OST
Ostschweizer
Fachhochschule

IoT Coffee Roaster

Studienarbeit

Studiengang Informatik

OST – Ostschweizer Fachhochschule

Campus Rapperswil-Jona

Herbstsemester 2020

Abstract

Ein Start-Up stellt Kaffeeröster her und möchte diese in eine IoT-Umgebung einbinden. Der Röster soll über eine App gesteuert werden und ein Admin Interface als Web Applikation soll einen Überblick über alle Röster geben. Die App ist bereits vorhanden, arbeitet derzeit aber mit einer USB-Verbindung. Über diese kann eine Röstung gestartet und gestoppt werden. Während der Röstung können die verschiedenen Sensoren im Röster überwacht werden. Das Admin Interface gibt eine Übersicht über alle registrierten Röster. In einer Detail Ansicht sollen die Sensordaten der einzelnen Röster angezeigt werden. Der Röster empfängt Kommandos, welche er bearbeiten wird und sendet Sensordaten an die App bzw. dem Interface zurück. Diese Kommunikation soll mit einem Kommunikationsprotokoll definiert werden, welches leicht erweiterbar sein soll, falls in Zukunft neue Befehle oder Sensoren hinzustossen.

Das Start-Up äusserte seine Wünsche während des Kickoffs des Projektes. Diese mussten eingegrenzt und in Ziele ausformuliert werden. Da sich das Team nicht mit der Materie auskannte, musste sich jedes Teammitglied ins Thema einlesen. Nach dem Verfassen des Projektplans und der Anforderungen, wurde mit der Umsetzung begonnen. Als erstes wurde das Kommunikationsprotokoll erstellt. Es gab Probleme mit dem Eval Board, welches in den Röster installiert werden sollte, das für das Senden von Sensor Daten und Empfangen von Kommandos zuständig ist. Die vom Hersteller bereitgestellten Libraries waren teilweise inkompatibel, spärlich dokumentiert und das Team verfügte über zu wenige Kentnisse in der Elektrotechnik. Deswegen hat das Team eine Alternativlösung vorgeschlagen, damit die Arbeit fortgesetzt werden konnte. Diese Alternative bestand darin, einen Röster auf einem Raspberry Pi 4 als State Machine zu simulieren, sodass die Kommunikation dennoch erstellt werden konnte. Zum Schluss wurde noch das Admin Interface und eine Library für die IoT Funktionen implementiert. Das Kommunikationsprotokoll basiert auf MQTT.Dabei handelt es sich um den de-facto Standard in der IoT Entwicklung. Weil mit dem AWS IoT Core von Amazon gearbeitet wurde, konnten nicht alle MQTT Features verwendet werden. Die App wurde mit React Native entwickelt, entsprechend wurden das Admin Interface und die Library mit React und Typescript programmiert.

Das Ergebnis dieser Arbeit ist eine funktionierende Kommunikation zwischen App, Admin Interface und Röster. Es ist möglich, einen Befehl über die implementierte Library an den Raspberry Pi (simulierter Röster) zu senden. Erhält dieser einen Befehl zum Rösten, wechselt er in den "Roasting" State. In diesem simuliert er das Erhalten von verschiedenen Sensordaten. Diese Werte werden veröffentlicht und können auf dem Admin Interface betrachtet werden. Soll eine Röstung abgebrochen werden, kann dies mit einem "Abbruch" Befehl getan werden. Das Kommunikationsprotokoll kann einfach erweitert werden, falls neue Befehle oder Sensoren hinzugefügt werden. Auch das Admin Interface und die Library können mit nur wenig Aufwand ergänzt werden. Als nächster Schritt sollte das Eval Board konfiguriert werden, damit die Sensordaten nicht mehr simuliert werden müssen, sondern direkt echte Daten verwendet werden können.

Management Summary

Ausgangslage

Ein Start-Up stellt Kaffeeröster her, die in der Gastronomie eingesetzt werden sollen. Es besteht bereits eine App, womit die Röster gesteuert werden können. Diese App muss aber zur Zeit über ein USB-Kabel mit einem Röster verbunden werden. Dies ist nicht mehr zeitgemäss und deswegen soll diese Kommunikation neu über das Internet laufen. Des Weiteren soll eine Übersicht über alle Röster erstellt werden, das sogenannte Admin Interface. Während einer laufenden Röstung sollen die Sensordaten der röstenden Maschine über das Admin Interface ersichtlich sein.

Vorgehen

Da für die Umsetzung der Wünsche des Start-Up die aktuellen Fachkenntnisse des Teams nicht ausreichten, verbrachte jedes Mitglied zunächst Zeit, diese Wissenslücken zu füllen. Danach wurden in mehreren internen Besprechungen die Wünsche eingegrenzt und in konkrete Anforderungen umformuliert. Dies beinhaltete ein grobes Konzept, wie jede dieser Anforderungen erreicht werden sollte. Nachdem die grobe Planung des Projektes vollzogen war und die Umsetzung begann, wurden die Teammeetings auf eine wöchentliche Lagebesprechung reduziert. Jede zweite Woche gab es zusätzlich zu den internen Meetings auch Besprechungen mit dem Start-Up, damit stets sichergestellt werden konnte, dass das Projekt in die gewünschte Richtung ging.

Ergebnisse

Das Ergebnis ist die erfolgreiche Kommunikation zwischen App, Admin Interface und Röster. Da es ein Problem mit der Hardware des Rösters gab, war es dem Team nicht möglich Sensordaten vom Röster selbst zu erhalten. Um trotzdem einen Kommunikationspartner für App und Admin Interface zu haben, wurde mit Einverständnis des Start-Up ein alternativer Weg eingeschlagen. Statt des vom Start-Up vorgeschlagenen Eval Boards wurde als Hardware ein Raspberry Pi 4 verwendet. Darauf wurde ein Röster simuliert, der Kommandos von der App empfangen und laufend Sensordaten an das Admin Interface senden kann. Das Admin Interface wurde erfolgreich erstellt und besteht aus zwei Teilen. Erstens dient es als eine Übersicht über alle verbundenen Röster und zweitens können die Sensordaten einzelner Röster überwacht werden. Die für die Kommunikation benötigte Funktionalität wurde in eine Programmbibliothek ausgelagert, die sowohl in der App als auch für das Admin Interface

verwendet werden kann. Dabei wurde darauf geachtet, dass später mit wenig Mehraufwand weitere Features hinzugefügt werden können.

Ausblick

Obwohl mit dem Projekt die Kommunikation über das Internet realisiert wurde, ist der Röster selbst noch nicht Teil dieser Lösung. Der nächste Schritt besteht darin das Eval Board zu konfigurieren. Somit wird der Röster kommunikationsfähig und die Röster-Simulation kann damit abgelöst werden.

Inhaltsverzeichnis

1. Einleitung

Ein Start-Up hat eine Vision im Bereich der Bohnenröstung. Die Rösterindustrie kauft Kaffeebohnen ein, röstet diese und verkauft sie an Kaffeeverkäufer. Dabei nehmen sie 50% des Verkaufspreis des Kaffees ein. Das Start-Up will nun Roasting as a Service anbieten, so dass die Kaffeeverkäufer Bohnen selber rösten können. Dies bringt den Kaffeebauern mehr Geld, die Röstung ist frischer und kann personalisiert werden.

Das Start-Up ist auf gutem Weg, diese Vision zur Realität zu machen. Diese Studienarbeit ist ein kleiner Teil dieses Vorhabens. Aktuell ist der Röster über eine App via USB steuerbar. Da dies nicht mehr zeitgemäss ist, ist das Ziel dieser Studienarbeit, dass die ganze Kommunikation über eine Cloud verläuft.

Zuerst wurde ein Projektplan für die Studienarbeit erstellt und der Wunsch vom Start-Up analysiert und in Use Cases aufgeteilt (Analyse). Es folgen weitere festgelegte Richtlinien unter Vorgehensweise. In den restlichen Kapitel werden die erreichten Ergebnisse dokumentiert.

2. Analyse

Im Kapitel Änalyse" ist der Projektplan und die Anforderungsspezifikationen zu finden. Die Analyse wurde zu Beginn der Arbeit erstellt und daraufhin bewusst nicht mehr verändert, damit ein guter Vergleich zwischen Planung und Resultat gemacht werden kann. Als Grundlage für die Anforderungsanalyse dient das FURPS+[7] Modell.

2.1. Projektplan

2.1.1. Übersicht

Die Studienarbeit wurde am Montag, 14.09.2020, gestartet und die Deadline ist am 18.12.2020. Geplant sind 240 Stunden pro Teammitglied. Verteilt auf die einzelnen Wochen entspricht dies ungefähr einem Aufwand von 17 Stunden pro Teammitglied.

Jedes Teammitglied erfasst die geleistete Zeit mit einer Genauigkeit von 30 Minuten, damit die Zeitvorgaben möglichst genau kontrolliert werden können. Die Zeit wird über Google Tabellen erfasst.

Projektdauer	14 Wochen
Teamgrösse	3 Personen
Totale Arbeitsstunden	720 Stunden
Wöchentliche Arbeitsstunden pro Person	17 Stunden
Projektstart	14.09.2020
Projektende	18.12.2020

2.1.2. Zeitliche Planung

Im Folgenden sind die grössten Arbeitsprodukte aufgelistet und die 720 Stunden darauf verteilt. Der Projektplan wird nach der Elaboration Phase nicht mehr verändert, damit am Ende die effektiv gebrauchte Zeit mit der geplanten Zeit verglichen werden kann.

Die Zeit der Arbeitsprodukte wird auf 10 Stunden gerundet.

Externe Meetings	8x Vorbereitung (2h) und Meeting (3x1h)	40h
Interne Meetings	14x Wöchentliche Meetings (3x1h)	40h
Projektadministration	GitLab, Sprintvorbereitung, Leitung	20h
Einlesen	In die Thematik einlesen, AWS IoT kennenlernen	20h
Projektplan	Projektplan erstellen	20h
Requirements	Requirements Dokument erstellen	20h
Architektur	Architektur Dokument erstellen	20h
Methodik	Projektmanagement, Entwicklungsflow usw. beschreiben	20h
Restliche Dokumente	Time Report, Design, Testing, usw.	40h
Abschlussbericht	Abstract, Management Summary, Persönlicher Bericht, usw.	40h
Dokumentation finalisieren	Vollständigkeit, Gegenlesen	20h
Umgebung aufsetzen	Umgebung aufsetzen	20h
Prototyp erstellen	Erste Kommunikation läuft	20h
Protokoll definieren	Protokoll entwerfen und dokumentieren	40h
User Interface	User Interface erstellen	10h
Use Cases	Bearbeitung der Use Cases (6x30h)	180h
Performance Tests	Performance Tests erstellen	10h
Simulation	Inklusive Unit Tests	50h
Implementation beenden	Bugfixes, Code Cleanup, Unfertiges Beenden	20h
Total Organisatorisches	Inklusive Meetings	120h
Total Dokumentation	Erstellen verschiedenster Dokumente	180h
Total Implementation	Infrastruktur, Coding, Testing	350h
Reserve	10% geplante zeitliche Reserve	70h
Total	Alle Arbeitsprodukte zusammengerechnet	720h

2.1.3. Sprints

Sprint 1	(Woche 1)
Start	14.09.2020
Ende	20.09.2020
Beschreibung	Kickoff, Einlesen
Vorgehen	Treffen in Luzern

Sprint 2	(Woche 2-3)
Start	21.09.2020
Ende	04.10.2020
Beschreibung	Projektplan, Requirements
Vorgehen	Projektplan, Requirements und Use Case Diagramm erstellen

Sprint 3	(Woche 4-5)
Start	05.10.2020
Ende	18.10.2020
Beschreibung	Requirements, Architektur, Protokoll definieren
Vorgehen	Requirements abschliessen, Architektur erstellen, Protokoll definieren

Sprint 4	(Woche 6-7)
Start	19.10.2020
Ende	01.11.2020
Beschreibung	Bearbeitung Use Cases Teil 1
Vorgehen	Use Cases umsetzen

Sprint 5	(Woche 8-9)
Start	02.11.2020
Ende	15.11.2020
Beschreibung	Bearbeitung Use Cases Teil 2
Vorgehen	Use Cases umsetzen

