker Container erstellt und im jeweiligen GitLab Projekt in der Container Registry bereitgestellt. Die einzelnen Services können also einfach als Container heruntergeladen und laufen gelassen werden.

Um das aber noch einfacher zu machen, haben wir Docker-Compose eingesetzt. In einem File sind alle Services angegeben welche benötigt werden (inkl. Datenbank) und die Konfigurationen sind bereits vorgenommen worden.

Mit dieser Installation werden alle Services auf einem einzelnen Host installiert.

Zuerst muss auf dem Linux Server Docker sowie Docker-Compose installiert werden. Danach muss man über die Konsole die GitLab Registry bei Docker registrieren, nur damit können die vorhandenen Docker Container heruntergeladen werden: docker login registry.gitlab.com

Wenn man sich das Docker Compose File[15] heruntergeladen hat, kann man die Applikation ganz einfach mit folgendem Befehl starten: docker-compose up -d

Beim ersten Login kann man sich mit dem Benutzernamen "admin" und dem Passwort "123456" einloggen (was natürlich sofort geändert werden sollte).

C.2 In der Cloud

Die Installation der Applikation in der Cloud lag ausserhalb des Umfanges unserer Arbeit.

Aber da die einzelnen Services als Container bereitstehen, können diese natürlich auch in der Cloud gestartet werden. Die einzelnen Services wurde so konzipiert, dass mehrere Instanzen eines Services gestartet werden können.

Beim deployen bei einem Cloud-Anbieter, kann Traefik natürlich auch mit dem Loadbalancer des Anbieters ersetzt werden.

C.3 Sicherheit

Mindestens bei der Installation für die Produktions-Umgebung muss darauf geachtet werden, dass das Backend nur verschüsselt erreichbar ist.

Dies kann erreicht werden, indem Traefik so konfiguriert wird, dass er nur noch verschlüsselte Verbindungen annimmt. Traefik kann auch automatisch SSL Zertifikate von "Let's Encrypt" beziehen[16].

Auch ist darauf zu achten, dass ausgehende Verbindungen, zum Beispiel vom Post Output Service, nur an Verschlüsselte Endpunkte geschehen. Es werden dabei alle SSL Zertifikate akzeptiert, wessen Root CAs im default Truststore von Java enthalten sind.

[15] `https://gitlab.com/lorawan-management-platform/docker-compose`
[16] `https://docs.traefik.io/user-guide/docker-and-lets-encrypt/`

C.4 Entwicklungsumgebung

Zur Entwicklung wurde der IntelliJ IDEA von Jetbrains verwendet. Dieser sollte installiert werden, sowie die Java Version 11.

Datenbank Wir empfehlen die Datenbank als Docker Container zu starten. Am besten klont man dazu unser Docker-Compose Order: git clone git@gitlab.com:lorawan-management-platform/docker-compose.git

Darin kann man nun per Docker-Compose die Datenbank starten: docker-compose up -d lmp-mongo

Die Datenbank ist nun auf Port 40000 erreichbar, mit dem Benutzernamen "root" und Passwort "R48jp4VCUjFrgvrh". Diese Informationen findet man auch im docker-compose.yml File.

Backend Der Code der einzelnen Services muss aus den Git-Repositories geklont werden. Zum Beispiel: git clone git@gitlab.com:lorawan-management-platform/webservice.git

Diese Projekte können nun jeweils im IntelliJ IDEA geöffnet werden. Die Dependencies können per Maven bezogen werden. Dazu kann im IDEA Rechtsklick auf das File "pom.xml" gemacht werden und im Kontextmenü unter "Maven" die Packages importiert werden.

In der Datei "src/main/resources/application.properties" können einige Einstellungen vorgenommen werden. Zum einen sollte hier jeweils die Datenbankverbindung hinterlegt werden. Diese Einstellungen beginnen mit "spring.data.mongodb". Zum anderen kann man hier auch den Port definieren, auf welchem der Service läuft. Startet man mehrere Services ist es nötig, dass jeder einen unterschiedlichen Port hat, da die Services ansonsten nicht starten. Den Port definiert man mit: server.port = 9092

Man definiert hier auch unter "eureka.client.serviceUrl.defaultZone" die URL zum Eureka Server.

Die Services können nun einfach mit einem Klick auf den Play-Button im IntelliJ IDEA gestartet werden.

Parsing Service Eine Ausnahme bildet der Parsing Service. Dieser ist ein NodeJS Projekt.

Es muss NodeJS sowie Python3 installiert werden.

Nach dem Klonen des Parsing Services aus dem Git-Repository, müssen im Ordner "shared" die Abhängigkeiten installiert werden: npm install

Um den Service zu starten führt man folgenden Befehl aus: npm run start

Falls man Konfigurationen vornehmen will, kann man folgende Environment-Variablen setzen:

Variable	Default	Beschreibung
PORT	80	Unter welchem Port der Service startet
EUREKA_HOST	"lmp-eureka-server"	Hostname (oder IP) unter welchem Eureka erreichbar ist
EUREKA_PORT	8761	Port von Eureka
PYTHON_EXEC	"python3"	Der Name des Python Interpreters

Frontend Voraussetzungen sind, dass man bereits NodeJS installiert hat. Auch empfehlen wir den WebStorm IDEA von Jetbrains zu nutzen.

Nachdem man das Projekt geklont hat, öffnet man es im WebStorm. Mit Rechtsklick auf "package.json" öffnet sich das Kontextmenü, indem man "run npm install" auswählen kann um alle Abhängigkeiten zu installieren.

Im selben Kontextmenü findet sich auch der Befehl "show npm Scripts". Klickt man dies an, öffnet sich ein neues Fenster indem man "serve" anklicken kann um das Frontend zu starten.

Die Einstellungen findet man in der Datei ".env". Man kann dieses kopieren mit dem Namen ".env.local" um die Einstellungen zu überschreiben. Darin gibt es die Variable "VUE_APP_API_URL", unter welcher man die URL zur Backend-API definieren muss. Bei Änderungen muss das Frontend wie oben beschrieben neu gestartet werden.

D Anleitung für die Weiterentwicklung

Die Anleitungen richten sich Entwickler, welche die Plattform ausbauen möchten. Analog zu den Codebeispielen in den nachfolgenden Kapiteln, sollten auch die bereits umgesetzten Projekte als Referenz bzw. Hilfestellung genommen werden.

