SPRINT ABSCHLUSSBERICHT

Sprintdokumentation
06.03.2019 – 08.05.2019

Telemetry Gateway Monitoring - IBM CIC

Maciej Dzialoszynski, Sebastian Grünewald, David Jovanovic, Jordi Rieder

Inhaltsverzeichnis

Änderungsverzeichnis

Version	Autor	QS	Datum	Status
1.0	David Jovanovic	Sebastian Grünewald	29.03.2019	erledigt
2.0	Maciej Dzialoszynski	David Jovanovic	08.05.2019	erledigt

1 Sprintbericht

Wir haben unser Projekt in Userstories eingeteilt, welche aus mehreren Sub-Items bestehen. Die Userstories wurden den einzelnen Sprints zugewiesen.

1.1 4390 IBM Cloud

Hier werden grundlegende Konfigurationen auf der IBM-Cloud durchgeführt, welche notwendig sind, um das Projekt zu realisieren. Mit dem Auftraggeber wurde bereits eine Softwarearchitektur erarbeitet, welche es umzusetzen gilt. Weiters werden die Verknüpfungen zwischen den Microservices hergestellt.

1.2 4391 Frontend – Wireframes

Es werden Wireframes erstellt, welche die GUI repräsentieren. Die intuitive Bedienung wird angestrebt.

1.3 4392 Backend – Setup

Hier werden grundlegende Strukturen des Backends aufgebaut. Es müssen alle notwendigen Module installiert, verknüpft und lauffähig gemacht werden. Weiters soll es durch diese Userstory, den Entwicklern ermöglicht werden, direkt mit der Implementierung zu beginnen.

1.4 4393 Backend – GW – Cloud

Es wird möglich sein Daten vom Gateway auf die Cloud zu übertragen, um damit in weiteren Schritten arbeiten zu können.

1.5 4413 Frontend – Setup

Hier werden grundlegende Strukturen des Frontends aufgebaut. Es müssen alle notwendigen Module installiert, verknüpft und lauffähig gemacht werden. Weiters soll es durch diese Userstory, den Entwicklern ermöglicht werden, direkt mit der Implementierung zu beginnen.

ID	Name	Zugewiesen	Schätzung	Status	Akzeptanzkriterium
4390	**IBM Cloud**		**3.5 hrs**	**In Progress**	
4394	Anlegen einer IoT-Plattform	Maciej Dzialoszynski	0.5 hrs	**Completed**	Die IoT-Plattform ist über die IBM-Cloud-Accounts der Teammitglieder zugänglich.
4395	Einrichten einer Frontend-Cloud-Foundary-App	Maciej Dzialoszynski	0.5 hrs	**Completed**	Die Sample-App ist lauffähig.
4396	Einrichten einer Backend-Cloud-Foundary-App	David Jovanovic	0.5 hrs	**Completed**	Die Sample-App ist lauffähig.
4397	Anlegen von Git-Repository für Frontend	Maciej Dzialoszynski	0.5 hrs	**Completed**	Git-Repository ist klon- und veränderbar.
4398	Anlegen von Git-Repository für Backend	Jordi Rieder	0.5 hrs	**Completed**	Git-Repository ist klon- und veränderbar.
4399	Kommunikation zwischen Backend und Datenbank (Cloudant)	Jordi Rieder	0.5 hrs	**Open**	Verbindung ist hergestellt

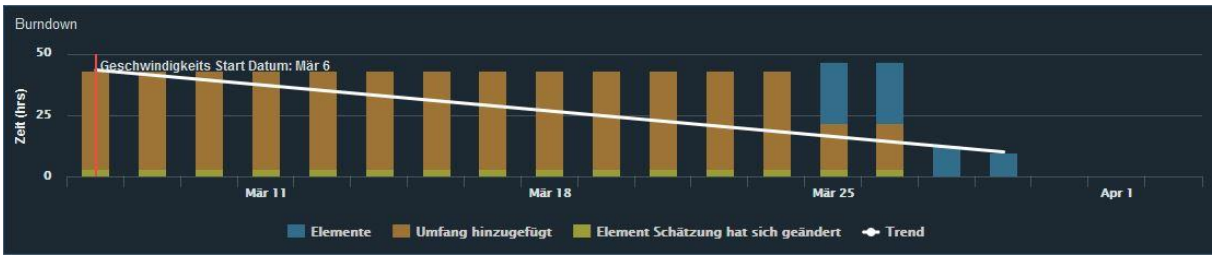
4400	Kommunikation zwischen Backend und Frontend	Jordi Rieder	0.5 hrs	**Open**	Verbindung ist hergestellt
4391	**Frontend - Wireframes**		**10 hrs**	**Completed**	
4405	Login/Registrierung	Maciej Dzialoszynski	1 hrs	**Completed**	Das Wireframe wurde von Papier auf das Mockup-Tool Balsamiq übertragen.
4406	Gerät-Registrierung	Sebastian Grünewald	3 hrs	**Completed**	Das Wireframe wurde von Papier auf das Mockup-Tool Balsamiq übertragen.
4407	Gerät-Liste	Sebastian Grünewald	3 hrs	**Completed**	Das Wireframe wurde von Papier auf das Mockup-Tool Balsamiq übertragen.

4408	Gerät-Details	Sebastian Grünewald	3 hrs	**Completed**	Das Wireframe wurde von Papier auf das Mockup-Tool Balsamiq übertragen
4392	**Backend - Setup**		**3 hrs**	**In Progress**	
4409	Ordnerstruktur	David Jovanovic	1 hrs	**Completed**	Jedes Modul und jede zusätzliche Technologie hat einen eigenen Ordner.
4410	Node-Arbeitsumgebung aufsetzen	David Jovanovic	2 hrs	**Completed**	Die Mitglieder haben Zugriff auf die Node.js App auf der Cloud und können ihre Änderungen pushen. Die neuste Version sollte dann immer verfügbar sein über den Link auf der Cloud
4393	**Backend - GW-Cloud**		**10 hrs**	**In Progress**	

4411	GW-Datenempfang	David Jovanovic	5 hrs	**Open**	Die korrekten Daten werden empfangen.
4412	Datenfilterung	Jordi Rieder	5 hrs	**In Progress**	Empfang der gewünschten Daten
4413	**Frontend - Setup**		**7 hrs**	**In Progress**	
4414	Ordnerstruktur	Sebastian Grünewald	1 hrs	**Completed**	Jede Komponente und jede zusätzliche Technologie hat einen eigenen Ordner.
4415	React-App mit der Node-App verbinden	Sebastian Grünewald	1 hrs	**Completed**	Datenaustausch zwischen jenen Apps möglich.
4416	Routen festlegen	Maciej Dzialoszynski	4 hrs	**In Progress**	Navigation zwischen Login/Reg-Seite und Hauptseite funktioniert und die Navigation zwischen einzelnen Komponenten funktioniert.