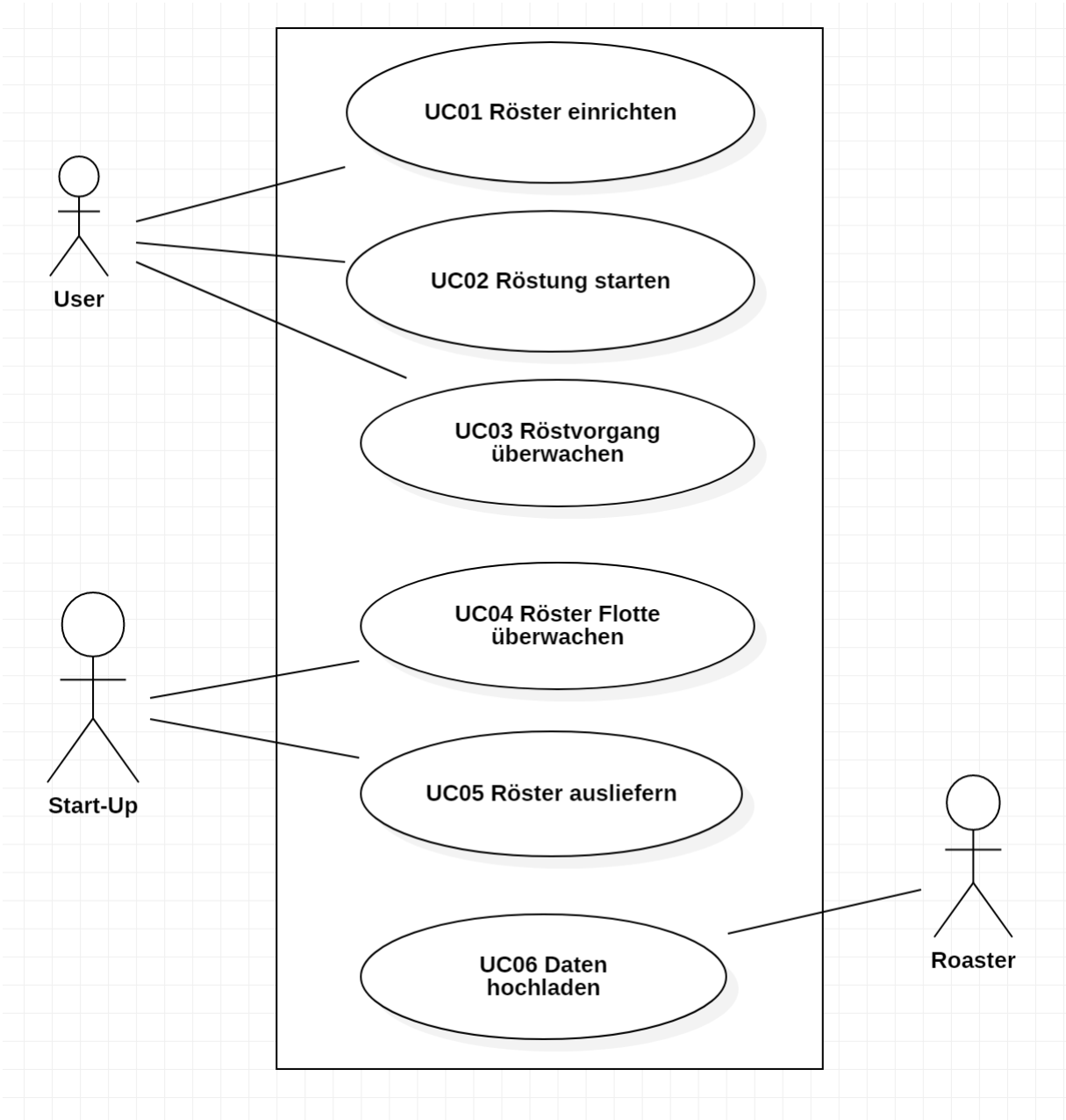
Sprint 6	(Woche 10-11)
Start	16.11.2020
Ende	29.11.2020
Beschreibung	Testing, Qualitätskontrolle, Beenden Use Cases
Vorgehen	Use Cases bearbeiten, Tests durchführen

Sprint 7	(Woche 12-13)
Start	30.11.2020
Ende	13.12.2020
Beschreibung	Reservezeit, Projekt abschliessen, Finale Kontrolle, Dokumentation vervollständigen
Vorgehen	Produkt bereit zur Ablieferung, alle benötigten Dokumente zusammenführen

Sprint 8	(Woche 14)
Start	14.12.2020
Ende	18.12.2020
Beschreibung	Projekt abgeben, Produkt abliefern
Vorgehen	Abstract hochladen, alle Dokumente hochladen, Produkt abliefern

2.2. Funktionale Anforderungen

2.2.1. Use Case Diagramm



2.2.2. Beschreibungen (Brief)

UC01 Röster einrichten

Der Benutzer verbindet den Röster mit dem Internet, damit die App die Zustandsdaten des Rösters anzeigen kann.

UC02 Röstung starten

Will ein Benutzer eine Röstung starten, muss der Röster über eine Internet Verbindung verfügen. Der Benutzer selektiert eine Röstung über die App und startet diese mit einem Knopfdruck.

UC03 Röstvorgang überwachen

Ist ein Röster mit dem Internet verbunden, kann der Benutzer Daten über den Röster abfragen.

UC04 Röster Flotte überwachen

Ein Superuser hat die Möglichkeit, über ein Admin Interface alle mit dem Internet verbundenen Röster zu überprüfen.

UC05 Röster ausliefern

Das Start-Up bereitet den Röster samt Zubehör vor, damit er bereit für die Auslieferung an den Kunden ist. Dazu werden Zertifikate im AWS hinterlegt und überprüft, ob die App die Zustandsdaten des Rösters erkennt.

UC06 Daten hochladen

Ein Röster, der mit dem Internet verbunden ist, soll seine gemessenen Daten teilen. Erhält das Eval Board neue Daten von den Sensoren, werden diese an den Server übermittelt.

2.2.3. Beschreibungen (Fully Dressed)

UC01 Röster einrichten

- Primary Actor: Benutzer
- Preconditions: Der Benutzer hat die App korrekt installiert. Der Röster wurde korrekt ausgeliefert.
- Success Guarantee: Der Benutzer steuert den Röster über die App.
- Main Success Scenario:
  1. Der Röster stellt ein Wifi Access Point (WAP) bereit.
  2. Der Benutzer gibt die Zugangsdaten ein vom WAP des Rösters.
  3. Der Benutzer wird verbunden mit dem WAP.
  4. Der Benutzer wählt ein WLAN aus und loggt sich ein.
  5. Der Röster verbindet sich mit dem Internet.
  6. Die App benachrichtigt den Benutzer, dass der Röster verbunden ist.
- Extensions (or Alternative Flows):

  1a. Der Röster verbindet sich nicht mit dem WLAN:
  Die App benachrichtigt den Benutzer. Dieser muss bei 2. weiterfahren.

UC02 Röstung starten

- Primary Actor: Benutzer
- Preconditions: Der Röster ist mit AWS verbunden. Die App verfügt über eine Internetverbindung.
- Success Guarantee: Der Röster startet eine Röstung. Die App zeigt den Zustand der aktuellen Röstung an.
- Main Success Scenario:
  1. Der Benutzer wählt das gewünschte Röstprofil über die App aus.
  2. Der Benutzer bestätigt die Auswahl und startet damit die Röstung mit einem Knopfdruck.
  3. Der Benutzer füllt den Röster mit Kaffeebohnen.
  4. Der Röster beginnt die Röstung nach dem ausgewählten Profil.
- Extensions (or Alternative Flows):

1a. Der Röster ist nicht mit dem Internet verbunden:
Die App liefert keine aktuellen Daten. Die App informiert den Benutzer, dass der Röster nicht erreichbar ist. UC01 muss zunächst ausgeführt werden.

3a. Der Benutzer füllt den Röster nicht mit Bohnen:
Das App benachrichtigt den Benutzer und wartet bis 3. ausgeführt wird.

UC03 Röstvorgang überwachen

- Primary Actor: Benutzer
- Preconditions: Eine Röstung ist im Gange. Der Benutzer verfügt über die nötigen Berechtigungen. Der Röster verfügt über eine Verbindung mit dem Internet.
- Success Guarantee: Der Benutzer sieht den Zustand der aktuellen Röstung.
- Main Success Scenario:
  1. Der Röster verarbeitet Kaffeebohnen.
  2. Der Benutzer wählt die aktuelle Röstung auf der App aus.
  3. Die App zeigt die aktuellen Daten über die laufende Röstung.
- Extensions (or Alternative Flows):

1-3a. Der Röster bricht die Röstung ab:
Die App informiert den Benutzer über den Abbruch.

1-3b. Der Röster verliert die Verbindung zum Internet:
Der Röster beendet die aktuelle Röstung und versucht die Verbindung wiederherzustellen. Der Benutzer erhält in der Zwischenzeit keine Daten.

3a. Der Benutzer bemerkt, dass der Röstvorgang nicht wie erwartet verläuft:
Der Benutzer bricht die Röstung ab.

UC04 Röster Flotte überwachen

- Primary Actor: Superuser
- Preconditions: Der Röster wurde korrekt ausgeliefert.
- Success Guarantee: Der Superuser sieht im Admin Interface alle Röster, die im Umlauf sind.
- Main Success Scenario:
  1. Das System erlaubt dem Superuser den Zugriff auf das Admin Interface.
  2. Der Superuser sieht eine Übersicht aller Röster im Umlauf.

3. Der Superuser klickt auf einzelne Röster, um deren Zustand zu überprüfen.

- Extensions (or Alternative Flows):

1a. Ein Benutzer hat die Superuser Berechtigung nicht:
Das Admin Interface steht dem Benutzer nicht zur Verfügung.

1b. Ein Röster ist nicht mit dem Internet verbunden:
Der Röster wird im Admin Interface als offline angezeigt. Der Superuser kann den letzten Zustand dieses Rösters überprüfen. Erst bei erneuter Internetverbindung wird der aktuelle Zustand des Rösters dargestellt.

UC05 Röster ausliefern

- Primary Actor: Das Start-Up
- Preconditions: Die Hardware steht zur Verfügung.
- Success Guarantee: Der Röster ist im AWS registriert und besitzt die zugehörigen Zertifikate.
- Main Success Scenario:
  1. Das Start-Up stellt die Zertifikate im AWS aus.
  2. Diese werden auf dem Röster hinterlegt.
  3. Das Eval-Board wird getestet und die App erkennt die Zustandsdaten des Rösters.
  4. Der Superuser sieht den Röster im Admin Interface.

UC06 Daten hochladen

- Primary Actor: Röster
- Trigger: Der Röster aktualisiert seine Zustandsdaten.
- Preconditions: Der Röster ist mit dem Internet verbunden.
- Success Guarantee: Die App erhält laufend Daten über den Röster.
- Main Success Scenario:
  1. Das Eval Board erkennt eine Veränderung der Daten.
  2. Der Röster sendet die Daten an AWS.
- Extensions (or Alternative Flows):

1a. Der Röster kann keine Verbindung zu AWS herstellen:
Der Röster versucht die Verbindung mit AWS herzustellen. In der Zwischenzeit werden keine Daten zwischengespeichert.

2.3. Nicht-Funktionale Anforderungen

2.3.1. Usability

- Momentan gibt es noch keine Röster bei Kunden. Die App ist jedoch bereits implementiert. Somit wird auf die Festlegung von Usability Anforderungen verzichtet.
- Das Admin Interface soll leicht zu verstehen sein und innerhalb von zehn Minuten sollen alle Funktionen erklärt worden sein.

2.3.2. Reliability

- Bei Verlust der Netzwerkverbindung soll der Röster den aktuellen Task zu Ende führen und versuchen die Verbindung wieder aufzubauen.

2.3.3. Performance

Als Normalbedingungen wird angenommen, dass:

- Röster und App über eine stabile Internetverbindung verfügen
- Röster und App sich in keinem Fehlerzustand befinden

Anforderung	**Minimal**	**Ziel**	**Optimal**
Verstrichene Zeit bis vom Röster gesendete Daten in der App ersichtlich sind (unter Normalbedingungen)	10s	2s	100ms
Verstrichene Zeit bis Befehle von der App vom Röster ausgeführt werden (unter Normalbedingungen)	10s	2s	100ms

2.3.4. Supportability

- Das Kommunikationsprotokoll zwischen Röster und App muss um zusätzliche Datenwerte für neue Sensoren oder Befehle erweiterbar sein.

2.3.5. Design Constraints

- Die Lösung wird in die bestehende Umgebung integriert.

2.3.6. Implementation Constraints

- Die Library wird in TypeScript entwickelt, damit diese in der App und im Admin Interface verwendet werden kann.
- Das Admin Interface wird in TypeScript und mit React geschrieben.
- Das Eval Board wird mit C/C++ programmiert.
- Die Röster Simulation wird mit Python entwickelt.
- Die Kommunikation zwischen Röster und App wird über AWS IoT Core abgewickelt.

2.3.7. Interface Constraints

- Die Kommunikation zwischen App und Röster erfolgt über den, von Amazon zur Verfügung gestellten, Service AWS IoT Core.

2.3.8. Physical Constraints

- Als Eval Board wird das B-L475E-IOT01A Discovery Kit for IoT[4] verwendet
- Die App soll auf einem Gerät mit Android 9 oder höher und mindestens 2GB RAM laufen

3. Vorgehensweise

In diesem Kapitel wird beschrieben, wie das Team gearbeitet hat und welche Richtlinien für das Projekt festgelegt wurden.

3.1. Projektmanagement

Es wird eine agile Projektmethode mit Sprint Längen von zwei Wochen verwendet. Dabei sind der erste und letzte Sprint nur eine Woche lang.

Als Projekt Managemet Tool wird ClickUp [5] verwendet. Über ClickUp können Sprints geplant und Arbeitspakete erstellt werden. Auch das Start-Up hat auf ClickUp Zugriff und kann dadurch den Projekt Status jederzeit selbst einsehen.

Durch die Covid-19 Situation ist das Studium grössten Teils noch in Form des Fernunterrichts. Deswegen werden Meetings über Microsoft Teams [13] gehalten. Als weiterer Kommunikationsweg gilt Slack. Über Slack [17] kann mit dem Start-Up, sowie dem Betreuer jederzeit kommuniziert werden.

Die Arbeitszeit wird mit der Hilfe eines Google Tabellen [10] Spreadsheet notiert. Dadurch wurden verschiedene Auswertungen getätigt, siehe Reflexion. Unsere rohen Stundenerfassung ist im Anhang ersichtlich.

Die Dokumentation wird mit LaTeX[12] geschrieben und über GitLab [9] verwaltet.

3.2. Arbeitspakete

Die Arbeitpakete werden im ClickUp als Tasks verwaltet. Dort gibt es für jeden Sprint ein Board. Die Tasks können zwischen diesen Boards hin und her geschoben werden. Ebenfalls besteht jedes Board aus fünf Spalten, in welche Tasks verschoben werden können.

1. Backlog
    a) Tasks, welche noch nicht für diesen Sprint geplant wurden.
    b) Tasks, welche noch nicht konkret definiert wurden.
2. Planned
    a) Tasks, welche konkret definiert wurden und für den Sprint geplant sind.
3. In Progress
    a) Tasks, welche in der ersten Umsetzungs Phase sind.

b) Tasks, welche durch ein Review gefallen sind und überarbeitet werden müssen.

4. Review
   a) Tasks, welche fertiggestellt wurden, aber noch überprüft werden müssen.
   b) Tasks, welche korrigiert wurden und erneut überprüft werden müssen.
5. Complete
   a) Tasks, welche fertiggestellt und erfolgreich geprüft wurden.