D.1 Netzwerkanbieter hinzufügen

Um einen neuen Netzwerkanbieter hinzuzufügen, müssen die nachfolgenden Punkte umgesetzt werden. **Hinweis:** *NetProv* muss jeweils mit dem neuen Network-Providernamen (Bsp. NetProv = Swisscom -> SwisscomProvider) ausgetauscht werden.

D.1.1 Neues ProviderService-Projekt erstellen

Analog zu den bereits umgesetzten Netzwerkanbietern Swisscom und Loriot muss ein neues Spring Boot Projekt erstellt werden. Der Name des Projekts soll gemäss den bisherigen Implementationen *NetProv*Service lauten (Bsp. SwisscomProviderService). Der Code des Projekts muss zusätzlich auf GitLab innerhalb des Lorawan-Management-Platform Projektes [17] integriert werden. Danach muss das Git-Submodule LMP-Common integriert werden. Dies kann mit dem Befehl

```
git submodule add ../lmp-common.git src/main/java/ch/hsr/common
```

ausgeführt werden.

Folgende Grundstruktur soll das Projekt besitzen:

[17] `https://gitlab.com/lorawan-management-platform`

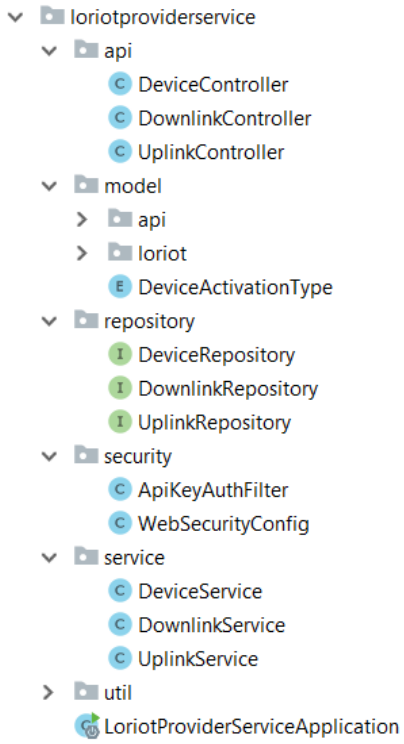


Abb. 62: Benutzeranleitung Netzwerkanbieter: Projektstruktur

D.1.2 Endpunkte implementieren

Für die Operationen Downlink senden, Uplink empfangen und Device registrieren/aktualisieren/deregistrieren müssen Schnittstellen geschaffen werden.

Listing 3: Template NetworkProvider: DownlinkController

```
@RestController
@RequestMapping("/downlink")
public class DownlinkController {
    @Autowired
    DownlinkService downlinkService;

    @PostMapping
    public void sendDownlinkToDevice(@Valid $$@RequestBody DownlinkDto
        downlink) {
         downlinkService.sendDownlink(downlink);
    }
}
```

Listing 4: Template NetworkProvider: Uplinkcontroller

```
@RestController
@RequestMapping("/uplink/")
public class UplinkController {
```

```
5       @Autowired
6       private UplinkService uplinkService;

8       @PostMapping
9       public void receiveUplink(@RequestBody LoriotUplink loriotUplink) {
10          uplinkService.receiveUplink(loriotUplink);
11      }
12  }
```

Listing 5: Template NetworkProvider: Devicecontroller

```
1   @RestController
2   @RequestMapping(value = "/device")
3   public class DeviceController {
4       @Autowired
5       DeviceService deviceService;

7       @PostMapping
8       public ResponseEntity registerDevice(@Valid $$@RequestBody DeviceDto
            device) {
9           deviceService.registerDevice(device);
10          return ResponseEntity.ok("Device registered");
11      }

13      @PutMapping(path = "/{deveui}")
14      public ResponseEntity updateDevice(@PathVariable("deveui")String
            deveui) {
15          deviceService.updateDevice(deveui);
16          return ResponseEntity.ok("Device updated");
17      }

19      @DeleteMapping(path = "/{deveui}")
20      public ResponseEntity deregisterDevice(@PathVariable("deveui")String
            deveui) {
21          deviceService.deregisterDevice(deveui);
22          return ResponseEntity.ok("Device deregistered");
23      }
24  }
```

D.1.3 Services implementieren

Zu den entsprechenden Controller-Endpunkten müssen nun noch die Services erstellt werden, welche referenziert sind. Die Implementierung der Methoden ist Netzwerkanbieter-Spezifisch.

Listing 6: Template NetworkProvider: Downlinkservice

```
1   @Service
2   public class DownlinkService {
3       @Autowired
4       private DownlinkRepository downlinkRepository;
```

```
5       @Autowired
6       private DeviceRepository deviceRepository;

8       public void sendDownlink(DownlinkDto downlinkDto) {
9           // TODO: Implement network provider logic
10      }
11  }
```

Listing 7: Template NetworkProvider: Uplinkservice

```
1   @Service
2   public class UplinkService {
3       @Autowired
4       private UplinkRepository uplinkRepository;
5       @Autowired
6       private DeviceRepository deviceRepository;
7       @Autowired
8       private ExecutorService executorService;
9       @Autowired
10      PostServiceProvider postServiceProvider;
11      @Autowired
12      InfluxServiceProvider influxServiceProvider;
13      @Autowired
14      ParsingServiceProvider parsingServiceProvider;
15      @Autowired
16      AlertingServiceProvider alertingServiceProvider;

18      public void receiveUplink(NetProvUplink netProvUplink) {
19          DeviceEntity device =
                deviceRepository.findByDeveui(netProvUplink.DevEUI);
20          UplinkEntity uplink = netProvUplink.generateUplink();
21          uplink.set_id(ObjectId.get());

23          if (device != null) {
24              // TODO: Implement parsing and alerting of device and
                    notifyClientApplications
25          } else {
26              // TODO: Save uplink to database if device is not known by
                    platform
27          }
28      }

30      private void notifyClientApplication(DeviceEntity device, UplinkEntity
            uplink) {
31          // TODO: Implement Notification of client applications
32      }
33  }
```

Listing 8: Template NetworkProvider: Deviceservice

```
1   @Service
2   public class DeviceService {
3       @Autowired
```

```
4       DeviceRepository deviceRepository;

6       public void registerDevice(DeviceDto deviceDto) {
7           // TODO: Register Device on network provider
8       }

10      public String deregisterDevice(String deveui) {
11          // TODO: Deregistration of device on network provider
12      }

14      public String updateDevice(String deveui) {
15          // TODO: Update device information on network provider
16      }
17  }
```

D.1.4 FeignClient erstellen

Nachdem der Netzwerk Provider Service implementiert wurde, muss für den Gebrauch des Services ein neuer FeignClient im Common-Projekt hinterlegt werden.