4417	Sass aufsetzen	Maciej Dzialoszynski	1 hrs	**Completed**	Sass in Projekt eingebunden und funktionstüchtig.

1.6 Sprint-Burndown-Chart



2 Sprintbericht 03.04 – 20.04

2.1 Message Handler – PSP: 2,1

Message Handler

Beschreibung

Die Aufgabe des Message Handler ist es die von den Gateways gesendeten Daten aus der Watson IOT-Plattform auszulesen und an den DB-Handler zu übergeben. Hierbei handelt es sich um eine Nodejs-Application welche mit dem Modul „ibmiotf“ arbeitet.

Akzeptanz

Der Eventhandler gilt dann als korrekt und fertig implementiert, wenn zuverlässig und ohne Ausfälle alle Gateway-Daten auf der IOT-Plattform abgelesen werden können und an den DB-Handler übergeben werden. Ein abschließendes Kriterium ist das Deployment in der IBM-Cloud.

Durchführung

Bei der Implementierung muss zunächst das Modul „ibmiotf“ über die CLI mit `npm i –save ibmiotf` heruntergeladen werden. Anschließend muss das Modul in der Node-App geladen werden.
Anschließend muss eine die Konfiguration definiert werden, in welcher sich die Daten zum Anmelden an der IOT-Plattform befinden.

```
const appClientConfig = {
  'org' : 'xwjcou',
  'id' : 'gw1',
  'domain': 'internetofthings.ibmcloud.com',
  'type': 'Multitech',
  'auth-key' : 'secret',
```

```
  'auth-token' : 'secret'
}
```

Dann wird mit der Config ein Client erstellt und mit der Plattform verbunden:

```
const appClient = new Client.IotfApplication(appClientConfig);
appClient.connect();
```

Da wir alle Daten des Gerätetyps Multitech empfangen wollen, wird eine Funktion implementiert welche bei Verbindung zur IOT-Plattform alle Events der Multitech Geräte abonniert und empfängt.

```
appClient.on('connect', function () {
  //Subscribing to all Device Events of type Multitech
  appClient.subscribeToDeviceEvents('Multitech');
});
```

Als letztes muss noch eine Funktion implementiert werden, welche die empfangenen Daten an den DB-Handler weiterleitet. Die Weiterleitung erfolgt über die im DB-Handler implementierte Funktion *saveToDb.*

```
appClient.on('deviceEvent', function (deviceType, deviceId, eventType,
format, payload) {
    //If the received event is of type data, the received data gets
persisted in the database
    if (eventType === 'data') {
        database.saveToDb(JSON.parse(payload));
    }
});
```

Dokumentation

1. https://github.com/ibm-watson-iot/iot-nodejs

2.2 DB Handler – PSP: 2,2

DB Handler Cloudant

Beschreibung

Die Aufgabe des „DB-Handler Cloudant“ ist es die vom Message-Handler empfangenen Daten in der Cloudant-Datenbank. In der Cloudant-DB, bei welcher es sich um eine NoSQL Datenbank handelt, sollen jeweils nur die aktuellsten Datensätze abgespeichert werden. Bei diesem Arbeitspacket handelt es sich um eine Nodejs-Application welche mit dem Modul „@cloudant/cloudant“ arbeitet.

Akzeptanz

Der „DB-Handler Cloudant“ gilt dann als korrekt und fertig implementiert, wenn zuverlässig und ohne Ausfälle alle vom Message-Handler Gateway-Daten in der Datenbank abgespeichert werden, hierbei sollen jedoch die alten Datensätze immer überschrieben

werden sodass immer nur die aktuellsten Daten vom jeweiligen Gateway zu Verfügung stehen. Ein abschließendes Kriterium ist das Deployment in der IBM-Cloud.

Durchführung

Bei der Implementierung muss zunächst das Modul „ibmiotf“ über die CLI mit `npm i --save @cloudant/cloudant` heruntergeladen werden. Anschließend muss das Modul in der Node-App geladen werden.
Anschließend muss eine die Konfiguration definiert werden, in welcher sich die Daten zum Anmelden an der Cloudant-Plattform befinden.

```
var Cloudant = require('@cloudant/cloudant');
const cloudant = new Cloudant({
  account: '1f81396c-51b2-466d-b231-84c50a16cdb4-bluemix',
  plugins: {
    iamauth: {
      iamApiKey: 'secret'
    }
  }
});
```

Danach wird noch die entsprechende Datenbank ausgewählt:

```
const dbiot = cloudant.db.use('tgm-iot');
```

In der Funktion saveToDb wird die Id des Devices auch zur Id der Cloudant-Datenbank gesetzt. Anschließend wird die die Datanbank durchsucht ob es bereits einen Eintrag zu diesem GW gibt, da die Daten überschreiben werden sollen. Hierfür wird die _rev Nummer benötigt. Abschließend werden die Daten in die DB gespeichert.

```
var deviceId = data.deviceId;
data._id = deviceId;

var rev = "";
dbiot.find({selector:{ "_id":deviceId }}, function(err, result) {
  if (err) return console.log(err.message);
  data._rev = result.docs[0]._rev;
  rev = result.docs[0]._rev;
  dbiot.insert(data, function (err, result) {
    if (err) {
      throw err;
    }
  });
});
```

Dokumentation

1. https://github.com/cloudant/nodejs-cloudant

DB Handler DB2-Warehouse

Beschreibung

Die Aufgabe des „DB-Handler DB2-Warehouse“ ist es die vom Message-Handler empfangenen Daten in dem Db2-Warehouse. In dem Db2-Warehouse, bei welcher es sich um eine SQL Datenbank mir sehr schnellen Zugriffzeiten handelt, sollen jeweils alle Datensätze dauerhaft persistiert werden Bei diesem Arbeitspacket handelt es sich um eine Nodejs-Application welche mit dem Modul „ibm_db“ arbeitet.