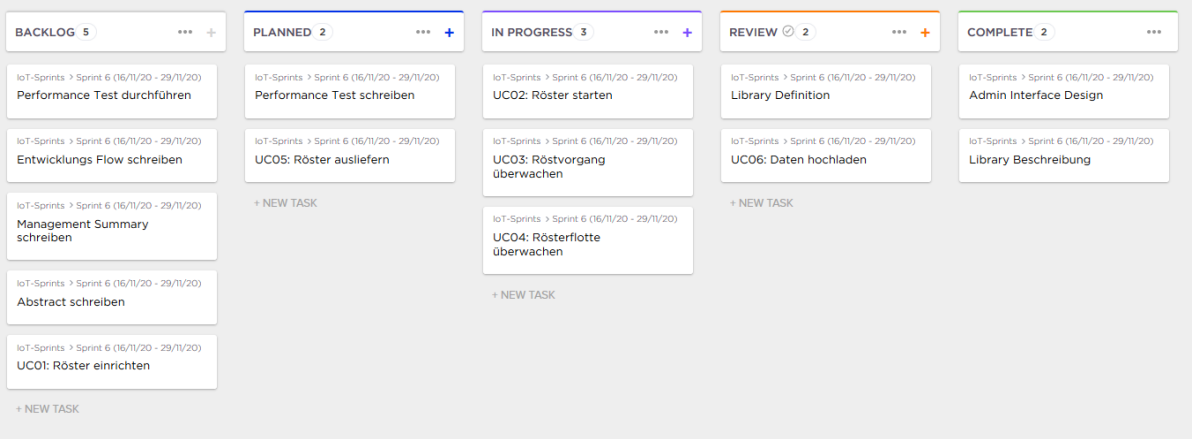


Abbildung 3.1.: ClickUp: Beispiel Projekt Administration

3.3. Workflows

Die Dokumentation wird über GitLab verwaltet während der Code über GitHub[8] gehostet wird, da dort eine Organisation hinterlegt wurde, in welcher alle Projekte verwaltet werden.

3.3.1. Dokumentation

Die Dokumentation wird mit LaTeX geschrieben. Versionen werden mit Hilfe von GitLab verwaltet. Im GitLab Repository wird auch direkt eine Pipeline gebaut, welche uns ein PDF der Dokumentation erstellt. Das Geschriebene wird jeweils von einer zweiten Person durchgelesen und bewertet.

3.3.2. Code Reviews

Für jeden Task wird ein eigener Branch mit Merge Request erstellt. Auf diesem Branch werden alle Änderungen entwickelt. Vor dem Merge muss ein Merge Request durch ein Code Review gehen. Bei Fehlern müssen diese korrigiert werden und dann erneut durch ein Review.

3.4. Coding Guidelines

3.4.1. Admin Interface

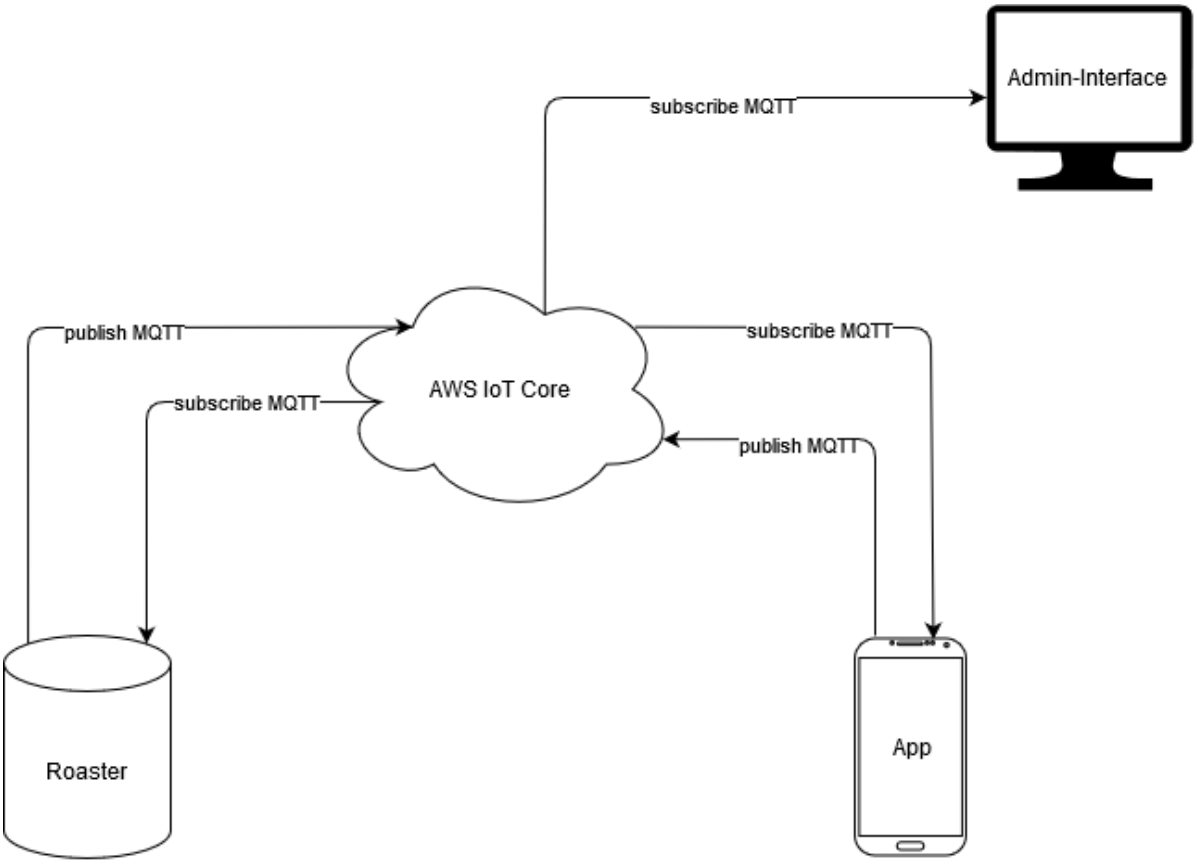
Für die Admin Interface React App wird TypeScript bzw. TSX verwendet. Zum Einhalten der Guidelines wurden ESLint und Prettier verwendet. Es wurde eine Liste an verschiedenen Plugins[11] verwendet, die ESLint und Prettier mit React, TypeScript und TSX kompatibel machen.

3.4.2. Röster

Für den Röster-Simulation Code wird der PEP8[15] Standard verwendet.

4. Systemübersicht

Das System wird in vier Teile unterteilt. Das Kommunikationsprotokoll, welches verwendet wird basiert auf MQTT[14]. Es gibt drei MQTT Clients und einen MQTT Broker. AWS übernimmt die Rolle des MQTT Broker. Der Röster, die App sowie das Web Interface der Admin Applikation sind die MQTT Clients. Clients kommunizieren nicht miteinander, die ganze Kommunikation findet über den Broker statt.



Um zu überprüfen wie schnell eine Nachricht via AWS IoT Core beim Empfänger ankommt, wurde ein kleiner Performancetest durchgeführt. Dazu wurden vom Raspberry Pi 10000 Nachrichten an sich selbst gesendet und überprüft, wie lange es dauert, bis diese wieder ankommen. Aufgrund dieser Resulate kann davon ausgegangen werden, dass die Performance-Anforderungen erreicht werden.

Minimum	46.04 ms
Maximum	282.11 ms
Median	55.37 ms
Durchschnitt	59.58ms

5. Protokoll

Für die Kommunikation zwischen App, Röster und Admin Interface musste ein Kommunikationsprotokoll geschaffen werden. Dieses Protokoll basiert auf MQTT, da dies der derzeitige Standard in der IoT Entwicklung ist. MQTT ist sehr leichtgewichtig und verwendet einen publish und subscribe Mechanismus, durch den verschiedene Clients ein Topic abonnieren können. Da die Topics hierarchisch aufgebaut sind, ist es möglich Wildcards einzusetzen. Dadurch können bestimmte Sensordaten von allen Clients auf einmal abgefragt werden.

```
Geräretyp
    Thingname
        Sensor
            TEMPERATURE_1
            TEMPERATURE_2
            TEMPERATURE_3
            TEMPERATURE_4
            DRUM_SPEED
            FAN_SPEED
            FLAP_FUNNEL_POSITION
            FLAP_CHASSIS_POSITION
            ...
        Command
            startRoasting
            cancelRoasting
            ...

Management
    Command
        grantAccess
        ...
```

Die oben abgebildete Hierarchie beschreibt den Aufbau des Protokolls. Gerätetyp steht dabei für einen String welcher den Typ des Rösters definiert. Der Thingname ist eine eindeutige Identität die jeder Maschine bei der Erfassung zugewiesen wird. Management und seine Subtopics wurden für Verwaltungsaufgaben reserviert. Die mit drei Punkten (...) gekennzeichneten Topics beschreiben eine mögliche Erweiterung mit weiteren Topics.

Da AWS nur ein MQTT Protokoll mit Einschränkungen erlaubt musste das Protokoll dementsprechend angepasst werden. Vor allem folgende Punkte mussten berücksichtigt werden.

1. Das Retain Flag kann nicht gesetzt werden.
2. Das Quality of Service Level kann nur 0 oder 1 sein aber nicht 2.

3. die Nachrichten Reihenfolge ist nicht garantiert.

5.1. Sensor

Das Sensor Topic ist das Über-Topic für alle Sensordaten des Rösters, der Subscriber muss sich also auf einen spezifischen Sensor beziehen. Durch diese Sensoren kann die Überwachung stattfinden. Alle Sensor Topics haben einen ähnlichen Aufbau. Das Quality of Service Flag wird auf 0 gesetzt. Das heisst eine Nachricht wird maximal einmal versendet. Dies ist möglich da es keine grossen Konsequenzen mit sich zieht wenn ein Eintrag verloren geht und es wird weniger Speicher auf dem Röster besetzt. Die Payload beinhaltet den Value des Sensor. Dieser Value passt sich den Sensor Daten an.

```
Quality of Service  : 0
Payload             : value
```

5.2. Command

Da ein Command nur exakt einmal ausgeliefert werden sollte, wäre hier das Quality of Service Level 2 notwendig. Da dies nicht möglich ist wird eine Alternative definiert. Jeder Command hat ein Unter-Topic `/accepted` und in diesem werden Bestätigungen vom Röster gesendet. Der Röster bestätigt also die Ankunft eines Commands. Damit die Commands identifiziert werden können wird in der Payload zusätzlich eine ID mitgesendet. Die erste ID wird zufällig generiert und dann um eins inkrementiert. Durch dies kann überprüft werden ob das Command auch wirklich den Röster erreicht hat. Auf dem Röster kann durch die CommandId auch eine Reihenfolge der Commands ermittelt werden.

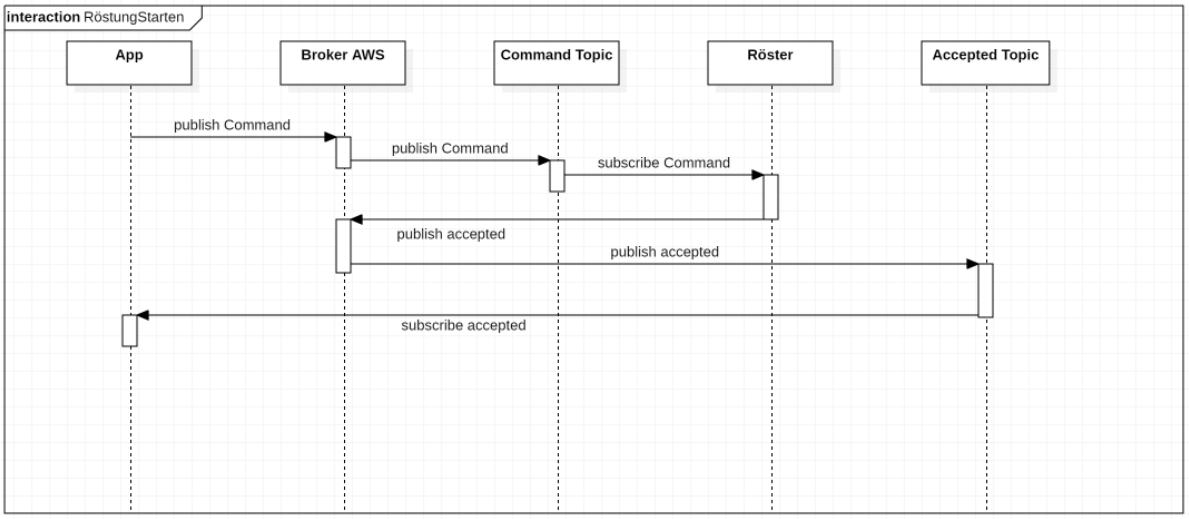


Abbildung 5.1.: Sequenzdiagramm: Command von App an Röster senden

Bei dem Sequenzdiagramm wird der Ablauf des Commands gezeigt. Dabei wird angenommen, dass die App bereits eine Subscription auf das Accepted Topic und der Röster auf das Command Topic gesetzt hat. Der Mechanismus jeder Nachricht wurde mit publish oder subscribe angegeben.