Listing 9: Template NetworkProvider: Feign Client

```
1  @Service
2  @FeignClient(name = "netProv-provider-service", configuration =
       NetProvService.NetProvFeignClientConfiguration.class)
3  public interface NetProvProviderService {
4      class NetProvFeignClientConfiguration {
5          @Bean
6          public BearerTokenRequestInterceptor
               bearerHeaderAuthRequestInterceptor(@Value("${lmp.netprov.apikey}")
               String apiKey) {
7              return new BearerTokenRequestInterceptor(apiKey);
8          }
9      }

11     @RequestMapping(
12             method= RequestMethod.POST,
13             value="/downlink",
14             consumes="application/json")
15     void postDownlink(DownlinkDto downlinkDto);

17     @RequestMapping(
18             method= RequestMethod.POST,
19             value="/device",
20             consumes="application/json")
21     ResponseEntity registerDevice(DeviceRegisterDto device);

23     @RequestMapping(
24             method= RequestMethod.PUT,
25             value="/device/{deveui}",
26             consumes="application/json")
27     ResponseEntity updateDevice(@PathVariable("deveui") String deveui);
```

```
29      @RequestMapping(
30              method = RequestMethod.DELETE,
31              value = "/device/{deveui}",
32              consumes = "application/json")
33      ResponseEntity deregisterDevice(@PathVariable("deveui") String deveui);
34  }
```

Zudem muss die Hauptmethode des neuen Services um die folgenden beiden Annotations ergänzt werden

```
@EnableEurekaClient
@EnableFeignClients(basePackages = "ch.hsr.common.services")
```

D.1.5 NetworkProvider in Datenbank hinterlegen

Für jeden NetworkProvider gibt es unterschiedliche Konfigurationen, welche hinterlegt werden müssen (Bsp: API-Key, URL,Benutzername, Password, etc.), um mit der externen Schnittstelle kommunizieren zu können. Diese werden auf der Datenbank in der Collection NetworkProvider hinterlegt. Im LMP-Common Projekt muss somit eine neue Entity mit dem Namen ProvNameConfigurationEntity und dessen spezifischen Feldern erstellt werden. Diese werden beim CRUD des Netzwerkanbieters entsprechend befüllt.

D.1.6 Webservice anpassen

Einerseits müssen die CRUD Operationen für die Networkprovider ausgebaut werden. Entsprechend soll die neue Konfiguration auch in der Datenbank hinterlegt werden können. In der Klasse NetworkProviderService müssen die folgenden Methoden ergänzt werden:

- createNetworkProvider
- updateNetworkProvider

Andereseits muss der Netzwerkprovider noch in Einsatz gebracht werden. Dazu muss die Klasse DeviceService im WebService angepasst werden. Einerseits muss der FeignClient des neuen Networkproviders als Bean hinterlegt werden:

```
@Autowired
private NetProvService netProvService;
```

Zusätzlich befinden sich in den folgenden Methoden FeignClient Aufrufe, welche mit dem neuen Service ergänzt werden muss:

- sendDownlink

- updateDevice
- deleteDevice
- registerDeviceOnProvider
- deregisterDeviceOnProvider

Die Codestruktur, die verändert werden muss, sieht dabei wie folgt aus:

Listing 10: Template NetworkProvider: Method-Call in webservice example

```
1 if(deviceEntity.getNetworkProvider().getSwisscomConfiguration() != null) {
2     swisscomProviderService.updateDevice(deviceEntity.getDeveui());
3 } else if(deviceEntity.getNetworkProvider().getLoriotConfiguration() !=
    null) {
4     loriotProviderService.updateDevice(deviceEntity.getDeveui());
5 } // TODO: Add new feign client network provider call
```

D.1.7 Frontend für neuen Netzwerkanbieter ausbauen

Analog den bisherigen Netzwerkanbieter CRUD Funktionen im Frontend, müssen die spezifischen Eigenschaften des neuen Netzwerkanbieters editierbar gemacht werden. Vom Frontend wird ein entsprechendes Configuration-Object als JSON mitgesendet. Am Besten wird hierbei der bereits umgesetzte Code eingesehen, um es umzusetzen.

D.2 Output-Service hinzufügen

Die Plattform kann auch um neue Output-Möglichkeiten ausgebaut werden, um die Client Applikationen über Uplinks resp. Alerts zu informieren.

Hinweis: *OutputType* muss jeweils mit dem neuen Output-Typ (Bsp. OutputType = Post -> PostOutputProvider) ausgetauscht werden.

D.2.1 Neues OutputService-Projekt erstellen

Als Referenz können die bereits umgesetzten OutputService-Projekte (z.B. PostOutpuProvider) zu Hilfe gezogen werden. Die folgende Beschreibung bezieht sich auf eine Umsetzung mit einem Spring-Boot Projekt. Der Name des Projekts soll gemäss den bisherigen Implementationen *OutputType*OutputService lauten (Bsp. PostOutputService). Der Code des Projekts muss zusätzlich auf GitLab innerhalb des Lorawan-Management-Platform Projektes [18] integriert werden. Danach muss das Git-Submodule LMP-Common integriert werden. Dies kann mit dem Befehl

```
git submodule add ../lmp-common.git src/main/java/ch/hsr/common
```

ausgeführt werden.