Akzeptanz

Durchführung

Bei der Implementierung muss zunächst das Modul „ibmiotf“ über die CLI mit `npm i --save ibm_db` heruntergeladen werden. Anschließend muss das Modul in der Node-App geladen werden.
Anschließend muss noch die Verbindung zur Datenbank mit entsprechendem Connection-String.

```
const ibmdb = require('ibm_db');
ibmdb.open("DATABASE=BLUDB;HOSTNAME=dashdb-txn-flex-yp-fra02-
240.services.eu-
de.bluemix.net;UID=bluadmin;PWD=secret;PORT=50000;PROTOCOL=TCPIP",
function (err,conn) {
```

Es wird eine SQL-Query benötigt welche die empfangen Daten in das DB2-Warehouse abspeichert:

```
var statement = "INSERT INTO gateway (latestCheckin, deviceId,
macAddress, ip, uptime, signal) VALUES ('"+data.latestCheckin+"',
"+data.deviceId+", '"+data.macAddress+"', '"+data.ip+"',
"+data.uptime+", "+data.signal+");";
conn.query(statement, function (err, data) {
   if (err) console.log(err);
   else console.log(data);
```

Abschließend muss die Verbindung zur Datenbank geschlossen werden:

```
conn.close(function () {
  console.log('done');
});
```

Dokumentation

1. https://github.com/ibmdb/node-ibm_db/

2.3 REST API – PSP: 2,6

Gateway - Cloud

Beschreibung

Im zweiten Sprint ist zu dieser Hauptuserstory ein Arbeitspaket dazugekommen. Wir haben uns als Aufgabe definiert, eine NodeRed-Applikation zu entwickeln, welche auf die REST-Schnittstelle vom Gateway zugreift, diese in ein JSON formatiert und anschließend zum MQTT-Broker der Watson-IoT-Plattform schickt.

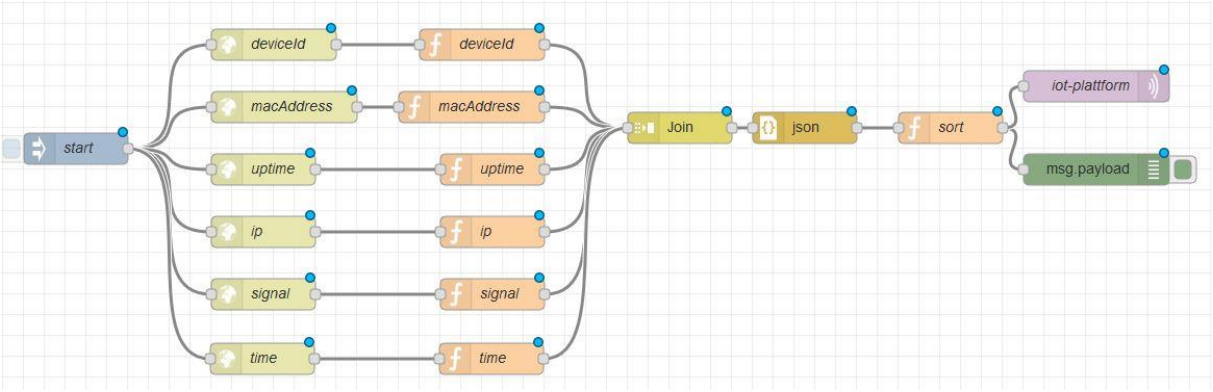
Akzeptanz

Das Arbeitspaket ist erledigt, wenn die korrekten Daten als JSON zur Cloud geschickt werden und dort empfangen werden. Dazu gehören folgende Attribute:

- Timestamp (Wann wurde die Nachricht verschickt?)
- IP
- MAC
- DeviceID
- Signal-strength
- Up-Time (Wie lange läuft das GW schon?)

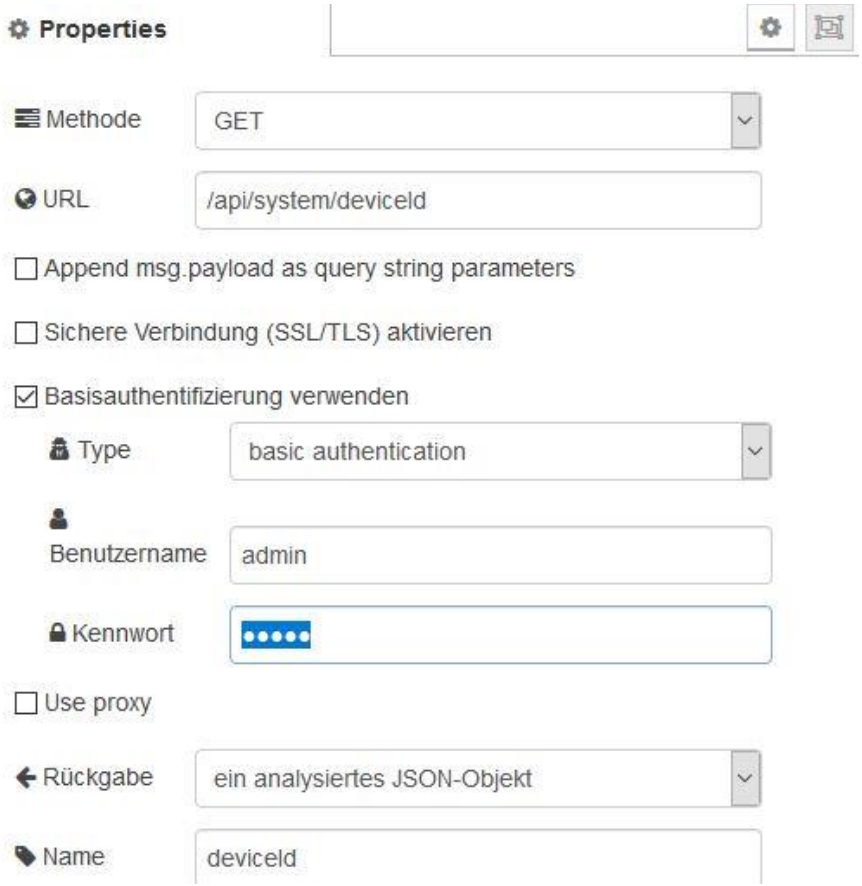
Durchführung

Node-Red ermöglich es, in Node.js zu entwicklen mittels Flows. Unser Flow sieht folgendermaßen aus:



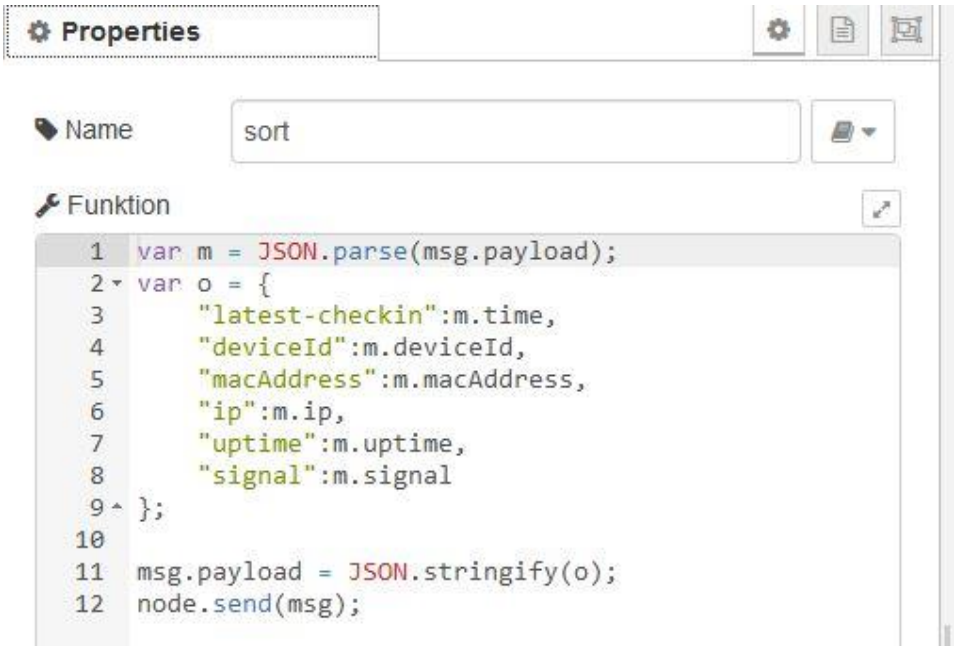
Das Erste Element mit dem Namen **start** ist der Impulsgeber, das bedeutet in unserem Fall, in welchem Intervall der Flow ausgeführt wird, das ist abhängig von der Situation, aber wir haben es im Moment auf alle zehn Sekunden eingestellt.

Die Zweite Spalte, oder auch die gelben Elemente sind jeweils GET-Requests an die API des Gateways. Hier sieht man ein Beispiel von der deviceId:

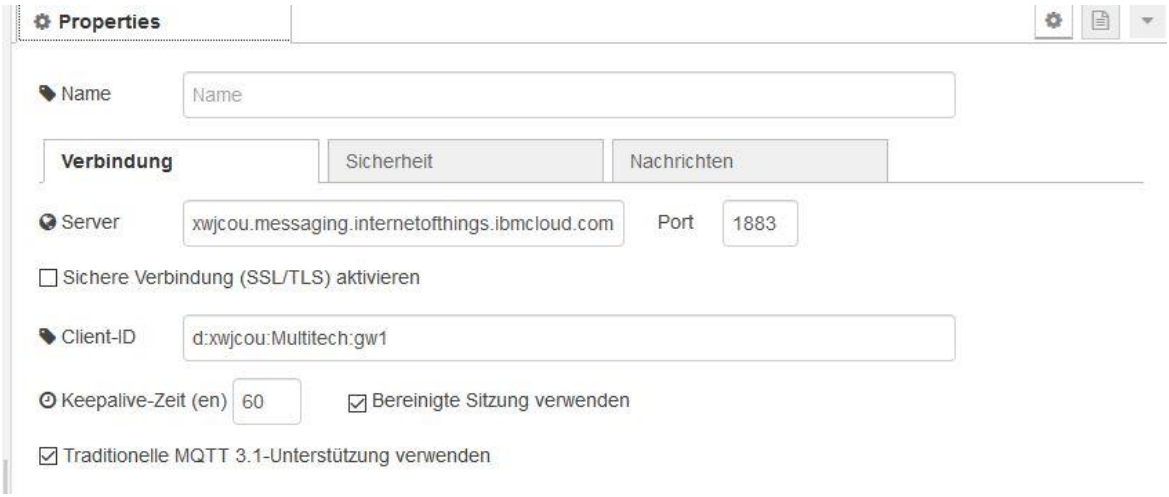


Die nächste Spalte besteht aus Funktionen. Da werden die eingehenden Infos aufbereitet und anschließend weitergeleitet:

Da ein JSON empfangen wird von der GW-API, wird daher nur der konkrete Wert rausgefiltert, darum **.result** in der Ersten Zeile, danach wird das msg Objekt bearbeitet und weitergeleitet.
Danach kommt ein Join, welches alle msg-Objekte zu einem fügt und es weiterschickt an eine konkrete Funktion, welches das JSON aufbereitet für den MQTT-Broker:



Anschließend werden die Konfigurationen im MQTT-Node eingetragen und die Cloud wird mit Daten versorgt.



Dokumentation

1. https://nodered.org/
2. https://github.com/node-red/node-red

Cloudant - REST-API

Beschreibung

Es soll eine REST-API erstellt werden, welche alle Einträge, aus der Cloudant-Microservice(NoSQL-Datenbank), bereitstellt als JSON.

Akzeptanz

Es können GET-Requests geschickt werden, durch die man die Daten bekommt, in unserem Fall das Frontend für die Darstellung.

Durchführung

```
//Imports
const express = require('express');
const app = express();
var Cloudant = require('@cloudant/cloudant');
var cors = require('cors')
// Objekt für das deployen auf am
// Cloud Foundry application environment
var cfenv = require('cfenv');
var appEnv = cfenv.getAppEnv();

// Zugang zum Cloudant
const cloudant = new Cloudant({
        account: '1f81396c-51b2-466d-b231-84c50a16cdb4-bluemix',
        plugins: {
                iamauth: {
                        iamApiKey: 'LlhLoyUW_bQI7GF5ms32miAIZln_rNgcLK2V8YXs7JPx'
                }
        }
});

const db = cloudant.db.use('tgm-iot');


//Holen der Daten + bereitstellung auf /gw1
app.get('/gw1', cors(), (req, res, next) => {
        db.get("19650069").then((data) => {
                console.log(data);
                res.send(data);
        });
});

// Den Server am angegebenen Port und IP starten
// Ausgabe wenn es läuft
app.listen(appEnv.port, '0.0.0.0', function () {
        // print a message when the server starts listening
        console.log("Listening");
        console.log("Url:" + appEnv.url);
        console.log("Port:" + appEnv.port);
});
```

Dokumentation

1. https://www.npmjs.com/package/cfenv
2. https://github.com/cloudant/nodejs-cloudant
3. https://expressjs.com/de/guide/routing.html