5.2.1. Röstung starten

Die Röstung starten wird durch das Topic `startRoasting` verkörpert. Die Payload beschreibt das Röstungsprofil.

```
Quality of Service  : 1
Payload             : {Roast Profile, CommandId}
```

5.2.2. Röstung abbrechen

Die Röstung abbrechen wird durch das Topic `cancelRoasting` verkörpert. Als Payload wird die CommandId mitgesendet.

```
Quality of Service  : 1
Payload             : {CommandId}
```

5.3. Accepted und Rejected

Die Command Topics verfügen jeweils über zwei Sub-Topics "accepted" und "rejected". Diese Topics werden verwendet, um den Erhalt der Commands zu bestätigen. Da nicht garantiert werden kann, dass ein Command ankommt, und es selbst dann zu einem Fehler im System kommen kann, werden diese Topics zur Hilfe gezogen. Dieser Mechanismus ist analog zu den von AWS bereitgestellten Topics umgesetzt. Ein Client welcher Commands versendet soll sich zusätzlich auf diese zwei Topics subscriben. Sendet ein Client nun einen Command, kommt über eines dieser beiden Topics eine Antwort. Auf Client Seite kann mit der CommandId eingesehen werden ob es sich tatsächlich um die richtige Anfrage handelt.

```
Quality of Service  : 1
Payload             : {CommandId} | {CommandId, err, msg}
```

5.4. Status des Röster

Der Status beschreibt ob eine Verbindung zum Internet besteht und ob der Röster gerade am rösten ist. Bei dieser Nachricht sollte das Retain Flag auf True gesetzt sein. Da dies aber nicht möglich ist, wird eine Alternative definiert. AWS bietet Shadow Objekte[2] an, die den letzten bekannten State des Gerätes wiederspiegeln, auch wenn das Gerät nicht mehr verbunden ist. Über diesen Shadow soll nun der Status gespeichert werden. Der Shadow kann über das Topic `$aws/things/Thingname/shadow/update` angesprochen werden. Der Shadow besteht aus drei Objekten. Dem desired, reported und delta Objekt. In diesem Protokoll wird lediglich mit dem reported Status gearbeitet, um den aktuellen Status zu erhalten. Der Status des Röster kann auf drei Werte gesetzt werden:

1. Offline: Der Röster ist ausgeschaltet und/oder kann keine Verbindung zu AWS herstellen.
2. Online: Der Röster ist eingeschaltet und mit AWS verbunden.
3. Roasting: Der Röster führt derzeit eine Röstung durch. In diesem Zustand kann das aktive Röstprofil über den Key `roastProfile` eingesehen werden.

```
{
    "desired": { ... },
    "reported": {
        "status": "Offline" | "Online" | "Roasting",
        "roastProfile": { ... }
    }
    "delta": { ... }
}
```

5.5. Management

Die Management-Topics sind für Verwaltungsaufgaben vorgesehen. Sie sind nicht an einen einzelnen Röster gebunden.

5.5.1. grantAccess

Für UC01 - Röster einrichten wird ein Mechanismus benötigt, um einem Benutzer Zugriff auf einen bestimmten Röster zu gewähren. Dazu sendet ein Röster die benötigten Daten an dieses Topic was mittels einer Rule eine AWS Lambda Funktion auslöst, die dem Nutzer die gewünschten Rechte gibt.

```
Quality of Service   : 1
Payload              : {identityId, thingName}
```

5.6. Erweiterbarkeit

Das Protokoll kann einfach erweitert werden, ohne dass dies die bisherige Kommunikation beeinflusst. Topics erfordern keine spezielle zusätzliche Konfiguration über AWS.

5.6.1. Erweiterung Command

Soll ein neuer Command definiert werden, kann dies mit einem neuen Topic gemacht werden. Neue Commands sollten weiterhin eine CommandId als Payload mitsenden.

5.6.2. Erweiterung Sensor

Wird ein neuer Sensor hinzugefügt, kann ein neues Topic unter Sensor definiert werden.

5.7. Sicherheit

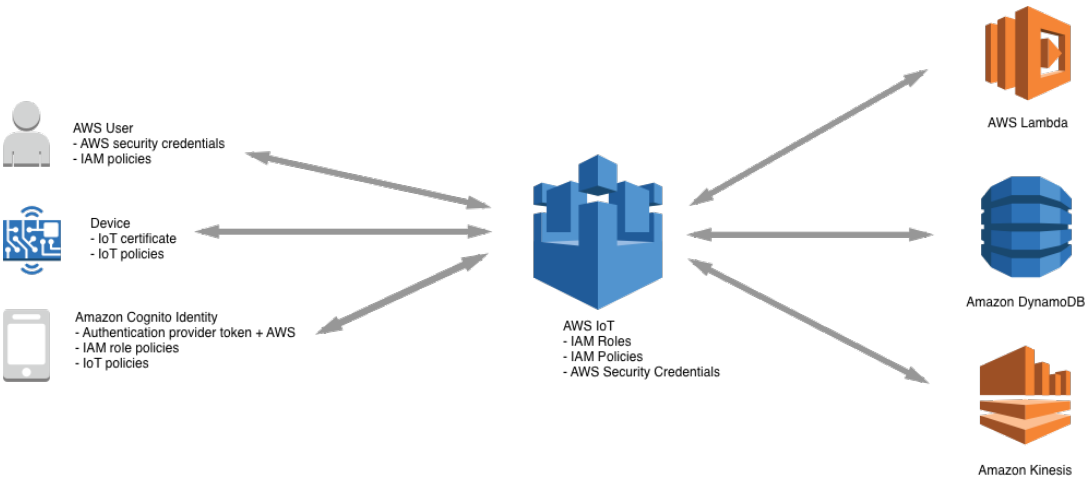


Abbildung 5.2.: Sicherheitsmechanismen in AWS IoT Core [3]

Damit ein Client Zugriff auf Topics erhalten kann, muss er zwei Phasen durchlaufen. In der Authentifizierung wird die Identität des Clients überprüft. Danach erfolgt die Autorisierung, die dem Client eine Zugriffserlaubnis für bestimmte Topics ausstellt.

5.7.1. Authentifizierung

Zur Authentifizierung eines Things verwendet AWS IoT Core eine Public-Key-Infrastruktur bereit. Die darüber ausgestellten Zertifikate verwenden das RSA Verfahren mit einem 2048 bit langen Schlüssel.

Für die Authentifizirung eines Users über die App und das Admin Interface wird AWS Cognito verwendet. [3]

5.7.2. Autorisierung

Für die Autorisierung stellt AWS IoT Core Policies zur Verfügung. Dabei handelt es sich um ein JSON-Dokument, in dem die Zugriffsrechte definiert werden. Eine Policy kann einem Thing oder einer Cognito Identity zugewiesen werden.

Für die Röster gibt es eine einzige Policy, die den Publish- und Subscribe-Zugriff auf `Gerätename/Thingname/*` erlaubt. Zudem erhalten alle Röster Zugriff auf die Management Topics.

Für das Admin Interface steht eine Policy bereit, die der jeweiligen Cognito Identity Vollzugriff auf alle Topics gewährt.

Für Nutzer der App wird jeweils eine eigene Policy erstellt, die der jeweiligen Cognito-Identity Zugriff auf `Gerätename/Thingname/*` eines bestimmten Rösters erlaubt. Da für jeden Röster eine eigene Policy verwendet wird und AWS IoT Core eine Limite von 10 Policies pro Cognito Identity hat, kann ein Nutzer maximal 10 Röster haben. [1]

6. Admin Interface und Library

Das Admin Interface dient zur Überwachung aller verbundenen Röster mit AWS, sowie des Röstvorgang eines einzelnen Rösters. Für genauere Informationen, siehe UC03 - Röstflotte überwachen und UC04 - Röstvorgang überwachen. Die Informationen, die das Interface anzeigt, kommen von der Library.

6.1. Admin Interface

Das Admin Interface ist eine Web Applikation, die mit React und Typescript umgesetzt wurde. Das Interface besteht aus zwei Seiten (Overview, Details), die aus wiederverwendbaren Komponenten bestehen. Dazu kommt noch eine Login-Seite von AWS.

Die Ordnerstruktur sieht wie folgt aus. Bei Amplify handelt es sich um AWS Tools, welche die Backend Konfiguration unterstützen.

```
amplify
    backend
public
    images
    favicon.ico
    index.html
src
    pages
    components
    library
    styles
    App.tsx
    index.tsx
```

6.1.1. Login

Die Login-Seite ermöglicht es, sich mit einem Konto ins Admin Interface einzuloggen. Es war eine bewusste Entscheidung, dass kein neues Konto via die Web Applikation erstellt werden kann. Der Grund dafür ist, dass sich nur bestimmte Personen ins Admin Interface einloggen können. Ein neues Konto kann nur über die AWS Console erstellt werden.

Amplify

Das Login wurde mit Amplify umgesetzt. Amplify erlaubt ein Amazon Login direkt als React Komponente zu verwenden, ohne dass der Entwickler einen grossen Mehraufwand hat. Über die AWS Console können verschiedenen Userpools erstellt werden und Einstellungen vordefiniert werden. Diese können dann in verschiedenen Projekten verwendet werden.

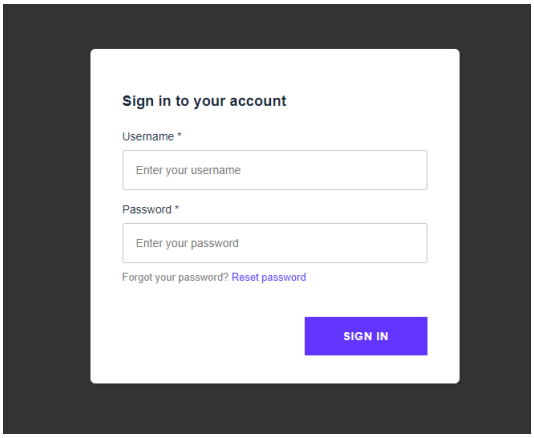


Abbildung 6.1.: Login Seite

6.1.2. Komponente

Komponente von React sind wie Bausteine, die für die Erstellung einer Seite gebraucht werden. Diese Bausteine sind so klein wie möglich gehalten worden, damit der Aufbau der Applikation übersichtlich bleibt. Es folgt eine Übersicht mit allen verwendeten Komponenten.

```
Overview
    FilterBar.tsx
    RoasterIcon.tsx
    RoasterList.tsx
Details
    SensorGraph.tsx
    SensorTable.tsx
    SensorValue.tsx
Design
    Header.tsx
    ActivityIndicator.tsx
```

Komponente unter 'Overview' und 'Details' wurden für die Erstellung der respektiven Seiten gemacht. Diese werden im Teil 'Overview' bzw. 'Details' genauer beschrieben. Unter 'Design' sind die Komponente zu finden, die nicht unbedingt zu einer bestimmten Seite gehören. Der

Header besteht aus dem Logo, dem Seitentitel und der Logout Möglichkeit. Das Logo verlinkt auf die Übersichtsseite. Der ActivityIndicator ist der Ladetext mit Ladeicon. Er wird zurzeit zwar nur auf der Übersichtsseite verwendet, falls das Admin Interface aber zu einem späteren Zeitpunkt erweitert werden soll, kann der ActivityIndicator gut weiterverwendet werden.

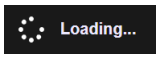


Abbildung 6.2.: Ladeanzeige

6.1.3. Overview

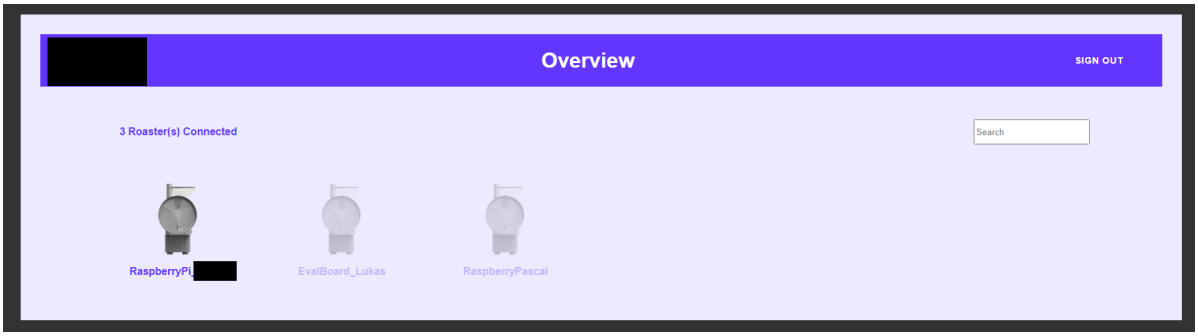


Abbildung 6.3.: Admin Interface - Übersicht

Die Seite 'Overview' zeigt verbundene Röster mit dem AWS an. Beim Klicken auf einen Röster wird die Detailsseite dieses Rösters angezeigt. Die Übersicht besteht aus den drei Komponenten FilterBar, RoasterIcon und RoasterList. Das RoasterIcon ist eine Komponente, welches den Namen des Röster und seinen Status (online/offline) anzeigt. Die RoasterList ist eine Liste der verschiedenen Röster. Die Liste beinhaltet mehrere RoasterIcons. Die FilterBar zeigt die Anzahl der verbundenen Röster an. Zudem kann nach bestimmten Röstern mit der Suchfunktion gesucht werden. Falls viele Röster mit AWS verbunden sind und nicht alle mit der ersten Abfrage angezeigt werden können, besteht die Option 'Load more Roasters'. Für weitere Informationen, siehe den Teil 'GetMoreRoasters' in 'Library'.

6.1.4. Details

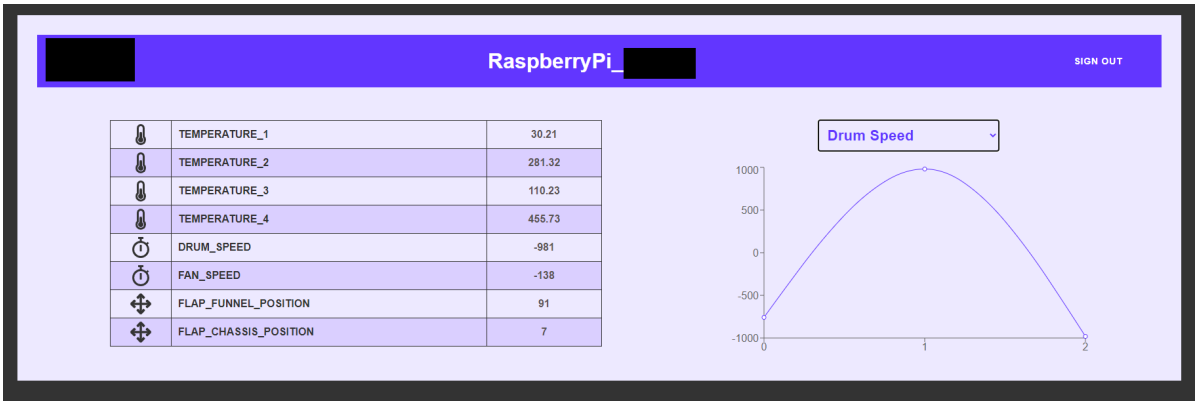


Abbildung 6.4.: Admin Interface - Detailansicht

Die Detailsseite wird dynamisch generiert und zeigt aktuelle Messwerte eines Rösters an. Die Seite 'Details' besteht aus den Komponenten SensorGraph, SensorTable und SensorValue. SensorValue besteht aus einem Icon, dem Sensornamen und dem gemessenen Sensorwert. Je nach Typ des Sensors wird ein anderes ausgewählt. Es gibt vier verschiedene Icons, ein Default Icon und je eines für Geschwindigkeit, Temperatur und Position. SensorTable ist die Tabelle mit den aktuellen Sensorwerten. Jede Reihe besteht aus einem SensorValue. Der SensorTable gibt dem SensorValue den Röster, den Sensornamen und den Sensortypen mit. Der SensorGraph ist eine Liniengrafik, die laufend Sensorwerte anzeigt. Aus einem Dropdown kann zwischen den Sensoren ausgewählt werden. Da die gemessenen Daten nicht zwischengespeichert werden, wird die Liniengrafik bei einem Wechsel des Sensor keine vorherigen Werte anzeigen, sondern nur zukünftige.