Folgende Projektstruktur sollte man berücksichtigen:

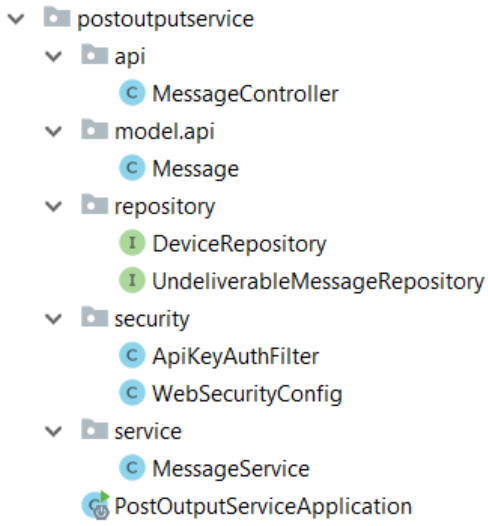


Abb. 63: Benutzeranleitung OutputService: Projektstruktur

D.2.2 Endpunkte implementieren

Je nach Output sehen die Endpunkte unterschiedlich aus. Damit ein entsprechender Output der Daten initiiert werden kann, muss die folgende Controller-Schnittstelle umgesetzt werden:

[18] `https://gitlab.com/lorawan-management-platform`

Listing 11: Template OutputService: MessageController

```
@RestController
@RequestMapping("/message/")
public class MessageController {

    @Autowired
    private MessageService messageService;

    @PostMapping
    public void sendMessage(@RequestBody Message message) {
        messageService.relayMessage(message);
    }
}
```

D.2.3 Service implementieren

Zum entsprechenden Controller-Endpunkt muss noch der Service erstellt werden, welcher mit dem Repository kommuniziert und den Output ausführt.

Listing 12: Template OutputService: MessageService

```
@Service
public class MessageService {
    @Autowired
    private DeviceRepository deviceRepository;
    @Autowired
    private ExecutorService executorService;

    public void relayMessage(Message message) {
        // TODO: implement output logic. Use executorService to send
           outputs async
    }
}
```

D.2.4 FeignClient erstellen

Nachdem der OutputService implementiert wurde, muss für den Gebrauch des Services ein neuer FeignClient im Common-Projekt hinterlegt werden.

Listing 13: Template OutputService: Feign Client

```
@Service
@FeignClient(name = "OutputType-output-service", configuration =
    OutputTypeServiceProvider.OutputTypeServiceFeignClientConfiguration.class)
public interface OutputTypeServiceProvider {
    class OutputTypeServiceFeignClientConfiguration {
```

```
5           @Bean
6           public BearerTokenRequestInterceptor
                bearerHeaderAuthRequestInterceptor(@Value("${...}") String
                apiKey) {
7               return new BearerTokenRequestInterceptor(apiKey);
8           }
9       }

11      @RequestMapping(
12              method= RequestMethod.POST,
13              value="/message/",
14              consumes="application/json")
15      String sendMessage(OutputTypeMessageDto OutputTypeMessageDto);
16  }
```

Zudem muss die Hauptmethode des neuen Services um die folgenden beiden Annotations ergänzt werden:

```
@EnableEurekaClient
@EnableFeignClients(basePackages = "ch.hsr.common.services")
```

D.2.5 OutputService-Konfiguration in Datenbank hinterlegen

Eventuell ist es nötig, dass einer ClientApplikation Eigenschaften bezüglich dem neuen OutputService hinterlegt werden müssen (z.B. URL, OutputTyp, etc.). Dabei muss eine neue OutputConfigurationEntity im LMP-Common Projekt hinterlegt werden, welche dann die Konfigurationen mit der Datenbank teilen kann. Wichtig ist, dass alle hinterlegten Client Applikationen dieses Configuration-Objekt hinterlegt haben (auch wenn als Wert null hinterlegt ist).

D.2.6 ProviderServices & AlertingService anpassen

Damit der neue OutputService auch angestossen wird, müssen die bereits im Einsatz stehenden ProviderServices ergänzt werden. In der Klasse **UplinkService** muss die Methode **notifyClientApplication** ergänzt werden.

Ebenfalls sollen die Alerts mit Hilfe des neuen OutputServices an die Client Applikationen gebracht werden. Dazu muss in der Klasse **AlertService** die Methode **executeAlert** um die nötigen Instruktionen ergänzt werden.

D.2.7 Frontend aktualisieren

Um die Konfigurationen auf den Client Applikationen hinterlegen zu können, sind einige Ergänzungen notwendig. Diese können analog der bisher umgesetzten OutputService-Konfigurationen auf den CRUD ClientApplication Views angepasst werden.

E API Spezifikation

LoRaWAN Management Platform

API and SDK Documentation

Version: 1.0.0-oas3

LoRaWAN Management Platform API Description

User

addAlertConfiguration

Adds an alert configuration

Adds an alert configuration to the system

POST

/alert-config

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//alert-config"
```

Parameters

Body parameters

Name	Description
body	▾ { [] _id: ▾ [] string *ObjectID* name: string code: ▾ [] string *Python code to execute on every uplink from device* }

Responses

Status: 201 - alert configuration added

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 409 - an existing item already exists

addClientApplication

Adds a client application

Adds a client application to the system

POST

/clientapp

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//clientapp"
```

Parameters

Body parameters

Name	Description
body	▾ { [] Required: _id,name,url _id: ▾ [] string *ObjectID* name: string apitoken: string influxoutput: ▾ [[] ▾ { [] url: string dbname: ▾ [] string *Database name* measurement: ▾ [] string *Table name* username: ▾ [] string *database username* password: ▾ [] string *database password* }] postoutputuris: ▾ [[] ▾ { [] url: string secret: ▾ [] string *Pre-shared secret to authenticate* }] }

Responses

Status: 201 - client application added

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 409 - an existing item already exists

addDevice

Adds a device

Adds an device to the system

POST

/device

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//device"
```

Parameters

Body parameters

Name	Description
body	▾ { [] Required: deveui,name deveui: string name: string account: ▾ [] string *ObjectID* clientapplication: ▾ [] string *ObjectID* networkprovider: ▾ [] string *ObjectID* devicetype: ▾ [] string *ObjectID* devicegroup: ▾ [] string *ObjectID* deviceclass: ▾ [] string *A or C* connectivityplan: ▾ [] string *in case of swisscom: connectivity plan id, else: null* otaaregistration: ▾ { [] applicationeui: string applicationkey: string } abpregistration: ▾ { [] networkaddress: string networksessionkey: string applicationsessionkey: string } }

Responses

Status: 201 - device created

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 409 - an existing item already exists

addDeviceGroup

Adds a device group

Adds a device group to the system

POST

/devicegroup

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//devicegroup"
```

Parameters

Body parameters

Name	Description
body	▾ { [] Required: name name: string devicegroup: string alerts: ▾ [[] ▾ { [] _id: ▾ [] string *ObjectID* text: string date: string }] }