2.4 Frontend – Setup – PSP: 3,2

Continous Delivery - Build

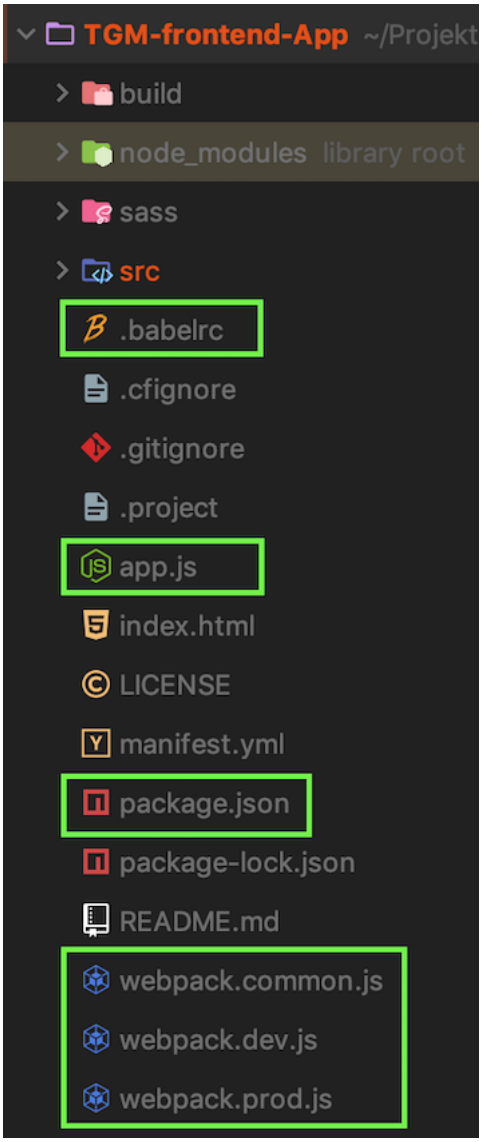
Beschreibung

Erste Phase der Continous Delivery Pipeline. Der "Build"- Prozess der App muss mit der Cloud abgestimmt werden.

Akzeptanz

Build Stage auf IBM Cloud auf "Phase bestanden".

Durchführung



Wichtig für einen fehlerfreien Build-Prozess der Frontend-Applikation sind die, grün markierten, Dateien. React.js benötigt einen Übersetzer. In unserem Fall verwenden wir **Babel**. In die **.babelrc**-Datei werden folgende „presets“ geschrieben:

```
"@babel/preset-env",
"@babel/preset-react“
```

Hiermit wird Babel für die Zusammenarbeit mit React.js vorbereitet. Um die Größe unserer Applikation minimal zu halten, verwenden wir einen Bundler, **Webpack**. Diese wurde in 3 Dateien aufgespalten. Eine Haupt- eine Dev- und eine Produktions (Deploy) – Datei. Wichtig hierbei ist folgendes.

In der Hauptdatei muss sich folgender Inhalt befinden.

```
entry: './src/index.js',
output: {
   path: path.resolve(__dirname, 'build'),
   publicPath: '/',
   filename: 'bundle.js'
},
resolve: {
   modules: [__dirname, 'src/components', 'node_modules'],
   extensions: ['*', '.js', '.jsx'],
},
```

Somit wird zum einen sichergestellt, dass Webpack die Dateiendung **.jsx**, mit welcher React arbeitet, annimmt und zum anderen wird die Quell-Datei (in diesem Fall **index.js**), die Output-Datei (in diesem Fall **bundle.js**) und der Output-Pfad (in diesem Fall **build**) gesetzt.

Vom genannten Pfad aus, wird letztendlich unsere Applikation aufgerufen und aktiviert. Damit dies ermöglicht werden kann, muss die Datei **app.js** folgende Zeile enthalten:

```
app.use(express.static(__dirname + '/build'));
```

Außerdem müssen in der **package.json** zusammenspielende Tool-Versionen zur Verwendung bereitgestellt sein.

Auf Seiten der IBM-Cloud wird die **Delivery Pipeline** unserer Frontend-Applikation aufgesucht und bei **Build Stage** auf „Phase konfigurieren“ gedrückt. Unter dem Abschnitt „Jobs“ wird „npm“ als **Buildertyp** gewählt und folgendes ins **Build-Script** hinzugefügt:

```
npm install
npm run build
```

Nun wird bei einem „git push“ auf der IBM Cloud korrekt „gebuildet“. Möchte man lokal „builden“, so führt man zweiteren, der Build-Script Befehle aus.

Dokumentation

1. https://babeljs.io
2. https://webpack.js.org
3. https://nodejs.org/en/

Continous Delivery - Deploy

Beschreibung

Zweite Phase der Continous Delivery Pipeline. Der "Deploy"- Prozess der App muss mit der Cloud abgestimmt werden.

Akzeptanz

Deploy Stage auf IBM Cloud auf "Phase bestanden".

Durchführung

Bei korrekt implementiertem Build-Prozess, welcher im Arbeitspaket mit dem Namen **Continous Delivery – Build** beschrieben wird, sollte nach einem „git push“ dem Deployment nichts im Wege stehen. Im Hintergrund wird die Applikation auf die Cloud übertragen. Die Dateien werden aus dem **build**-Ordner entnommen.

Webpack: File-Loader

Beschreibung

Um den require() Befehl in React anwenden zu können, muss ein File-Loader installiert werden. Der File-Loader löst den **import/require()** Befehl auf eine Datei in eine URL auf und gibt die Datei in das Ausgabeverzeichnis zurück.

Akzeptanz

`npm run build` erfolgreich

Durchführung

Installation

```
$ npm install file-loader --save-dev
```

Dokumentation

1. https://github.com/webpack-contrib/file-loader

Webpack: Image-Loader

Beschreibung

Um Bilder aller Arten (png, jpg, jpeg, svg, ...) in die React-App einbinden zu können, muss in Webpack ein Image Loader eingebaut werden.

Akzeptanz

File-Loader vorhanden + `npm run build` erfolgreich + Bilder werden angezeigt

Durchführung

Installation

```
$ npm install image-webpack-loader --save-dev
```

webpack.common.js

```
{
    test: /\.(gif|png|jpe?g|svg)$/i,
        use: [
            'file-loader',
            {
                loader: 'image-webpack-loader',
                options: {
                    bypassOnDebug: true, // webpack@1.x
                    disable: true, // webpack@2.x and newer
                },
            },
        ],
}
```

Dokumentation

1. https://www.npmjs.com/package/image-webpack-loader

Webpack: SASS/CSS-Loader

Beschreibung

Damit SASS-Files und dessen untergeordneten CSS-Files gültig von der React-App geladen werden, muss in Webpack ein geeigneter Loader eingebaut werden.