6.1.5. Style

```
$primary-color: #6236ff;
$secondary-color: #333333;

$primary-color-shadow: darken($primary-color, 5%);
$primary-color-light: lighten($primary-color, 30%);
$primary-color-lighter: lighten($primary-color, 35%);
$secondary-color-light: lighten($secondary-color, 30%);
```

Listing 6.1.: Farbdefinition in SCSS

Die Styles wurden mit scss erstellt. Dies ermöglicht die Verwendung von Variablen. Falls das Design geändert werden soll, können lediglich die Variablen 'primary-color' und 'secondary-

color' verändert werden. Alle anderen Stufen dieser Farben werden automatisch berechnet und es besteht kein Mehraufwand.

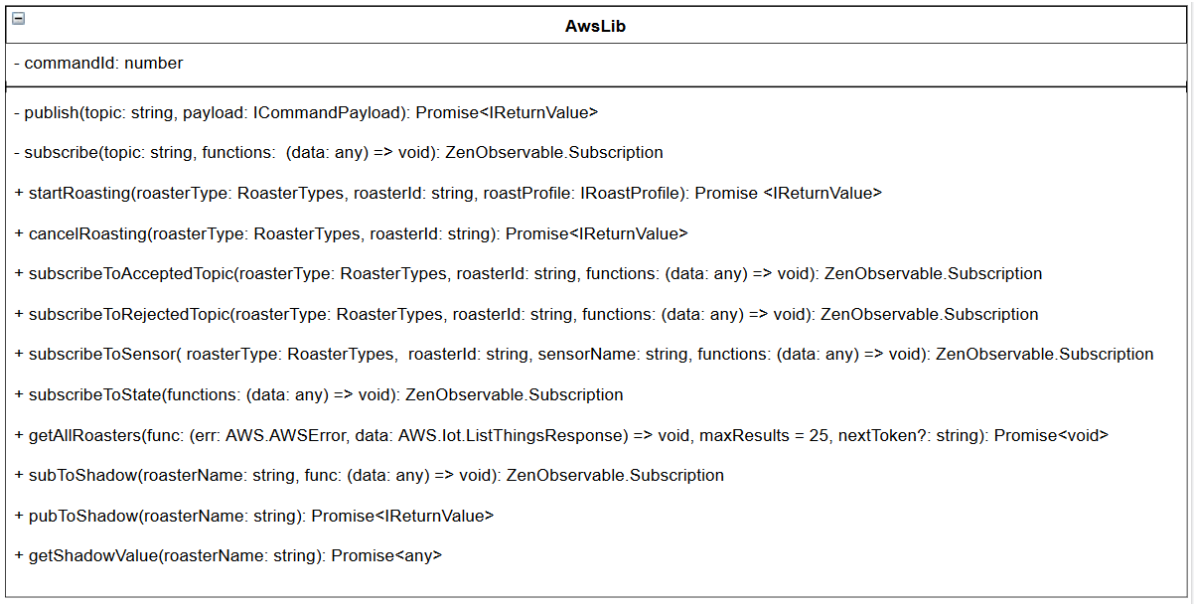
6.1.6. Index und Routing

Im 'index.tsx' File befindet sich die Amplify Konfiguration und die React '.render'-Methode, die für die Web Applikation benötigt wird. Im File App.tsx befindet sich das Routing. Dort wird festgelegt, welche Adresse angeben werden muss, um auf eine bestimmte Seite des Admin Interfaces zu gelangen. Die Funktion 'withAuthenticator()' ist für die Login Seite zuständig.

6.2. Library

Die Web Applikation wird in React geschrieben, während die App mit React Native geschrieben wird. Beide diese Technologien verwenden JavaScript, respektive TypeScript. Viele Features der beiden Platformen überschneiden sich. Deswegen wurde sich für eine Library entschieden, die in beide System eingebunden werden kann. Die Library soll alle notwendingen Funktionen der beiden Applikationen implementieren. Aus zeitlichen Gründen wurden die Änderung in der App nicht vorgenomen und nur die Library geschrieben. Diese kann aber einfach in die App integriert werden.

Die Library besteht aus einem Field und aus zwölf Funktionen und wird als Klasse dargestellt.



6.2.1. publish

Eine private Funktion die die publish Funktion des AWS Framework umspannt. Dabei werden Fehler abgefangen. Sie soll eine Nachricht auf ein Topic veröffentlichen.

Parameter:

1. topic (string): Der Name des Topics, auf welches eine Nachricht veröffentlicht werden soll.
2. payload (ICommandPayload): Die Nachricht, die auf dem Topic veröffentlicht werden soll.

Rückgabewert: Ein Rückgabeobjekt, welches beschreibt ob die Funktion erfolgreich war oder nicht. Bei eine Fehler wird dieser auch mitgegeben.

6.2.2. subscribe

Eine private Funktion, die die subscribe Funktion des AWS Framework umspannt.

Parameter:

1. topic (string): Der Name des Topics, welches abonniert werden soll.
2. function (data: any => void): Eine Funktion, die ausgeführt wird wenn eine neue Nachricht im Topic veröffentlicht wurde.

Rückgabewert: Ein Subscriber Element, mit welchem die Subscription terminiert werden kann mit dem Aufruf von `unsubscribe()`.

6.2.3. startRoasting

Eine public Funktion, die ein Rösterprofil als Nachricht auf das Topic `startRoasting` sendet.

Parameter:

1. roasterType (RoasterType): Der Typ des Röster, für welchen das Command ausgeführt werden soll.
2. roasterId (string): Der Thing Name des Röster, für welchen das Command ausgeführt werden soll.
3. roastProfile (IRoastProfile): Die Art der Röstung, die durchgeführt werden soll.

Rückgabewert: Ein Rückgabeobjekt, welches beschreibt, ob die Funktion erfolgreich war oder nicht. Bei eine Fehler wird dieser auch mitgegeben.

6.2.4. cancelRoasting

Eine public Funktion, die ein Command auf einem spezifischen Röster stoppt.

Parameter:

1. roasterType (RoasterType): Der Typ des Röster, für welchen das Command ausgeführt werden soll.
2. roasterId (string): Der Thing Name des Röster, für welchen das Command ausgeführt werden soll.

Rückgabewert: Ein Rückgabeobjekt, welches beschreibt, ob die Funktion erfolgreich war oder nicht. Bei eine Fehler wird dieser auch mitgegeben.

6.2.5. subscribeToAcceptedTopic

Eine public Funktion, die das Accepted Topic eines Röster abonniert.

Parameter:

1. roasterType (RoasterType): Der Typ des Rösters, für welchen das Topic abonniert werden soll.
2. roasterId (string): Der Thing Name des Rösters, für welchen das Topic abonniert werden soll.
3. function (data: any => void): Eine Funktion, die ausgeführt wird, wenn eine neue Nachricht im Topic veröffentlicht wurde.

Rückgabewert: Ein Subscriber Element, mit welchem die Subscription terminiert werden kann mit dem Aufruf von `unsubscribe()`.

6.2.6. subscribeToRejectedTopic

Eine public Funktion, die das Rejected Topic eines Röster abonniert.

Parameter:

1. roasterType (RoasterType): Der Typ des Röster, für welchen das Topic abonniert werden soll.
2. roasterId (string): Der Thing Name des Röster, für welchen das Topic abonniert werden soll.
3. function (data: any => void): Eine Funktion, die ausgeführt wird, wenn eine neue Nachricht im Topic veröffentlicht wurde.

Rückgabewert: Ein Subscriber Element, mit welchem die Subscription terminiert werden kann mit dem Aufruf von `unsubscribe()`.

6.2.7. subscribeToSensor

Eine public Funktion, die ein Sensor Topic eines Röster abonniert.

Parameter:

1. roasterType (RoasterType): Der Typ des Rösters, für welchen das Topic abonniert werden soll.
2. roasterId (string): Der Thing Name des Rösters, für welchen das Topic abonniert werden soll.
3. sensorName (string): Der Name des Sensors, welcher abonniert werden soll.
4. function (data: any => void): Eine Funktion, die ausgeführt wird, wenn eine neue Nachricht im Topic veröffentlicht wurde.

Rückgabewert: Ein Subscriber Element, mit welchem die Subscription terminiert werden kann mit dem Aufruf von `unsubscribe()`.

6.2.8. getAllRoasters

Eine public asynchrone Funktion, die alle Things bzw. Röster aus AWS zu holen. Diese Request verwendet eine Pagination.

Parameter:

1. func ((err: AWS.AWSError, data: AWS.Iot.ListThingsResponse) => void): Die Funktion, die aufgerufen werden soll, nachdem alle Daten erhalten wurden. Das data Element beinhaltet eine Liste der Things, sowie ein NextToken, falls noch mehr Röster verfügbar sind.
2. maxResults? (number): Die maximale Anzahl an Resultaten, die pro Request zurückkommen (standardmässig 25, maximum 60).
3. nextToken? (stirng): Ein NextToken, welches die nächsten Röster beantragt.

Rückgabewert: Diese Funktion hat keinen Rückgabewert.

6.2.9. subToShadow

Eine public Funktion, die ein Status Topic bzw. den Shadow eines Röster abonniert.

Parameter:

1. roasterName (string): Der Thing Name des Röster, für welchen das Topic abonniert werden soll.
2. func (data: any => void): Eine Funktion, die ausgeführt wird, wenn eine neue Nachricht im Topic veröffentlicht wurde.

Rückgabewert: Ein Subscriber Element, mit welchem die Subscription terminiert werden kann mit dem Aufruf von `unsubscribe()`.

6.2.10. pubToShadow

Eine public Funktion, die eine Nachricht auf das Status Topic bzw. den Shadow eines Röster sendet. Dadurch wird eine Nachricht auf das Subscribe Topic des Shadow gepostet.

Parameter:

1. roasterName (string): Der Thing Name des Röster, für welchen ein Status angefordert werden soll.

Rückgabewert: Ein Rückgabeobjekt, welches beschreibt ob die Funktion erfolgreich war oder nicht. Bei eine Fehler wird dieser auch mitgegeben.

6.2.11. getShadowValue

Eine asynchrone public Funktion, die direkt den Status des Röster zurückgibt.

Parameter:

1. roasterName (string): Der Thing Name des Röster, für welchen ein Status angefordert werden soll.

Rückgabewert: Ein Promise, welches entweder den Shadow oder einen Fehler zurückgibt.

7. Röster Simulation

Die Röster Simulation wird verwendet um die Schnittstellen zu implementieren über die später ein echter Röster Befehle entgegennehmen und Daten senden soll.

7.1. Hardware

Es war geplant, den Code der später auf dem Röster läuft auf einem EvalBoard (STMicroelectronics B-L475E-IOT01A[4]) zu implementieren. Auf den ersten Blick scheint dies keine schwere Aufgabe zu sein, da STMicroelectronics mehrere IDEs (STM32 Cube IDE, Atollic True Studio, SW4STM32), Libraries für die Hardware (STM32CubeL4), sowie eine Library (X-CUBE-AWS) zur Kommunikation mit AWS anbietet. Die STM32 Cube IDE wurde gewählt, da sie die anderen Optionen ablöst [18].

Das erste Problem war, dass der aktuellen Version 2.0.0 von X-CUBE-AWS keinerlei Beispiele für den verwendeten Microcontroller beilagen. Somit musste auf die Vorgängerversion 1.4.1 zurückgegriffen werden. Dieses Paket enthält Beispiel-Code für eine Vielzahl an Boards, die jedoch nur als Projekte für verschiedene IDEs vorliegen, nicht jedoch für die STM32 Cube IDE. Dieses Problem liess sich lösen, indem das Projektformat des STM32 Cube IDE Vorläufers SW4STM32 importiert wurde.

Die Beispiel-Projekte verwenden intern sehr viele Verknüpfungen von Dateien, was dazu führt, dass die Anordnung der Dateien auf dem Dateisystem stark vom eigentlichen Projektaufbau abweicht. Um den Code übersichtlich zu halten, wurde ein komplett neuer Projektaufbau erstellt. Dazu mussten die relevanten Dateien aus dem X-CUBE-AWS Paket identifiziert und an die richtige Stelle kopiert werden. Dies wurde dadurch erschwert, dass gewisse Komponenten sowohl in STM32CubeL4 als auch in X-AWS-CUBE in unterschiedlichen Versionen vorhanden waren, was zu Inkompatibilitäten führte.

Nachdem der Projektaufbau fertiggestellt war und sich ohne Fehler kompilieren liess, wurde als erstes versucht, eine WLAN Verbindung aufzubauen. Dies klappte nicht. Mit dem Debugger konnte nachvollzogen werden, dass bei der Initialisierung des WLAN-Moduls ein unerwarteter Interrupt ausgelöst wird, der das Programm zum Absturz bringt. Versuche herauszufinden, wieso der Interrupt ausgelöst wurde, blieben erfolglos.

Da zu diesem Zeitpunkt bereits die Halbzeit des Projekts überschritten war, wurde beschlossen statt einem Microcontroller auf den Raspberry Pi 4[16] zu setzen. Die folgenden Abschnitte beschreiben den Aufbau der Software auf dem Raspberry Pi.

