**FH JOANNEUM - University of Applied Sciences**

**Smart Application Platform Graz**

**Dokumentation**

**Integriertes Praxisprojekt**

**Verfasser:**

**Buchsbaum Julia**

**Janusch Michael**

**Betreuer:**

**FH-Prof. Dipl.-Ing. Dr.techn. Peter Salhofer**

**Graz, März 2018**



Inhaltsverzeichnis

[Inhaltsverzeichnis 2](#_Toc510191371)

[Abbildungsverzeichnis 4](#_Toc510191372)

[1 Einleitung 5](#_Toc510191373)

[1.1 Problemstellung 5](#_Toc510191374)

[1.2 Zielsetzung 5](#_Toc510191375)

[2 Infrastruktur 6](#_Toc510191376)

[2.1 FIWARE Komponenten 6](#_Toc510191377)

[2.1.1 Orion 6](#_Toc510191378)

[2.1.2 IDAS 6](#_Toc510191379)

[2.1.3 Cygnus 6](#_Toc510191380)

[2.1.4 Comet 7](#_Toc510191381)

[2.1.5 Mongo DB 7](#_Toc510191382)

[2.1.6 WireCloud 7](#_Toc510191383)

[2.2 Docker Infrastruktur Setup 7](#_Toc510191384)

[2.3 FIWARE Lab Infrastruktur Setup 10](#_Toc510191385)

[2.4 Testdaten anlegen 12](#_Toc510191386)

[3 Dashboard 13](#_Toc510191387)

[3.1 Wirecloud 13](#_Toc510191388)

[3.1.1 Widgets 13](#_Toc510191389)

[3.1.2 Operatoren 14](#_Toc510191390)

[3.1.3 Themes 15](#_Toc510191391)

[3.2 Vorhandene Widgets und Operatoren 17](#_Toc510191392)

[3.2.1 NGSI Source Operator 18](#_Toc510191393)

[3.2.2 NGSI Entity To PoI Operator 18](#_Toc510191394)

[3.2.3 NGSI Browser Widget 18](#_Toc510191395)

[3.2.4 JSON Editor Widget 19](#_Toc510191396)

[3.2.5 STH Source Operator 20](#_Toc510191397)

[3.2.6 Google Charts Widget 20](#_Toc510191398)

[3.3 Custom Widgets und Operatoren 21](#_Toc510191399)

[3.3.1 Attribute Selector Widget 22](#_Toc510191400)

[3.3.2 STH Multiple Entities Source Operator 23](#_Toc510191401)

[3.3.3 STH Location Operator 23](#_Toc510191402)

[3.3.4 STH Data to PoIs Operator 24](#_Toc510191403)

[3.3.5 PoI Beautifier Operator 25](#_Toc510191404)

[3.3.6 OpenLayers Map Widget 25](#_Toc510191405)

[3.3.7 Min/Max Selector Operator 27](#_Toc510191406)

[3.3.8 Heatmap Input Widget 27](#_Toc510191407)

[3.3.9 STH Multi Input Source Operator 29](#_Toc510191408)

[3.3.10 OpenLayers Historic Heatmap Widget 29](#_Toc510191409)

[3.3.11 Chart Input Widget 29](#_Toc510191410)

[3.3.12 STH Source Chart Operator 31](#_Toc510191411)

[3.3.13 Google Chart Operator 31](#_Toc510191412)

[3.3.14 Gauge Chart Operator 32](#_Toc510191413)

[3.3.15 IDAS & ORION REST Calls Operatoren 32](#_Toc510191414)

[3.3.16 Service/Entity/Device/Subscription Manager Widgets 32](#_Toc510191415)

[3.4 Use Cases 33](#_Toc510191416)

[3.4.1 Darstellung von Entitäten auf einer Karte 33](#_Toc510191417)

[3.4.2 Diagramme über die letzten Werte 34](#_Toc510191418)

[3.4.3 Darstellung von historischen Werte mit einer Heatmap 35](#_Toc510191419)

[3.4.4 Formular Abfrage von historischen Werten 35](#_Toc510191420)

[3.4.5 Administrations Tools 36](#_Toc510191421)

[4 Security 38](#_Toc510191422)

[5 Raspberry Pi Sensoraufbau 38](#_Toc510191423)

[6 Fazit 39](#_Toc510191424)

[7 Linksammlung 40](#_Toc510191425)

Abbildungsverzeichnis

[Abbildung 1: Docker Compose File 9](#_Toc510191426)

[Abbildung 2: Order Struktur der Konfigurationsordner 10](#_Toc510191427)

[Abbildung 3: Instanzmenü von FIWARE Lab 11](#_Toc510191428)

[Abbildung 4: Sicherheitsgruppen in FIWARE Lab 11](#_Toc510191429)

[Abbildung 5: Struktur von WireCloud Widgets/Operatoren 14](#_Toc510191430)

[Abbildung 6: Struktur für WireCloud Themes 15](#_Toc510191431)

[Abbildung 7: Struktur des Custom Themes für WireCloud 16](#_Toc510191432)

[Abbildung 8: Docker Compose Konfiguration für die WireCloud Komponente 17](#_Toc510191433)

[Abbildung 9: NGSI Browser Widget 19](#_Toc510191434)

[Abbildung 10: JSON Editor Widget 19](#_Toc510191435)