Responses

Status: 201 - Devicegroup successfully created

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 409 - an existing device group already exists

addDeviceType

Adds a device type

Adds a device type to the system

POST

/devicetype

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//devicetype"
```

Parameters

Body parameters

Name	Description
body	▾ { [] Required: _id,name _id: string name: string firmware: ▾ [] string *ObjectID* parserPerPortMap: ▾ { [] } }

Responses

Status: 201 - device type created

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 409 - an existing device type already exists

addFirmware

Adds a firmware

Adds a firmware to the system

POST

/firmware

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//firmware"
```

Parameters

Body parameters

Name	Description
body	▾ { [] _id: ▾ [] string *ObjectID* versionname: ▾ [] string *name of the firmware version* }

Responses

Status: 201 - firmware added

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 409 - an existing item already exists

addNetworkProvider

Adds a network provider

Adds a network provider to the system

POST

/networkprovider

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//networkprovider"
```

Parameters

Body parameters

Name	Description
body	▾ { [] Required: id,name id: integer name: string swisscomConfiguration: ▾ { [] clientprofile: ▸ [] string clientid: ▸ [] string clientsecret: ▸ [] string asid: ▸ [] string assecret: ▸ [] string appserviceroutingprofile: ▸ [] string } loriotConfiguration: ▾ { [] baseurl: ▸ [] string apikey: ▸ [] string applicationtoken: ▸ [] string appid: ▸ [] string } }

Responses

Status: 201 - network provider added

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 409 - an existing item already exists

addParser

Adds a parser

Adds a parser to the system

POST

/parser

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//parser"
```

Parameters

Body parameters

Name	Description
body	▾ { [] _id: ▾ [] string *ObjectID* name: ▾ [] string *name of the parser* code: ▾ [] string *Parsercode written in Python* testpayload: ▾ [] string *Payload to test against the parser code* }

Responses

Status: 201 - parser added

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 409 - an existing item already exists

authenticateUser

Authenticates a user with it's username and password

Authenticates a user with it's username and password

POST

/auth

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//auth"
```

Parameters

Body parameters

Name	Description

body	▾ { [] Required: password,username username: string password: string }

Responses

Status: 200 - Authentication successful

Schema

▾ { []
token: string
}

Status: 400 - invalid input, object invalid

Status: 401 - Invalid password or username

deleteAlertConfiguration

Delete an alert configuration

Delete an alert configuration from the system

DELETE

/alert-config/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X DELETE "http://lmp.dunkel.world/api//alert-config/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *alert configuration identification* Required

Responses

Status: 200 - alert configuration deleted

Status: 401 - Access token is missing or invalid

Status: 404 - alert configuration not found

deleteClientApplication

Delete a client application

Delete a client application from the system

DELETE

/clientapp/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X DELETE "http://lmp.dunkel.world/api//clientapp/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *client application identification* Required

Responses

Status: 200 - client application deleted

Status: 401 - Access token is missing or invalid

Status: 404 - device not found

deleteDevice

Delete a device

Delete a device from the system

DELETE

/device/{deveui}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X DELETE "http://lmp.dunkel.world/api//device/{deveui}"
```

Parameters

Path parameters

Name	Description
deveui*	String *device identification* Required

Responses

Status: 200 - device deleted

Status: 401 - Access token is missing or invalid

Status: 404 - device not found

deleteDeviceGroup

Delete a device group

Delete a device group from the system

DELETE

/devicegroup/{name}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X DELETE "http://lmp.dunkel.world/api//devicegroup/{name}"
```

Parameters

Path parameters

Name	Description
name*	String *device group name* Required

Responses

Status: 200 - Device group deleted

Status: 401 - Access token is missing or invalid

Status: 404 - device group not found

deleteDeviceType

Delete a device type

Delete a device type from the system

DELETE

/devicetype/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X DELETE "http://lmp.dunkel.world/api//devicetype/{id}"
```

Parameters

Path parameters

Name	Description
id*	Integer *device type identification* Required

Responses

Status: 200 - Device type deleted

Status: 401 - Access token is missing or invalid

Status: 404 - device type not found

deleteFirmware

Delete a firmware

Delete a firmware from the system

DELETE

/firmware/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X DELETE "http://lmp.dunkel.world/api//firmware/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *firmware identification* Required

Responses

Status: 200 - firmware deleted

Status: 401 - Access token is missing or invalid

Status: 404 - firmware not found

deleteNetworkProvider

Delete a network provider

Delete a network provider from the system

DELETE

/networkprovider/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X DELETE "http://lmp.dunkel.world/api//networkprovider/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *network provider identification* Required

Responses

Status: 200 - network provider deleted

Status: 401 - Access token is missing or invalid

Status: 404 - network provider not found

deleteParser

Delete a parser

Delete a parser from the system

DELETE

/parser/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X DELETE "http://lmp.dunkel.world/api//parser/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *parser identification* Required

Responses

Status: 200 - parser deleted

Status: 401 - Access token is missing or invalid

Status: 404 - parser not found

getAlertConfiguration

Get alert configuration information

By passing in the appropriate options, you can search for available alert configurations in the system

GET

/alert-config/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//alert-config/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *alert configuration identification* Required

Responses

Status: 200 - alert configuration was found

Schema

```
▾ [ []
    undefined
]
```

Status: 401 - Access token is missing or invalid

Status: 404 - alert configuration not found

getAlertsByDeveui

Get all device alerts

Get all device alerts

GET

/alert/{deveui}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//alert/{deveui}"
```

Parameters

Path parameters

Name	Description
deveui*	String *device identification* Required

Responses

Status: 200 - List of alerts by device

Schema

▾ [[]
undefined
]

Status: 401 - Access token is missing or invalid

Status: 404 - device not found

getAllAlertConfigs

Get all alert configurations

Get all alert configurations

GET

/alert-config

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//alert-config?page=&size="
```

Parameters

Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - List of alerts configurations

Schema

▾ [[]
undefined
]

Status: 401 - Access token is missing or invalid

Status: 404 - alert configuration not found

getAllClientApplications

Get all available client applications

Get all client applications

GET

/clientapp

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//clientapp?page=&size="
```

Parameters

Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - list of client applications

Schema

▾ [[]
undefined
]

Status: 400 - invalid input

Status: 401 - Access token is missing or invalid

getAllDeviceDownlinks

Get all downlinks of a device

Get all device downlinks

GET

/device/{deveui}/downlink

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//device/{deveui}/downlink?page=&size="
```