Akzeptanz

`npm run build` erfolgreich + Style-Änderungen werden übernommen

Durchführung

Installation

```
$ npm install sass-loader node-sass webpack --save-dev
$ npm install style-loader css-loader --save-dev
```

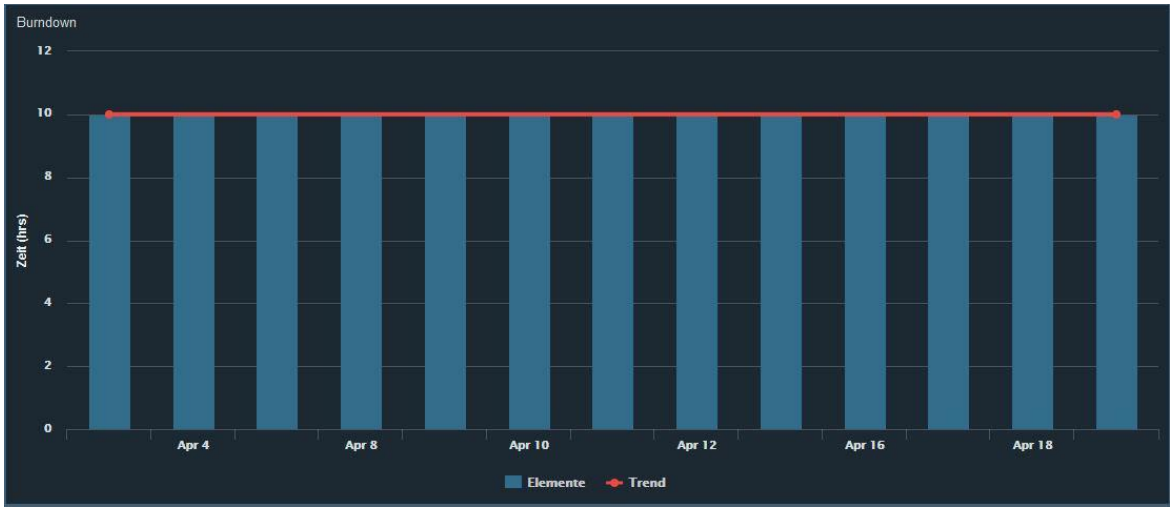
webpack.common.js

```
{
    test: /\.scss$/,
        use: [
            "style-loader", // creates style nodes from JS strings
            "css-loader", // translates CSS into CommonJS
            "sass-loader" // compiles Sass to CSS, using Node Sass by default
        ]
}
```

Dokumentation

1. https://github.com/webpack-contrib/sass-loader

Sprint-Burndown-Chart



ID	Name	Zugewiesen	Akzeptanz	Schätzunghätzung	Status
4424	Frontend - Setup - PSP: 3,2	IoT Plattform IBM		4.5 hrs	Closed
4425	Continous Delivery - Build	Maciej Dzialoszynski	Build Stage auf IBM Cloud auf "Phase bestanden".	0.5 hrs	Closed
4427	Continous Delivery - Deploy	Maciej Dzialoszynski	Deploy Stage auf IBM Cloud auf "Phase bestanden".	0.5 hrs	Closed
4428	ReactJS und Node-Server verknüpfen	Sebastian Grünewald	Verknüpfte Anwendung kann in die Delivery-Pipeline eingebunden werden	2 hrs	Closed
4429	Webpack - File Loader	Maciej Dzialoszynski	npm run build erfolgreich	0.5 hrs	Closed
4430	Webpack - Image Loader	Sebastian Grünewald	File Loader vorhanden + npm run build erfolgreich + Bilder werden angezeigt	0.5 hrs	Closed
4431	Webpack - SASS/CSS Loader	Sebastian Grünewald	npm run build erfolgreich +Style-Änderungen werden übernommen	0.5 hrs	Closed
4432	REST-API	David Jovanovic	Es können GET-Requests geschickt werden, durch die man die Daten bekommt, in unserem Fall das Frontend für die Darstellung	15 hrs	Closed
4433	NodeRed-App	David Jovanovic	Es müssen die korrekten Daten vom Gateway als JSON auf der Cloud ankommen	10 hrs	Closed

4434	API	David Jovanovic	Es können GET-Requests geschickt werden, durch die man die Daten bekommt, in unserem Fall das Frontend für die Darstellung	5 hrs	Closed
4435	Message-Handler	Jordi Rieder	Der Eventhandler gilt dann als korrekt und fertig implementiert, wenn zuverlässig und ohne Ausfälle alle Gateway-Daten auf der IOT-Plattform abgelesen werden können und an den DB-Handler übergeben werden. Ein abschließendes Kriterium ist das Deployment in der IBM-Cloud.	20 hrs	Closed
4436	DB-Handler Cloudant	Jordi Rieder	Der „DB-Handler Cloudant“ gilt dann als korrekt und fertig implementiert, wenn zuverlässig und ohne Ausfälle alle vom Message-Handler Gateway-Daten in der Datenbank abgespeichert	20 hrs	Closed

			werden, hierbei sollen jedoch die alten Datensätze immer überschrieben werden sodass immer nur die aktuellsten Daten vom jeweiligen Gateway zu Verfügung stehen. Ein abschließendes Kriterium ist das Deployment in der IBM-Cloud.		
4437	DB-Handler Db2-Warehouse	Jordi Rieder	Der „DB-Handler Db2-Warehouse“ gilt dann als korrekt und fertig implementiert, wenn zuverlässig und ohne Ausfälle alle vom Message-Handler Gateway-Daten in der Datenbank abgespeichert werden, hierbei sollen alle Datensätze dauerhaft persistiert werden. Ein abschließendes Kriterium ist das Deployment in der IBM-Cloud.	5 hrs	Closed

3 Sprintbericht 20.04 – 08.05

3.1 Frontend – Device Liste – PSP: 3,4

Device-Liste-Komponente Version 0.1

Beschreibung

Im Vorprojekt ist das registrierte Gateway innerhalb einer Tabellen-Komponente einsehbar. Hierbei handelt es sich um eine reine Text-Ausgabe.

Akzeptanz

Die Komponente ist verwendbar und nimmt alle nötigen Props entgegen.

Durchführung

Mithilfe von Material-UI habe ich eine einfache Tabellen-Komponente erstellt. Die „props“-Verwendung wurde angepasst. In dieser Version werden insgesamt 2 **props** entgegengenommen.