7.2. Übersicht

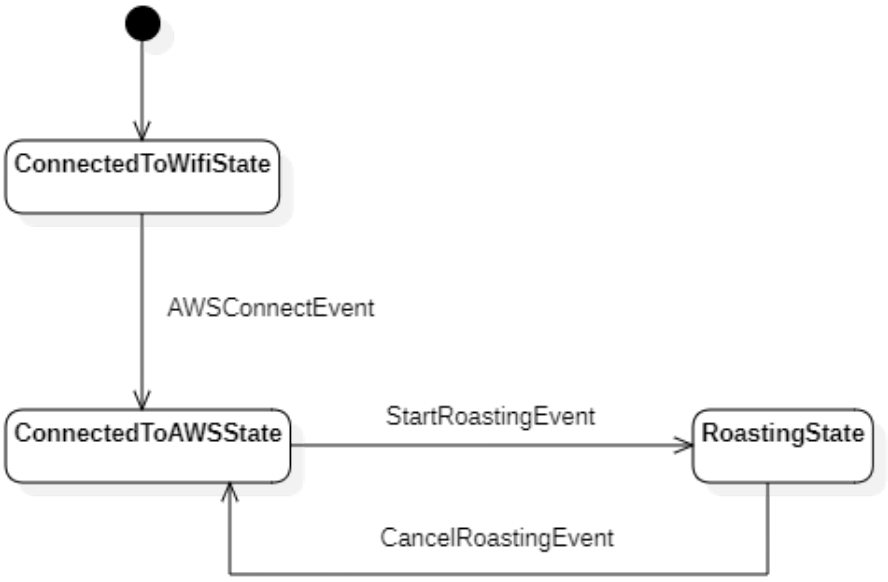


Abbildung 7.1.: Zustandsdiagramm Röster

Der Code auf dem Röster wurde als State-Machine implementiert. Dieser Ansatz wurde gewählt, um einfache Möglichkeiten zur nachträglichen Erweiterung zu bieten. Abbildung 7.1 gibt einen Überblick über die möglichen Zustände und Übergänge. Zusätzlich zu den im Diagramm gezeigten Übergängen besteht immer die Möglichkeit, den Röster abzuschalten.

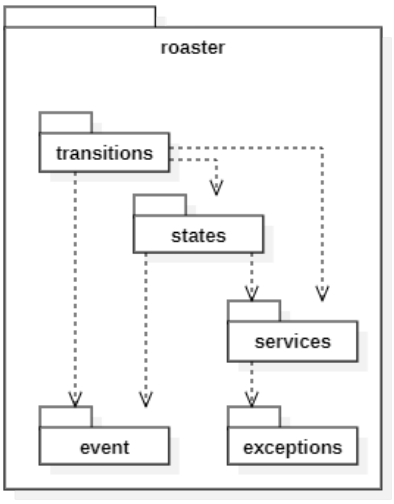


Abbildung 7.2.: Packagediagramm Röster

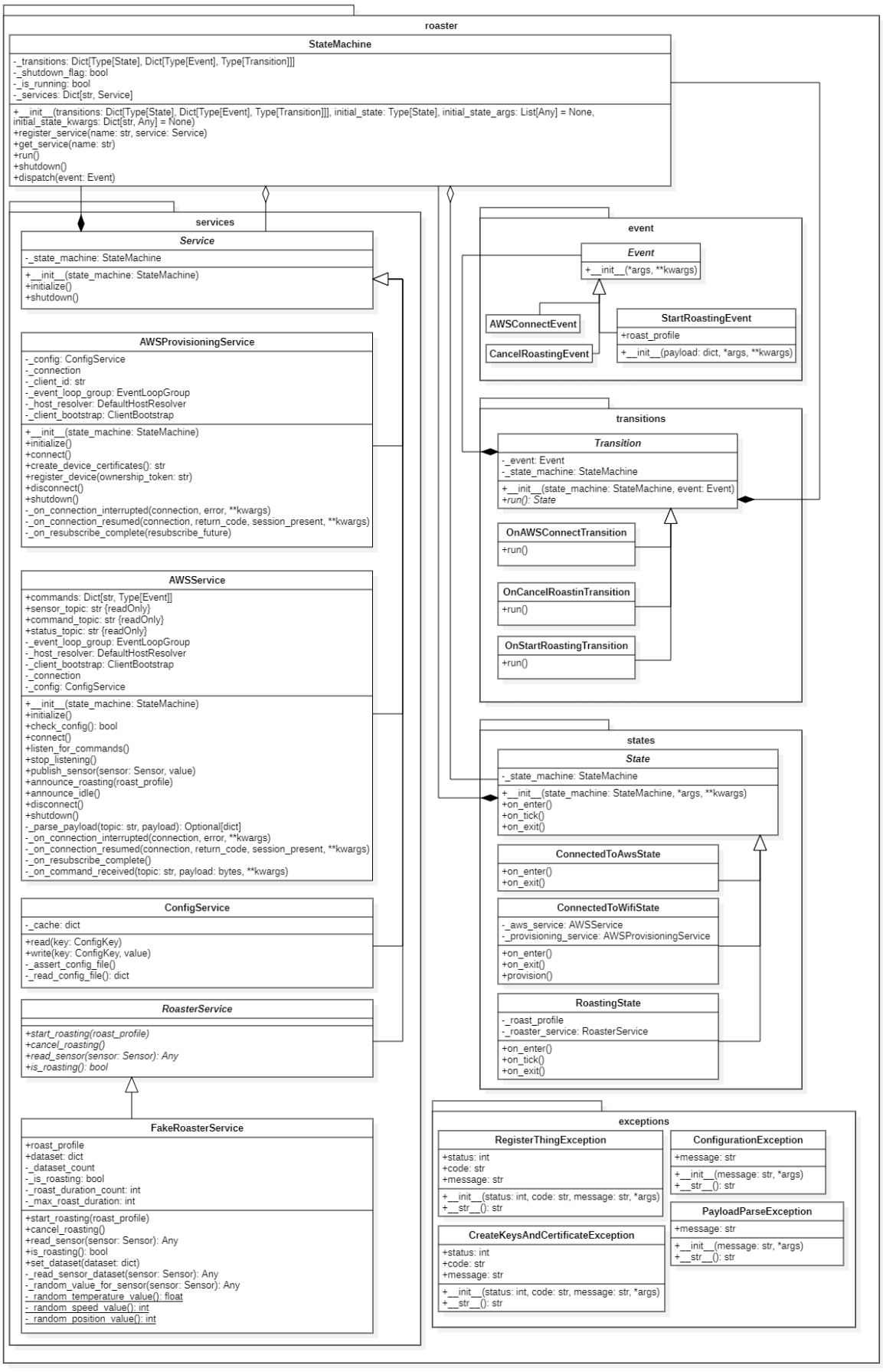


Abbildung 7.3.: Klassendiagramm Röster

7.3. Konzepte

Als Inspiration für die Architektur einer State Machine diente der Artikel "Implementing a simple state machine library in JavaScript" [6].

7.3.1. State

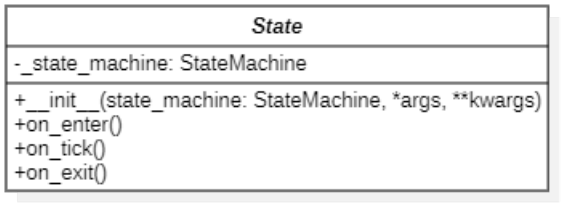


Abbildung 7.4.: Klassendiagramm State

State ist eine abstrakte Basisklasse, von der alle Zustände erben. Die Methoden `on_enter` und `on_exit` werden jeweils beim betreten und verlassen des Zustands von der StateMachine aufgerufen. Die `on_tick` Methode wird während des Verweilens in diesem Zustand in einer Endlosschleife aufgerufen.

7.3.2. Event

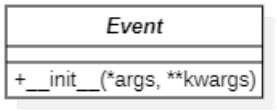


Abbildung 7.5.: Klassendiagramm Event

Events werden an die StateMachine gesendet, die aufgrund des Datentyps eine Transition auslöst. Falls für eine Transition zusätzliche Daten benötigt werden, können diese als Attribut im Event übermittelt werden.

7.3.3. Transition

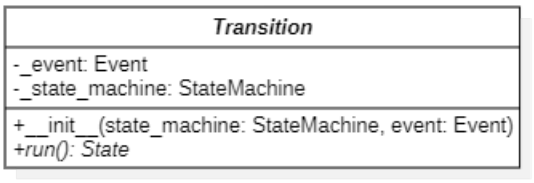


Abbildung 7.6.: Klassendiagramm Transition

Eine Transition wird für den Übergang von einem State in einen Anderen verwendet. Der neue State wird von der abstrakten `run` Methode erzeugt, die von allen Transitions implementiert werden muss. Falls für die Erzeugung des States zusätzliche Daten benötigt werden, müssen diese im Event übergeben werden.

7.3.4. Service

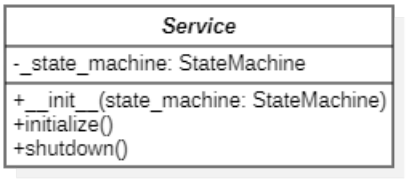


Abbildung 7.7.: Klassendiagramm Transition

Services stellen Funktionen zur Verfügung, die von States und Transitions genutzt werden können. Die Methoden `initialize` und `shutdown` werden jeweils beim Starten und Beenden der StateMachine aufgerufen.

7.4. StateMachine

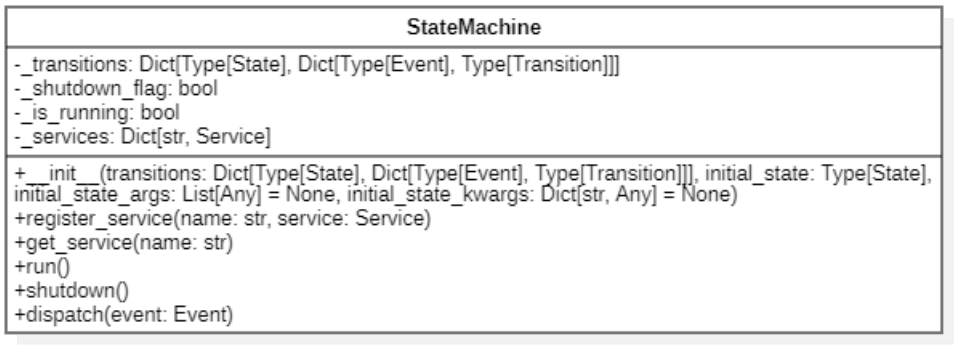


Abbildung 7.8.: Klassendiagramm StateMachine

Die Klasse StateMachine ist die Kernkomponente des Codes. Sie übernimmt die Verwaltung der verschiedene Zustände und Services und sorgt dafür, dass das Programm im Falle eines unerwarteten Fehlers ordnungsgemäss beendet wird.

Konstruktor Der Konstruktor erhält als ersten Parameter die Zuordnung, welche Events in welchem State verarbeitet werden können und welche Transition sie auslösen. Als zweiter Parameter wird die Klasse des initialen States übergeben. Die beiden letzen Parameter werden an den Konstruktor des initialen States weitergeleitet.

register_service Diese Methode fügt der StateMachine einen Service hinzu. Der Parameter "name" wird später für den Zugriff auf diesen Service verwendet.

get_service Diese Methode erlaubt den Zugriff auf einen bestimmten Service.

run Die Methode `run` startet die Ausführung der StateMachine.

shutdown `shutdown` beendet die Ausführung der StateMachine.

dispatch Über diese Methode werden Events an die StateMachine übergeben. Ist für den übergebenen Event im aktuellen State eine Transition registriert, wird diese ausgeführt und in einen anderen State übergegangen.

7.5. Services

AWSProvisioningService

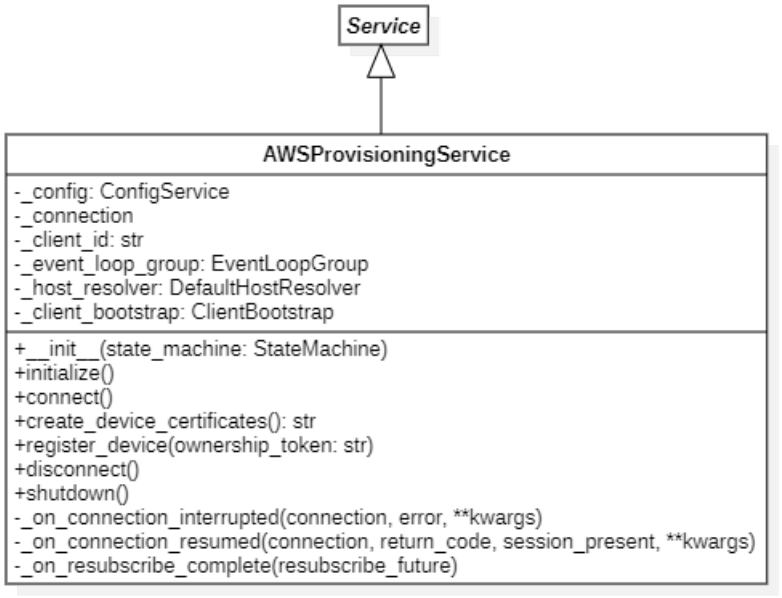


Abbildung 7.9.: Klassendiagramm AWSProvisioningService

Der AWSProvisioningService implementiert die benötigten Funktionen für die erste Anmeldung des Rösters an AWS IoT Core. Dafür werden auf dem Röster das Provisionierungs-Zertifikat und eine einmalige Seriennummer vorinstalliert. Damit ist der Röster in der Lage, sich auf AWS ein Gerätezertifikat ausstellen zu lassen und sich damit als Thing zu registrieren.