Parameters

Path parameters

Name	Description
deveui*	String *device identification* Required

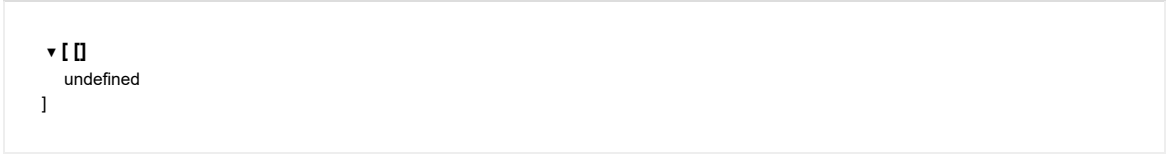
Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - list of downlink

Schema



Status: 400 - invalid input

getAllDeviceGroups

Get all available device groups

Get all device group

GET

/devicegroup

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//devicegroup?page=&size="
```

Parameters

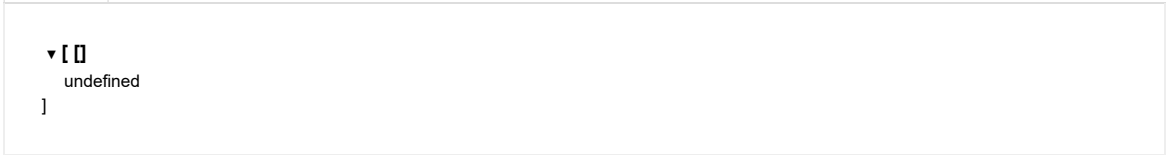
Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - list of devices groups

Schema



Status: 400 - invalid input

Status: 401 - Access token is missing or invalid

getAllDeviceTypes

Get all available device types

Get all device types

GET

/devicetype

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//devicetype?page=&size="
```

Parameters

Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - list of devices types

Schema

```
▾ [ []
    undefined
]
```

Status: 400 - invalid input

Status: 401 - Access token is missing or invalid

getAllDeviceUplinks

Get all uplinks of a device

Get all device uplinks

GET

/device/{deveui}/uplinks

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//device/{deveui}/uplinks?page=&size="
```

Parameters

Path parameters

Name	Description
deveui*	String *device identification* Required

Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - list of devices groups

Schema

```
▾ [ []
    undefined
]
```

Status: 400 - invalid input

Status: 401 - Access token is missing or invalid

getAllDevices

Get all personal devices

Get all personal devices

GET

```
/device
```

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//device?page=&size="
```

Parameters

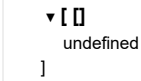
Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - list of devices

Schema



Status: 400 - invalid input

Status: 401 - Access token is missing or invalid

getAllFirmwares

Get all firmwares

Get all firmwares

GET

/firmware

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//firmware?page=&size="
```

Parameters

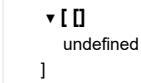
Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - List of firmwares

Schema



Status: 401 - Access token is missing or invalid

Status: 404 - firmware not found

getAllNetworkProviders

Get all available network providers

Get all network providers

GET

/networkprovider

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//networkprovider?page=&size="
```

Parameters

Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*
size	String *The numbers of items to return*

Responses

Status: 200 - list of network providers

Schema

```
▾ [ []
    undefined
]
```

Status: 400 - invalid input

Status: 401 - Access token is missing or invalid

getAllParsers

Get all parsers

Get all parsers

GET

/parser

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//parser?page=&size="
```

Parameters

Query parameters

Name	Description
page	String *The number of items to skip before starting to collect the result*

size	String *The numbers of items to return*

Responses

Status: 200 - List of parsers

Schema

▾ [[]
undefined
]

Status: 401 - Access token is missing or invalid

Status: 404 - parser not found

getClientApplication

Get client application information

By passing in the appropriate options, you can search for available client applications in the system

GET

/clientapp/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//clientapp/{id}"
```

Parameters

Path parameters

Name	Description
id*	Integer *client application identification* Required

Responses

Status: 200 - client application was found

Schema

▾ [[]
undefined
]

Status: 401 - Access token is missing or invalid

Status: 404 - client application not found

getDevice

Get device information

By passing in the appropriate options, you can search for available inventory in the system