- headData … Kopf der Tabelle - Spaltenbezeichnungen
- bodyData … gesamter Inhalt der Tabelle

Um an die Daten der props, welche zum einen als String-Array und zum anderen als Key-Value-Objekt-Liste mitgegeben werden, zu kommen, habe ich diese folgend durchiteriert:

```
const { classes, headData, bodyData } = props;

// data for the TableHead
const headItems = headData.map((item) =>
  <TableCell align={"center"}>{item}</TableCell>
);


// data for the TableBody
const bodyItems = bodyData.map((device) =>
  <TableRow>
    {Object.keys(device).map(key =>
      <TableCell align={"center"}>{device[key]}</TableCell>
    )}
  </TableRow>
);
```

Dokumentation

1. https://material-ui.com/demos/tables/#tables

Verwendung der Listen-Komponente

Beschreibung

Die Komponente muss in den Aufbau des Web-Interfaces integriert werden.

Akzeptanz

Die Komponente wird von der Eltern-Komponente angenommen und fehlerfrei angezeigt.

Durchführung

Die Komponente nimmt 2 verschiedene props an, welche derzeit mitgegeben werden müssen und dies in folgender Form:

```
<SimpleTable headData={["ID", "Name", "Device-Type", "Online/Offline", "Signal strength"]}
        bodyData={[
          {id: "1", name: "MockDevice", deviceType: "Multitech", status: "online", signalStrength: signal}
          ]}
/>
```

3.2 Frontend – Device Details – PSP: 3,6

Konzept entwickeln

Beschreibung

Erstellen eines Konzepts, zur Entwicklung und Umsetzung der Device Details

Akzeptanz

Nach Rücksprache 'OK' innerhalb des Teams.

Device-Details-Prototyp-Komponente

Beschreibung

Im Vorprojekt werden alle Daten innerhalb einer tabellenartigen Komponente ausgegeben.

Akzeptanz

Die Komponente ist verwendbar und nimmt alle nötigen **props** entgegen.

Durchführung

Die Device-Details-Komponente ist tabelarisch aufgebaut, nimmt Rohdaten des Gateways mittels **props** auf und gibt diese änderungslos aus.

Dokumentation

1. https://material-ui.com/demos/tables/#tables

Verwendung der Prototyp-Komponente

Beschreibung

Die Komponente muss in den Aufbau des Web-Interfaces integriert werden.

Akzeptanz

Die Komponente wird von der Eltern-Komponente angenommen und fehlerfrei angezeigt.

Durchführung

Die Rohdaten werden einzeln als props angegeben. Die Komponente erhält keine „children“.

```
<ExampleCard deviceId={deviceId} ip={ip} latestCheckin={latestCheckin} macAddress={macAddress} signal={signal}
uptime={uptime}/>
```

Grid-Implementierung

Beschreibung

Die GUI wird in Form eines Grids (Rasters) angezeigt.

Akzeptanz

Grid-Layout adaptiert und für alle wesentlichen Bildschirmgrößen einsehbar.

Durchführung

Das Grid-System von Material-UI bietet eine Rasteraufteilung der UI-Komponenten. Hierbei verwende ich zum einen eine „containe“-Grid-Komponente, welche die „item“-Grid-Komponenten umfasst und relativ positioniert. Eine Zeile des Grids enthält 12 Kolumnen. Wie viele Kolumnen ein Item umfassen soll, wird innerhalb der props xs, sm, md, lg oder xl als Ganzzahl angegeben.

```
<Grid item xs={12} sm={6} md={6} lg={6} xl={6}>
  <ExampleCard deviceId={deviceId} ip={ip} latestCheckin={latestCheckin} macAddress={macAddress} signal={signal}
uptime={uptime}/>
</Grid>
```

Dokumentation

1. https://material-ui.com/layout/grid/

3.3 Frontend – Api Calls – PSP: 3,7

Daten-Fetch von REST

Beschreibung

Es werden die, vom Backend bereitgestellten, Daten von einer REST-Schnittstelle geholt und ausgegeben.

Akzeptanz

Gerätedaten werden korrekt heruntergeladen und in die hierfür vorgesehenen Komponenten hineingesetzt.

Durchführung

Um Daten von einer REST-Schnittstelle herunterladen zu können, benötigt man einen „promise based http client“. Ich habe **Axios** verwendet und folgend installiert:

```
$ npm install axios
```

Nachdem ich Axios importiert hatte,

```
import axios from 'axios';
```

konnte ich mit der Implementierung loslegen.

Um eine, sich wiederholende, Datenabfrage zu erreichen, habe ich ein Intervall innerhalb der componentDidMount()-Methode angelegt, welches dann in der componentWillUnmount()-Methode „gecleart“ werden muss. Innerhalb des Intervalls wird die getData()-Methode aufgerufen, die folgenden Aufbau besitzt:

```
getData() {
  const {setData} = this;
    axios.get('https://tgm-backend-app.eu-de.mybluemix.net/gw1')
      .then(function (response) {
        // handle success
        console.log(response.data);
        setData(response);
      })
}
```

Hierbei handelt es sich um eine einfache GET-Request.

Dokumentation

1. https://stackoverflow.com/questions/51154269/reactjs-fetch-a-quote-every-5-seconds-with-axios-and-display-it-in-the-p
2. https://github.com/axios/axios

3.4 Frontend – Navbar – PSP: 3,8

Navbar-Komponente

Beschreibung

Es wird mit React und MaterialUI eine Navbar-Komponente aufgesetzt

Akzeptanz

Die Komponente ist verwendbar und nimmt alle nötigen Props entgegen.

Durchführung

navbar.jsx

```
<AppBar>
        <Toolbar>
            [...]
        </Toolbar>
</AppBar>
```

Aufbau einer Navbar in MaterialUI.

Dokumentation

1. https://material-ui.com/demos/app-bar/

Verwendung der Navbar-Komponente

Beschreibung

Die Komponente muss in den Aufbau des Web-Interfaces integriert werden.

Akzeptanz

Die Komponente wird von der Eltern-Komponente angenommen und fehlerfrei angezeigt.

Durchführung

navbar/index.js

```
import NavBar from './navbar.jsx';

export default NavBar;
```

Gestalten der Navbar-Komponente

Beschreibung

Die Navbar-Komponente wird so gestaltet, dass sie responsiv ist und alle Inhalte ideal lädt und positioniert.