AWSService

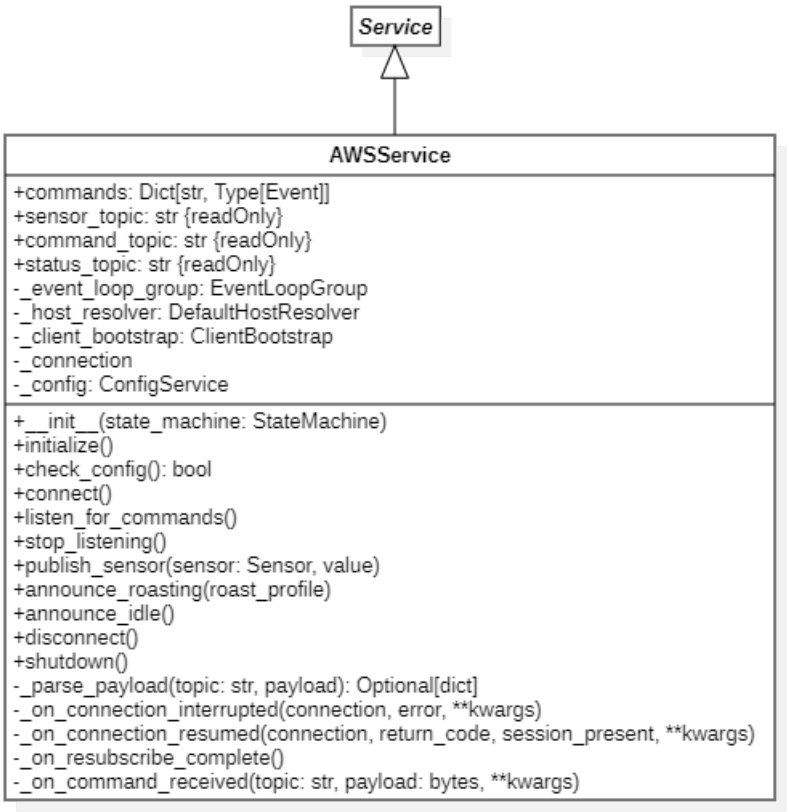


Abbildung 7.10.: Klassendiagramm AWSService

Der AWSService stellt die Funktionalitäten für das Empfangen von Commands und Senden von Sensordaten bereit. Wurde der Röster erstmalig mit dem AWSProvisioningService angemeldet, kommt ausschliesslich dieser Service für die Kommunikation mit AWS IoT Core zum Einsatz.

check_config Diese Methode überprüft, ob der Röster bereits für die Kommunikation mit AWS IoT Core konfiguriert wurde.

listen_for_commands Es wird eine Subscription auf die Command-Topics erstellt und ein Callback auf `_on_command_received` registriert. Im Callback werden die Commands auf ihre Gültigkeit geprüft und als Event an die StateMachine weitergegeben. Das passiert solange bis der Service mit shutdown beendet wird oder die Methode `stop_listening` aufgerufen wird.

announce_idle & announce_roasting Diese Methoden werden verwendet um den aktuellen Status des Rösters an AWS zu senden, wo sie durch eine Rule im Device Shadow gespeichert werden.

ConfigService

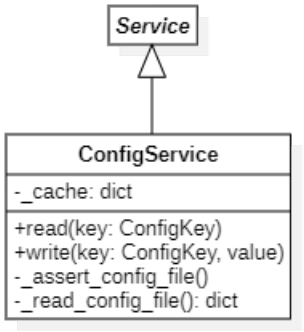


Abbildung 7.11.: Klassendiagramm ConfigService

Der ConfigService ist für das Lesen und Schreiben der Konfiguration zuständig. Die Konfiguration enthält die Verbindungs-Informationen für AWS sowie die Pfade der für die Authentifizierung verwendeten Zertifikate. Damit die Konfiguration nach einem Neustart nicht verloren geht, wird sie in einer JSON-Datei abgelegt. Falls keine Konfigurationsdatei existiert, wird automatisch eine neue Datei mit Standardwerten erstellt.

RoasterService

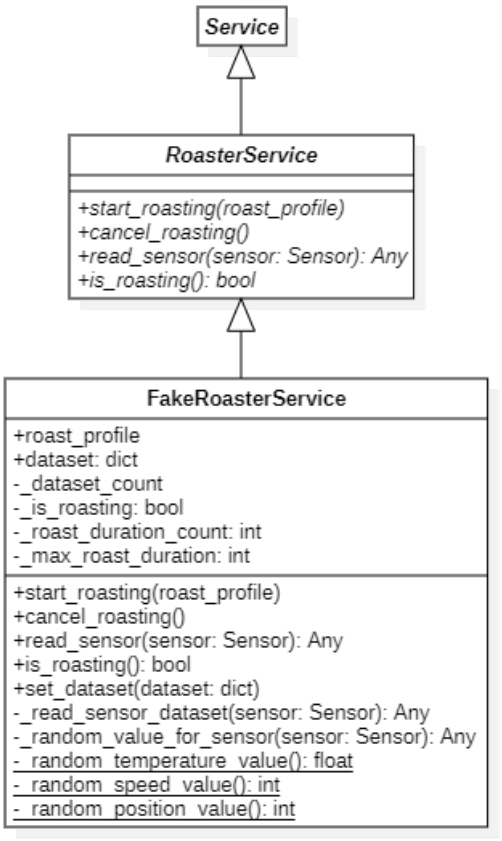


Abbildung 7.12.: Klassendiagramm RoasterService

Die Aufgabe des RoasterServices ist es, die Hardware des Rösters anzusteuern. RoasterService ist eine abstrakte Klasse von der später eine Implementierung abgeleitet werden kann, die effektiv auf die Hardware zugreift. Da für unser Projekt kein Röster zur Verfügung stand, wurde als Ersatz der FakeRoasterService implementiert, der diese Lücke füllt. Der FakeRoasterService kann entweder Zufallsdaten generieren oder zu Testzwecken eine Reihe an vorgegebenen Werten zurückgeben.

8. Reflexion

Im Kapitel "Reflexion" wird zuerst die Planung mit der geleisteten Arbeit verglichen. Dazu gehört ein Ist/Soll-Vergleich der geplanten Sprints und Use Cases, sowie die Auswertung der gebrauchten Arbeitsstunden. Dies soll helfen, in Zukunft genauere Anforderungen und Projektpläne zu schreiben. Anschliessend gibt jedes Teammitglied einen Einblick in die gemachten Erfahrungen während des Projektes.

8.1. Sprints Ist/Soll

Zu Beginn der Arbeit wurde ein Projektplan erstellt. Dieses Kapitel wird nun am Ende der Arbeit geschrieben, um zu überprüfen, inwiefern das Projekt nach Plan verlaufen ist.

8.1.1. Sprints

Sprint 1	(Woche 1)
Soll	Kickoff, Einlesen
Ist	Kickoff, Einlesen
Analyse	Alles nach Plan, da der Projektplan nach diesem Sprint erstellt wurde.

Sprint 2	(Woche 2-3)
Soll	Projektplan, Requirements
Ist	Einlesen, erste Tests, Projektplan, Requirements
Analyse	In diesem Sprint wurde das Projekt geplant, somit entspricht das Projekt auch dem Soll.

Sprint 3	(Woche 4-5)
Soll	Requirements, Architektur, Protokoll definieren
Ist	Requirements definieren, Architektur planen, Protokoll definieren und testen, Eval Board konfigurieren, Ausweichoption überlegen
Analyse	Das Projekt lief ziemlich planmässig und näherte sich der End of Elaboration. Ein unvorhergesehener Punkt war, dass das Eval Board noch nicht wunschgemäss konfiguriert war. Nach stundenlangem Versuchen wurde dem Team bewusst, dass dies ein grösseres Problem darstellen könnte. Daher wurden potentielle Ausweichoptionen gesucht.

Sprint 4	(Woche 6-7)
Soll	Bearbeitung Use Cases Teil 1
Ist	Alternativen finden, Konzepte und Dokumentation anpassen, Röster auf Raspberry Pi simulieren
Analyse	In diesem Sprint wurde klar, dass das Eval Board nicht rechtzeitig bereit sein würde. So wurde sich geeinigt, dass die vom Team vorgeschlagene Ausweichoption umgesetzt werden sollte. Die Planung wurde entsprechend angepasst, dass alles bereit für die Umsetzung der Alternative ist. Dieses Hindernis hat das Projekt um zwei Wochen verzögert.

Sprint 5	(Woche 8-9)
Soll	Bearbeitung Use Cases Teil 2
Ist	Use Cases 02, 03, 04 und 06 sind funktionsfähig
Analyse	Das angepasste Ziel für diesen Sprint war einen ersten Prototyp vorzeigen zu können. Dies wurde erreicht und beinhaltete vier Use Cases.

Sprint 6	(Woche 10-11)
Soll	Testing, Qualitätskontrolle, Beenden Use Cases
Ist	UC05 gemacht, Restliche Use Cases verbessert und finalisiert, Qualitätskontrolle
Analyse	Ein grosser Teil der Testing Pläne fiel ins Wasser, da das meiste mit dem echten Röster zu tun hatte. Die Use Cases (siehe Use Cases Ist/Soll), samt kleiner Qualitätskontrolle konnten trotzdem noch zeitlich abgeschlossen werden.

Sprint 7	(Woche 12-13)
Soll	Reservezeit, Projekt abschliessen, Finale Kontrolle, Dokumentation vervollständigen
Ist	Testing, Dokumentation
Analyse	Das Produkt war zeitlich in Sprint 6 abgeschlossen worden, somit blieb in diesem Sprint mehr Zeit für die Dokumentation. Nur ein kleiner Performance Test der Röster Simulation wurde noch durchgeführt.

Sprint 8	(Woche 14)
Soll	Projekt abgeben, Produkt abliefern
Ist	Finale Kontrolle, Projekt abgeben, Produkt abliefern
Analyse	Dieser Sprint muss eingehalten werden und das hat auch geklappt, somit keine Abweichung von Soll zum Ist. Die finale Kontrolle war in Sprint 7 vorgesehen, wurde jedoch in Sprint 8 gemacht.

8.1.2. Fazit

Nach dem vierten Sprint war das Projekt um zwei Wochen verzögert. Diese Verzögerung hätte noch viel grösser sein können, hätte das Team im Sprint davor keine gute Alternative gefunden. Um den Rückstand wett zu machen, wurde in jedem Sprint auf ein MVP hin gearbeitet und das Wichtigste priorisiert. Zudem hat es geholfen, dass im Projektplan Reserven eingeplant waren. Dies war eine Erkenntnis aus dem Engineeringprojekt. Schlussendlich wurde das Ende der Implementierung rechtzeitig erreicht und es blieb genügend Zeit, die Dokumentation zu vervollständigen.

8.2. Use Cases Ist/Soll

In den vorhergehenden Kapitel wurden die verschiedenen Resultate der Arbeit gelistet (Ist). Als Rückblick ist es interessant, nochmals über alle Use Cases zu gehen (Soll), die zu Beginn des Projektes erstellt wurden und einen Ist/Soll Vergleich zu machen. Für diesen Teil wird davon ausgegangen, dass die Simulation eines Rösters als Ersatz des Rösters gilt. Also wenn die Simulation ein Use Case ausführen kann, dann gilt dies als eine erfolgreiche Implementierung dieses Use Cases.

8.2.1. UC01 Röster einrichten

Da nicht mit dem Röster selbst gearbeitet wurde, sondern nur mit einer Simulation, wurde dieser Use Case nicht abgeschlossen. Das Team hat ursprüngliche Ideen und Konzepte angeschaut, die etwas weiter durchdacht wurden. Ein Ansatz war, die Einrichtung mit Bluetooth zu machen, dies brachte aber eine Reihe von Problemen mit sich. Da die Einrichtung des Rösters ohnehin anders aussehen als die der Simulation, wurde sich auf Einverständnis mit dem Start-Up nicht weiter Zeit darin investiert.

8.2.2. UC02 Röstung starten

Dieser Use Case wurde implementiert. Der Röster kann ein Röstprofil erhalten und über die App gestartet werden.

8.2.3. UC03 Röstvorgang überwachen

Dieser Use Case wurde implementiert. In der Detail-Ansicht im Admin Interface kann ein berechtiger Benutzer einen laufenden Röstvorgang überwachen.

8.2.4. UC04 Röster Flotte überwachen

Dieser Use Case wurde implementiert. Auf der Übersichtsseite im Admin Interface ist für berechtige Benutzer jeder verbundene Röster ersichtlich.

8.2.5. UC05 Röster ausliefern

Dieser Use Case wurde so bearbeitet, dass für den echten Röster alles vorbereitet ist.

8.2.6. UC06 Daten hochladen

Dieser Use Case wurde implementiert. Der Röster erkennt neue erhaltene Daten und übermittelt diese an den Server.

8.2.7. Fazit

Rückblickend fehlt ein grosser Use Case, der als Grundlage für die Bearbeitung der anderen dient. Die Kommunikation zwischen dem Röster und AWS herstellen. Dies war der wichtigste Teil des ganzen Projektes, doch er wurde nie direkt in den Use Cases aufgegriffen. So hat es für den Kunden teilweise ausgesehen, als würden wir nicht vom Fleck kommen und plötzlich waren alle Use Cases innert vier Wochen abgeschlossen. Doch viel Zeit wurde in diese Kommunikation investiert und sobald diese erfolgreich funktionierte, konnten viele Use Cases schnell abgeschlossen werden. Durch einen Unvorhersehbares Ereignis musste der Röster simuliert werden, was im Nachhinein auch einen Use Case verdient hätte. Das Team hätte mutiger sein können, nach der Elaboration Phase die Anforderungen trotzdem noch leicht abzuändern und zu erweitern.