GET

/device/{deveui}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//device/{deveui}"
```

Parameters

Path parameters

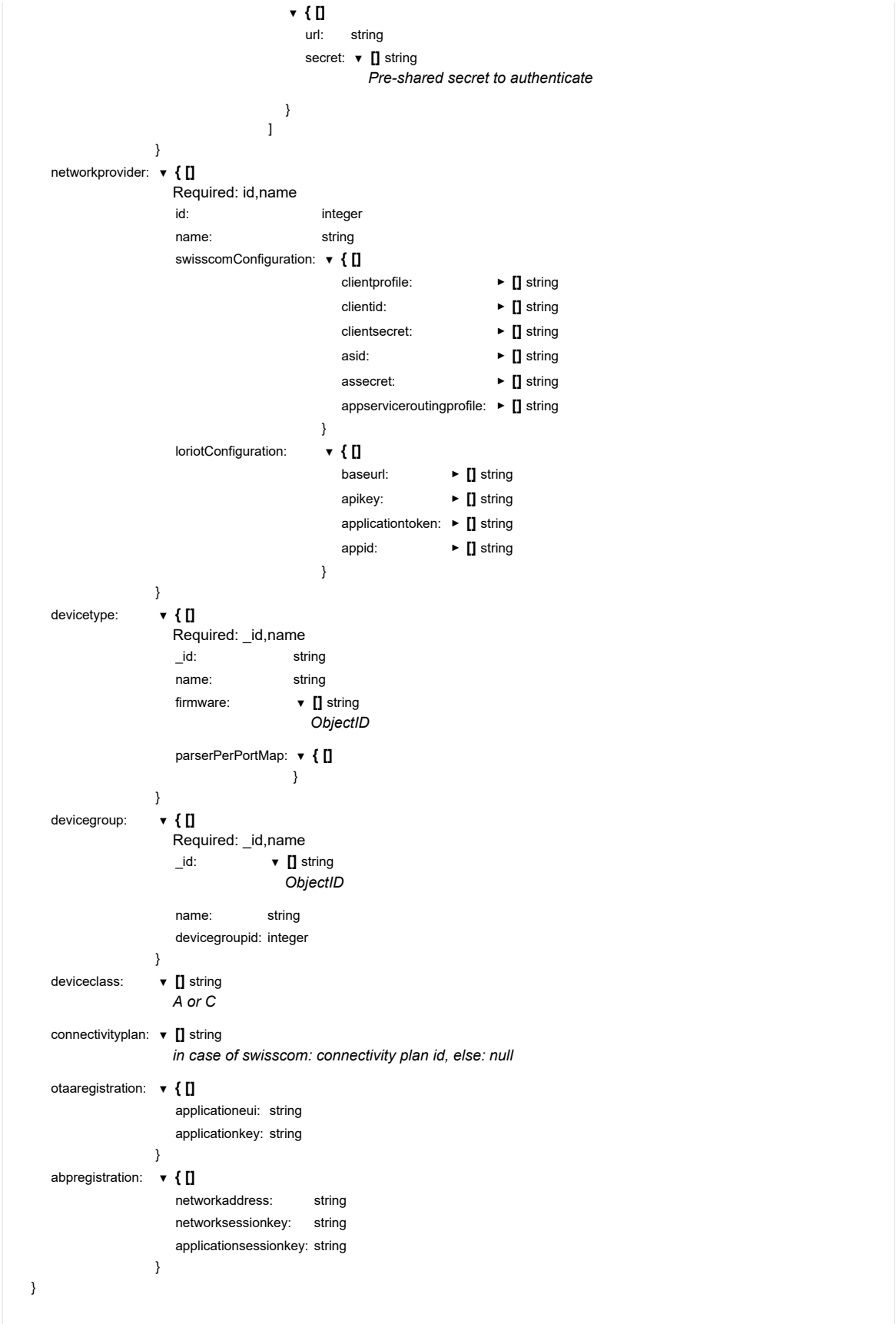
Name	Description
deveui*	String *device identification* Required

Responses

Status: 200 - device was found

Schema

▾ { []
Required: deveui,name
deveui: string
name: string
account: ▾ { []
id: integer
accountname: string
password: string
}
clientapplication: ▾ { []
Required: _id,name,url
_id: ▾ [] string
ObjectID
name: string
apitoken: string
influxoutput: ▾ [[]
▾ { []
url: string
dbname: ▾ [] string
Database name
measurement: ▾ [] string
Table name
username: ▾ [] string
database username
password: ▾ [] string
database password
}
]
postoutputuris: ▾ [[]


▾ { []
url: string
secret: ▾ [] string
Pre-shared secret to authenticate
}
]
}
networkprovider: ▾ { []
Required: id,name
id: integer
name: string
swisscomConfiguration: ▾ { []
clientprofile: ▸ [] string
clientid: ▸ [] string
clientsecret: ▸ [] string
asid: ▸ [] string
assecret: ▸ [] string
appserviceroutingprofile: ▸ [] string
}
loriotConfiguration: ▾ { []
baseurl: ▸ [] string
apikey: ▸ [] string
applicationtoken: ▸ [] string
appid: ▸ [] string
}
}
devicetype: ▾ { []
Required: _id,name
_id: string
name: string
firmware: ▾ [] string
ObjectID
parserPerPortMap: ▾ { []
}
}
devicegroup: ▾ { []
Required: _id,name
_id: ▾ [] string
ObjectID
name: string
devicegroupid: integer
}
deviceclass: ▾ [] string
A or C
connectivityplan: ▾ [] string
in case of swisscom: connectivity plan id, else: null
otaaregistration: ▾ { []
applicationeui: string
applicationkey: string
}
abpregistration: ▾ { []
networkaddress: string
networksessionkey: string
applicationsessionkey: string
}
}

Status: 401 - Access token is missing or invalid

Status: 404 - device not found

getDeviceGroup

Get device group information

By passing in the appropriate options, you can search for available device type in the system

GET

/devicegroup/{name}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//devicegroup/{name}"
```

Parameters

Path parameters

Name	Description
name*	String *device group name* Required

Responses

Status: 200 - device group was found

Schema

▾ [[]
undefined
]

Status: 401 - Access token is missing or invalid

Status: 404 - device group not found

getDeviceType

Get device type information

By passing in the appropriate options, you can search for available device type in the system

GET

/devicetype/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//devicetype/{id}"
```

Parameters

Path parameters

Name	Description
id*	Integer *device type identification* Required

Responses

Status: 200 - device type was found

Schema

```
▾ [ []
    undefined
]
```

Status: 401 - Access token is missing or invalid

Status: 404 - device type not found

getFirmware

Get firmware information

By passing in the appropriate options, you can search for available firmwares in the system

GET

```
/firmware/{id}
```

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//firmware/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *firmware identification* Required

Responses

Status: 200 - firmware was found

Schema

```
▾ [ []
    undefined
]
```

Status: 401 - Access token is missing or invalid

Status: 404 - firmware not found

getNetworkProvider

Get network provider information

By passing in the appropriate options, you can search for available network providers in the system

GET

/networkprovider/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//networkprovider/{id}"
```

Parameters

Path parameters

Name	Description
id*	Integer *network provider identification* Required

Responses

Status: 200 - network provider was found

Schema

```
▾ [ []
    undefined
]
```

Status: 401 - Access token is missing or invalid

Status: 404 - network provider not found

getParser

Get parser information

By passing in the appropriate options, you can search for available parsers in the system

GET

/parser/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X GET "http://lmp.dunkel.world/api//parser/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *parser identification* Required

Responses

Status: 200 - parser was found

Schema

```
▾ [ []
    undefined
]
```

Status: 401 - Access token is missing or invalid

Status: 404 - parser not found

registerAllDevicesInDeviceGroup

Register every device in device group

Register every device in device group

POST

```
/devicegroup/{id}/register
```

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//devicegroup/{id}/register"
```

Parameters

Path parameters

Name	Description
id*	String *device group name* Required

Responses

Status: 200 - All devices registered

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

sendDownlinkToDevice

Send a downlink package to device

Send a downlink package to the device

POST

/device/{deveui}/downlink

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//device/{deveui}/downlink"
```

Parameters

Path parameters

Name	Description
deveui*	String *device identification* Required

Body parameters

Name	Description
body *	▾ { [] Required: confirmed,deveui,payload,ports deveui: string ports: string payload: string confirmed: boolean }