Akzeptanz

Komponente ist auf allen Geräten responsiv

Durchführung

navbar.jsx

```
import { withStyles } from '@material-ui/core/styles';
import "./navbar.scss";
```

In der ersten Zeile wird von MaterialUI die Methode „withStyles“ geladen und eingebunden. Dieses wird dann zum Exportieren der Komponente benötigt

In der zweiten Zeile wird das externe SASS-File importiert.

```
export default withStyles(styles)(NavBar);
```

Am Ende des Files wird die Methode „withStyles“, welche die Konstante „styles“ als Paramter bekommt, verwendet, um die Komponente mit den angewendeten Styleänderungen zu exportieren.

navbar/navbar.jsx

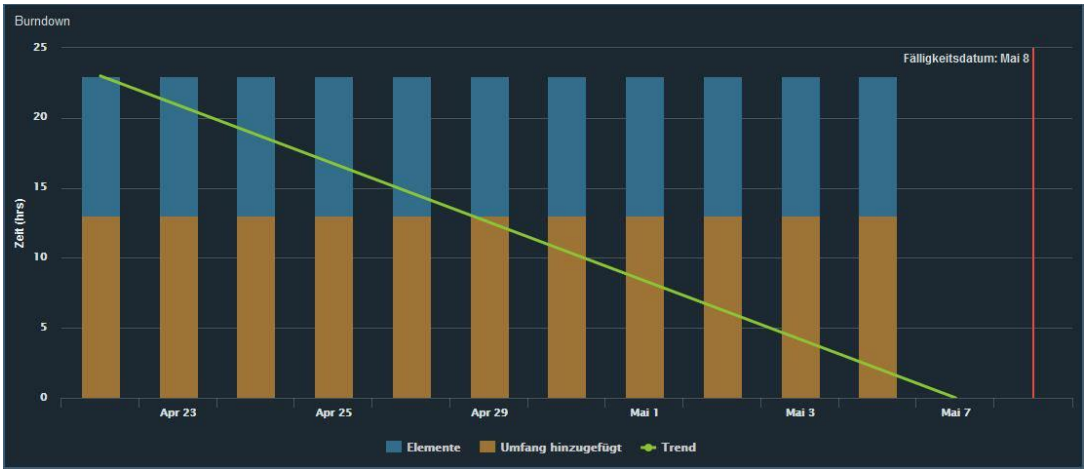
```
@import "../../../sass/variables";
```

Einbinden des SASS-Files, in welchem wir die wichtigsten Style-Attribute als Variablen gesetzt haben.

Dokumentation

1. https://material-ui.com/customization/overrides/

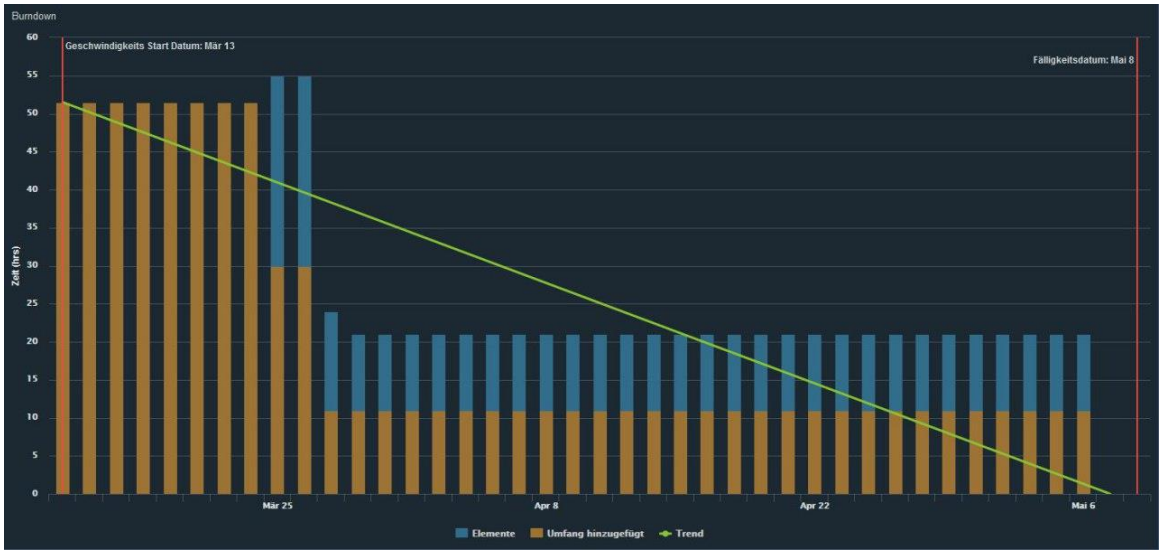
Sprint-Burndown-Chart



ID	Name	Zugewiesen	Akzeptanz	Schätzung	Status
4438	Daten-Fetch von REST	IoT Plattform IBM	Gerätedaten werden korrekt heruntergeladen und in die hierfür vorgesehenen Komponenten hineingesetzt.	1 hrs	Closed
4439	Frontend - Device Liste	Maciej Dzialoszynski		10 hrs	Closed
4440	Device-Liste-Komponente Version 0.1	Maciej Dzialoszynski	Die Komponente ist verwendbar und nimmt alle nötigen Props entgegen.	9 hrs	Closed
4441	Verwendung der Listen-Komponente	Maciej Dzialoszynski	Die Komponente wird von der Eltern-Komponente angenommen und fehlerfrei angezeigt.	1 hrs	Closed
4442	Frontend - Device Details	IoT Plattform IBM		6.5 hrs	Closed
4443	Konzept entwickeln	Sebastian Grünewald	Nach Rücksprache 'OK' innerhalb des Teams	1 hrs	Closed
4444	Device-Details-Prototyp-Komponente	Maciej Dzialoszynski	Die Komponente ist verwendbar und nimmt alle nötigen Props entgegen.	3 hrs	Closed
4445	Verwendung der Prototyp-Komponente	Maciej Dzialoszynski	Die Komponente wird von der Eltern-Komponente angenommen und fehlerfrei angezeigt.	0.5 hrs	Closed
4446	Grid-Implementierung	Sebastian Grünewald	Grid-Layout adaptiert und für alle wesentlichen Bildschirmgrößen einsehbar.	2 hrs	Closed
4447	Frontend - Navbar	IoT Plattform IBM		5 hrs	Closed
4448	Navbar-Komponente	Sebastian Grünewald	Die Komponente ist verwendbar und nimmt alle nötigen Props entgegen.	2 hrs	Closed

4449	Verwendung der Navbar-Komponente	Sebastian Grünewald	Die Komponente wird von der Eltern-Komponente angenommen und fehlerfrei angezeigt.	1 hrs	Closed
4450	Gestalten der Navbar-Komponente	Sebastian Grünewald	Komponente ist auf allen Geräten responsiv	2 hrs	Closed

Product-Burndown-Chart



Produktivität