9. Schlussfolgerung

9.1. Fazit

In der Arbeit wurde erfolgreich eine Kommunikation zwischen simuliertem Röster, Admin Interface und App umgesetzt. Dabei verwenden die App, sowie das Admin Interface eine TypeScript Library, welche die verschiedenen Funktionen den beiden Endpunkten zur Verfügung stellt. Die Kommunikation läuft über ein selbst definiertes Protokoll, welches auf MQTT basiert. Als MQTT Broker agiert der AWS IoT Core von Amazon. Über das Admin Interface können alle Röster angezeigt werden und in einer Detail Ansicht werden die Sensordaten jedes Rösters ersichtlich. Die Library stellt verschiedenen Funktionalitäten zur Verfügung. Es können Subscribers auf einen Sensor des Rösters erstellt, Befehle versendet und alle Röster abgerufen werden. Das Eval Board, welches an die Röster angehängt werden soll, um die Kommunikation zu ermöglichen, konnte nicht aufgesetzt werden. Dafür wurde eine State Maschine geschrieben, welche einen Röster simuliert. Das Problem mit dem Eval Board war das fehlende Wissen im Bereich der Elektrotechnik. Da das Eval Board nicht richtig eingerichtet werden konnte, wurde das Einrichten des Rösters (UC 01) nicht erledigt. Es wurden zwar Konzepte grundlegend analysiert, diese aber nicht weiter ausgearbeitet. Das liegt daran, dass das Arbeiten mit Bluetooth oder einem WAP deutliche Unterschiede auf den verschiedenen Systemen hat und daher kein Mehrwert gewonnen werden konnte. Ebenfalls wurde die App nicht umgeschrieben, da dies von der Zeit her nicht mehr möglich war. Dies liegt vor allem daran, dass die App derzeit stark auf der USB-Verbindung basiert. Die Zeit wurde bereits knapp, da das Team dies bereits vorhergesehen hat wurde sich bereits im Voraus dazu entschieden eine TypeScript Library zu erstellen. Die App sowie das Admin Interface verwenden TypeScript, daher konnte eine Library implementiert werden welche leicht in die App eingebunden werden kann und die Funktionalitäten trotzdem getestet werden konnten, ohne dass die App verändert wurde.

9.2. Nächste Schritte

Das Ergebnis dieser Arbeit muss noch erweitert werden, bevor es in den Praxiseinsatz aufgenommen werden kann.

Der grösste und wichtigste Punkt wäre das Konfigurieren des Eval Board. Da die Röster Simulation mit einer Hochsprache geschrieben wurde, kann der Code nicht eins zu eins übernommen werden. Es muss mit der AWS SDK für die auf dem Eval Board verwendete Sprache (voraussichtlich C/C++) gearbeitet werden. Ebenfalls muss eine Kommunikation zwischen Eval Board und dem Herz des Rösters ausgearbeitet werden, damit die Befehle zum Röster kommen und die Sensordaten zum Eval Board. Dies beeinflusst die anderen Teile wie Library und Protokoll aber nicht.

Ein weiterer Punkt wäre die Einrichtung des Rösters. Über die App soll ein Benutzer die Möglichkeit haben ein WLAN für seinen Röster zu selektieren und sich in dieses einzuloggen. Damit kann sich der Röster beim Kunden in das korrekte WLAN einloggen und dadurch eine Verbindung zum Internet und zu AWS aufbauen. Wird dabei mit dem EvalBoard (STMicroelectronics B-L475E-IOT01A) gearbeitet, sollte die Warnung der Hersteller beachtet werden, dass es zu Problemen kommen kann, wenn Bluetooth und WLAN Verbindungen gleichzeitig aktiv sind.

Sollte das Start-Up neue Sensoren in den Röster verbauen, können sie das Protokoll einfach um neue Topics erweitern. Auf dem Röster muss dieser Wert einfach publiziert werden und in der Library kann das Topic abonniert werden.

Bei neuen Befehlen muss das Protokoll erweitert werden. Bei Befehlen soll weiterhin eine Payload mitgesendet werden. Diese Payload beinhaltet mindestens eine CommandId, damit weiterhin mit dem Accepted und Rejected Topic gearbeitet werden kann.

Glossar

AWS Amazon Web Services. 16

ESLint Ein Code Analyse Tool welches problematische JavaScript Code Teile erkennt. Mit Plugins können auch TypeScript und TSX analysiert werden. 15

MQTT MQTT ist ein Protokoll welches als Standard für IoT Nachrichten gilt. 16

Prettier Ein Tool welches dabei hilft die Code Formatierung beizubehalten. 15

Shadow Ein Objekt welches den Device Status verfügbar stellt egal ob das Gerät verbunden ist oder nicht. 20

TSX Die TypeScript Variante von JSX. 15

WAP Wireless Access Point. 8

Literatur

[1] *AWS IoT Core endpoints and quotas*. URL: https://docs.aws.amazon.com/general/latest/gr/iot-core.html.

[2] *AWS IoT Device Shadow service*. URL: https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html (besucht am 23. 10. 2020).

[3] *AWS IoT security*. URL: https://docs.aws.amazon.com/iot/latest/developerguide/iot-security.html.

[4] *B-L475E-IOT01A: STM32L4 Discovery kit IoT node, low-power wireless, BLE, NFC, SubGHz, Wi-Fi*. URL: https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html.

[5] *ClickUp*. URL: https://clickup.com/ (besucht am 18. 10. 2020).

[6] Kent C. Dodds. *Implementing a simple state machine library in JavaScript*. 20. Jan. 2020. URL: https://kentcdodds.com/blog/implementing-a-simple-state-machine-library-in-javascript.

[7] Peter Eeles. *Capturing Architectural Requirements*. 15. Nov. 2005. URL: https://www.ibm.com/developerworks/rational/library/4706.html (besucht am 26. 09. 2020).

[8] *GitHub*. URL: https://github.com/ (besucht am 18. 10. 2020).

[9] *GitLab*. URL: https://gitlab.com/ (besucht am 18. 10. 2020).

[10] *Google Tabellen*. URL: https://www.google.com/sheets/about/ (besucht am 18. 10. 2020).

[11] Bryan Grill. *Create-React-App with TypeScript, ESLint, Prettier, and Github Actions*. URL: https://medium.com/@brygrill/create-react-app-with-typescript-eslint-prettier-and-github-actions-f3ce6a571c97.

[12] *LaTeX*. URL: https://www.latex-project.org/ (besucht am 18. 10. 2020).

[13] *Microsoft Teams*. URL: https://www.microsoft.com/en-us/microsoft-365/microsoft-teams/group-chat-software (besucht am 18. 10. 2020).

[14] *MQTT Essentials*. URL: https://www.hivemq.com/mqtt-essentials/ (besucht am 15. 10. 2020).

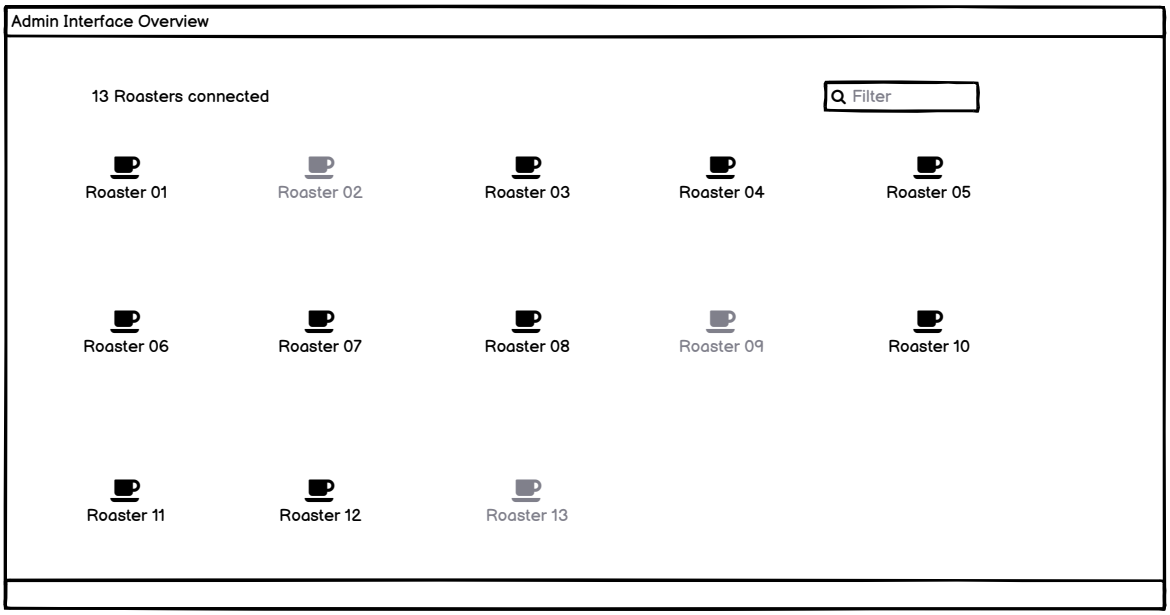
[15] *PEP8*. URL: https://pep8.org (besucht am 01. 01. 2020).

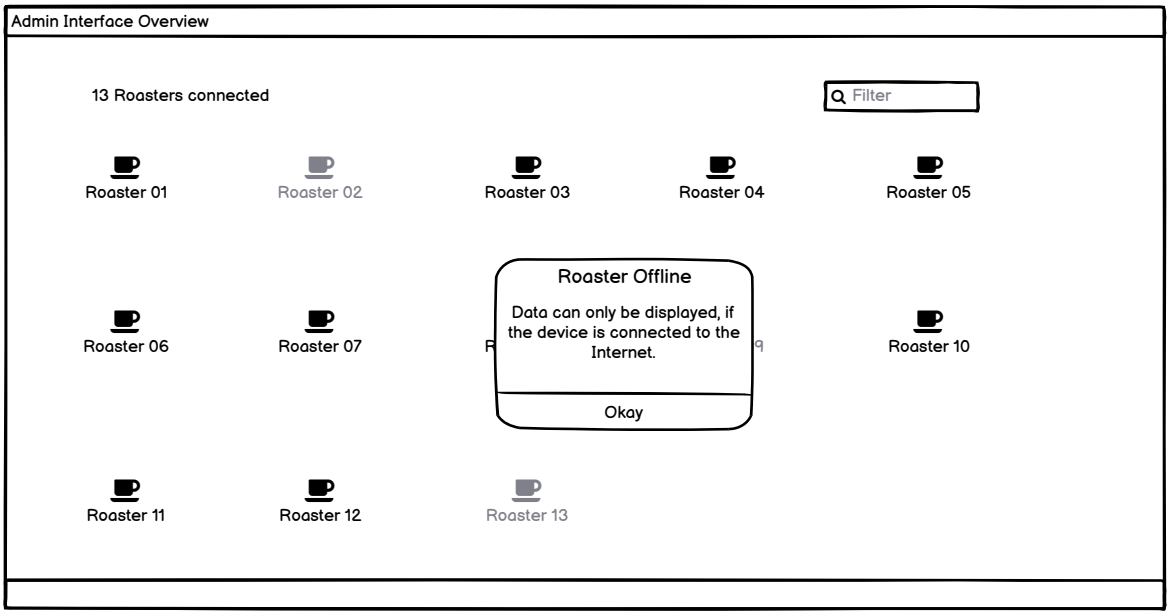
[16] *Raspberry Pi 4 Model B*. URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/.

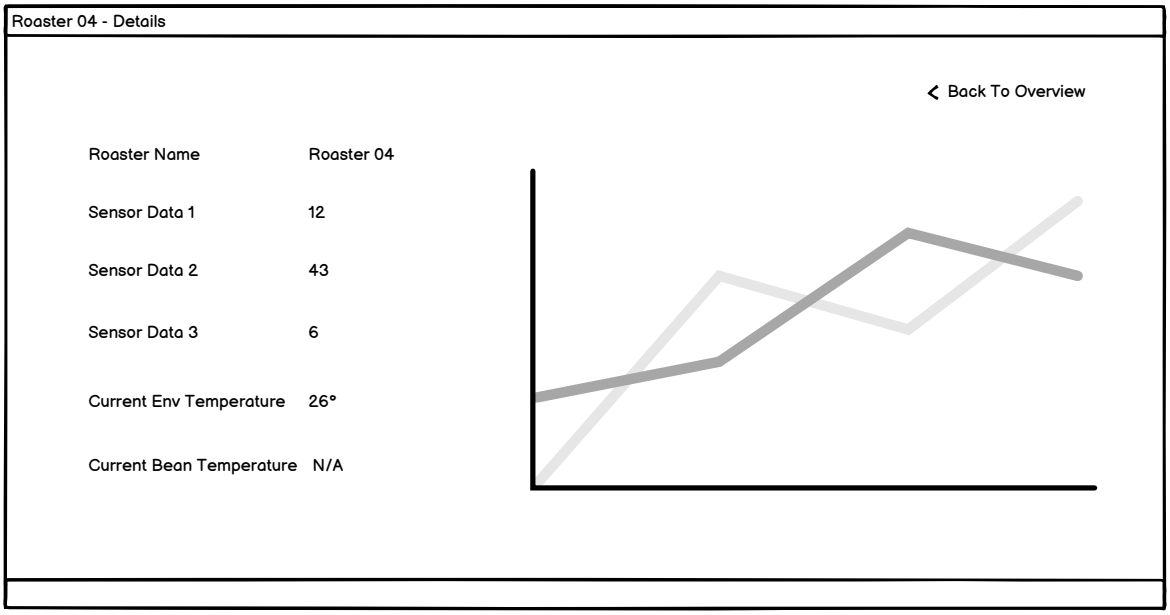
[17] *Slack*. URL: https://slack.com/ (besucht am 18. 10. 2020).

[18] *STM32CubeIDE: The First Free ST IDE with STM32CubeMX Built-in*. 28. Mai 2019. URL: https://blog.st.com/stm32cubeide-free-ide/.

A. Mockups


Admin Interface Overview
13 Roasters connected
Filter
Roaster 01
Roaster 02
Roaster 03
Roaster 04
Roaster 05
Roaster 06
Roaster 07
Roaster 08
Roaster 09
Roaster 10
Roaster 11
Roaster 12
Roaster 13


Admin Interface Overview
13 Roasters connected
Filter
Roaster 01
Roaster 02
Roaster 03
Roaster 04
Roaster 05
Roaster Offline
Data can only be displayed, if the device is connected to the Internet.
Okay
Roaster 06
Roaster 07
Roaster 10
Roaster 11
Roaster 12
Roaster 13


Roaster 04 - Details
< Back To Overview
Roaster Name
Roaster 04
Sensor Data 1
12
Sensor Data 2
43
Sensor Data 3
6
Current Env Temperature
26°
Current Bean Temperature
N/A