Responses

Status: 200 - downlink sent

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

sendDownlinkToDevicesInDeviceGroup

Send a downlink package to every device in device group

Send a downlink package to every device in device group

POST

/devicegroup/{id}/downlink

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X POST "http://lmp.dunkel.world/api//devicegroup/{id}/downlink"
```

Parameters

Path parameters

Name	Description
id*	String *device group name* Required

Body parameters

Name	Description
body *	▾ { [] Required: confirmed,deveui,payload,ports deveui: string ports: string payload: string confirmed: boolean }

Responses

Status: 200 - downlink sent

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

updateAlertConfiguration

Updates an alert configuration

Updates an alert configuration in the system

PUT

```
/alert-config/{id}
```

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X PUT "http://lmp.dunkel.world/api//alert-config/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *alert configuration identification* Required

Body parameters

Name	Description

body	▾ { [] _id: ▾ [] string *ObjectID* name: string code: ▾ [] string *Python code to execute on every uplink from device* }

Responses

Status: 204 - alert configuration updated

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 404 - alert configuration not found

updateClientApplication

Updates a client application

Updates a client application in the system

PUT

/clientapp/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X PUT "http://lmp.dunkel.world/api//clientapp/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *client application identification* Required

Body parameters

Name	Description

body	▾ { [] Required: _id,name,url _id: ▾ [] string *ObjectID* name: string apitoken: string influxoutput: ▾ [[] ▾ { [] url: string dbname: ▾ [] string *Database name* measurement: ▾ [] string *Table name* username: ▾ [] string *database username* password: ▾ [] string *database password* }] postoutputuris: ▾ [[] ▾ { [] url: string secret: ▾ [] string *Pre-shared secret to authenticate* }] }

Responses

Status: 204 - client application updated

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 404 - client application not found

updateDevice

Updates a device

Updates a device in the system

PUT

/device/{deveui}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X PUT "http://lmp.dunkel.world/api//device/{deveui}"
```

Parameters

Path parameters

Name	Description
deveui*	String *device identification* Required

Body parameters

Name	Description
body	▾ { [] Required: deveui,name deveui: string name: string account: ▾ { [] id: integer accountname: string password: string } clientapplication: ▾ { [] Required: _id,name,url _id: ▸ [] string name: string apitoken: string influxoutput: ▸[] [] postoutputuris: ▸[] [] } networkprovider: ▾ { [] Required: id,name id: integer name: string swisscomConfiguration: ▸ { } [] loriotConfiguration: ▸ { } [] } devicetype: ▾ { [] Required: _id,name _id: string name: string firmware: ▸ [] string parserPerPortMap: ▸ { } [] } devicegroup: ▾ { [] Required: _id,name _id: ▸ [] string name: string devicegroupid: integer } deviceclass: ▾ [] string *A or C* connectivityplan: ▾ [] string *in case of swisscom: connectivity plan id, else: null* otaaregistration: ▾ { [] applicationeui: string applicationkey: string } abpregistration: ▾ { [] networkaddress: string networksessionkey: string applicationsessionkey: string } }

Responses

Status: 204 - device updated

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 404 - device not found

updateDeviceGroup

Updates a device group

Updates a device group in the system

PUT

/devicegroup/{name}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X PUT "http://lmp.dunkel.world/api//devicegroup/{name}"
```

Parameters

Path parameters

Name	Description
name*	String *device group name* Required

Body parameters

Name	Description
body	▾ { [] Required: name name: string devicegroup: string alerts: ▾ [[] ▾ { [] _id: ▾ [] string *ObjectID* text: string date: string }] }

Responses

Status: 204 - Device group updated

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 404 - Device group not found

updateDeviceType

Updates a device type

Updates a device type in the system

PUT

/devicetype/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X PUT "http://lmp.dunkel.world/api//devicetype/{id}"
```

Parameters

Path parameters

Name	Description
id*	Integer *device type identification* Required

Body parameters

Name	Description
body	▾ { [] Required: _id,name _id: string name: string firmware: ▾ [] string *ObjectID* parserPerPortMap: ▾ { [] } }

Responses

Status: 204 - Device type updated

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 404 - Device type not found

updateFirmware

Updates a firmware

Updates a firmware in the system

PUT

/firmware/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X PUT "http://lmp.dunkel.world/api//firmware/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *firmware identification* Required

Body parameters

Name	Description
body	▾ { [] _id: ▾ [] string *ObjectID* versionname: ▾ [] string *name of the firmware version* }

Responses

Status: 204 - firmware updated

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 404 - firmware not found

updateNetworkProvider

Updates a network provider

Updates a network provider in the system

PUT

```
/networkprovider/{id}
```

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X PUT "http://lmp.dunkel.world/api//networkprovider/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *network provider identification* Required

Body parameters

Name	Description
body	▾ { [] Required: id,name id: integer name: string swisscomConfiguration: ▾ { [] clientprofile: ▸ [] string clientid: ▸ [] string clientsecret: ▸ [] string asid: ▸ [] string assecret: ▸ [] string appserviceroutingprofile: ▸ [] string } loriotConfiguration: ▾ { [] baseurl: ▸ [] string apikey: ▸ [] string applicationtoken: ▸ [] string appid: ▸ [] string } }

Responses

Status: 204 - network provider updated

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 404 - network provider not found

updateParser

Updates a parser

Updates a parser in the system

PUT

/parser/{id}

Usage and SDK Samples

Curl | Java | Android | Obj-C | JavaScript | C# | PHP | Perl | Python

```
curl -X PUT "http://lmp.dunkel.world/api//parser/{id}"
```

Parameters

Path parameters

Name	Description
id*	String *parser identification* Required

Body parameters

Name	Description

body	▾ { [] _id: ▾ [] string *ObjectID* name: ▾ [] string *name of the parser* code: ▾ [] string *Parsercode written in Python* testpayload: ▾ [] string *Payload to test against the parser code* }

Responses

Status: 204 - parser updated

Status: 400 - invalid input, object invalid

Status: 401 - Access token is missing or invalid

Status: 404 - parser not found

Suggestions, contact, support and error reporting;

Information URL: https://helloreverb.com (https://helloreverb.com)
Contact Info: mdunkel@hsr.ch (mdunkel@hsr.ch)