[Abbildung 11: JSON Objekt für eine GeoChart Darstellung mit dem Google Chart Widget 21](#_Toc510191436)

[Abbildung 12: JSON Objekt für eine ComboChart Darstellung mit dem Google Chart Widget 21](#_Toc510191437)

[Abbildung 13: Attribute Selector Widget Wiring 22](#_Toc510191438)

[Abbildung 14: Attribute Selector Widget 22](#_Toc510191439)

[Abbildung 15: STH Multiple Entities Source Operator Wiring 23](#_Toc510191440)

[Abbildung 16: STH Location Operator Wiring 24](#_Toc510191441)

[Abbildung 17: OpenLayers Map Widget Einstellungen 26](#_Toc510191442)

[Abbildung 18: Min/Max Selector Wiring 27](#_Toc510191443)

[Abbildung 19: Heatmap Input Widget 28](#_Toc510191444)

[Abbildung 20: Heatmap Input Widget Wiring 28](#_Toc510191445)

[Abbildung 21: STH Multi Input Source Operator Wiring 29](#_Toc510191446)

[Abbildung 22: Chart Input Widget 30](#_Toc510191447)

[Abbildung 23: Chart Input Widget Wiring 31](#_Toc510191448)

[Abbildung 24: Google Chart Operator Wiring 32](#_Toc510191449)

[Abbildung 25: Entity Manager Widget 33](#_Toc510191450)

1. Einleitung
   1. Problemstellung
   2. Zielsetzung
2. Infrastruktur

Im folgenden Kapitel wird die Infrastruktur welche für das Projekt geplant und implementiert wurde näher beschrieben. Zuerst werden die verwendeten FIWARE Komponenten kurz beschrieben, danach die Infrastruktur mit Docker und zum Schluss die FIWARE Lab Infrastruktur.

* 1. FIWARE Komponenten
     1. Orion

Der Context Broker genannt Orion ist ein zentraler Teil jeder FIWARE Infrastruktur. Die Aufgabe des Context Brokers ist das Management des gesamten Lebenszyklus von Context Informationen. Das bedeutet, der Context Broker ermöglicht das Erstellen von Context Elementen (Entiäten) und das Managen von Updates, Queries und Subscriptions. Für diese Aufgaben wird eine REST Schnittstelle basierenden auf NGSIv2 Spezifikation zur Verfügung gestellt.

Weitere Informationen unter 🡪 https://fiware-orion.readthedocs.io/en/master/

* + 1. IDAS

Ein weiterer wichtiger Teil der Infrastruktur ist der IoT Agent, genannt IDAS. Der IoT Agent ist die Schnittstelle zwischen den unterschiedlichen Sensoren und dem Context Broker. Die Kommunikation zwischen dem IoT Agent und den Sensoren kann über folgende Protokolle erfolgen:

* IDAS Ultralight2.0 & MQTT
* IDAS LWM2M

Bevor Sensordaten empfangen werden können müssen aber sogenannte Services und Devices angelegt werden. Dies geschieht über die von dem IoT Agent zur Verfügung gestellte REST API. Ein Service muss über einen eindeutigen „apikey“ verfügen und wird über diesen identifiziert. Ein Sensor wird als Device registriert und wird einem Service zugeordnet. Außerdem wird das Device auf ein Attribut auf ein Attribut einer bestehenden Entität gemapped.

Weitere Informationen 🡪 http://fiware-iot-stack.readthedocs.io/en/latest/device\_  
gateway/index.html

* + 1. Cygnus

Für das Speichern von historischen Werten wird die FIWARE Komponente Cygnus benötigt. Diese Komponente basiert auf Apache Flume und speichert die Daten von Events des Context Brokers in einem persistenten Speicher. Welche persistenten Speicher unterstützt werden, kann aus der offiziellen Dokumentation entnommen werden.

Für jedes Attribut einer Entität, deren historische Werte gespeichert werden sollten, muss eine Subscription auf den Context Broker angelegt werden. Somit ist Cygnus eine Schnittstelle zwischen dem Context Broker und einem persistenten Speicher.

Weitere Informationen 🡪 http://fiware-cygnus.readthedocs.io/en/latest/

* + 1. Comet

Für das Auslesen der historischen Werte wird die Short Term Historic (STH) Komponente, genannt Comet, benötigt. Wie auch die anderen Komponenten verfügt auch Comet über eine REST API, welche für Abfragen verwendet werden kann. Die Daten können in Rohform oder in aggregierter Form abgefragt werden.

Weitere Informationen 🡪 https://fiware-sth-comet.readthedocs.io/en/latest/index.html

* + 1. Mongo DB

Für das Projekt wurde außerdem eine Mongo DB als persistenter Datenspeicher ausgewählt. Die Komponenten Orion, IDAS, Cygnus und Comet greifen auf diese Datenbank zu.

* + 1. WireCloud

WireCloud ist eine weitere Komponente der FIWARE Plattform und ermöglicht die Erstellung von unterschiedlichen Dashboards in Form einer Web Applikation. Die Dashboards werden im WireCloud Kontext als Mashups bezeichnet und werden mit sogenannten Widgets und Operatoren aufgebaut. Diese Bausteine verfügen über Input sowie auch Output Endpunkte und können über diese miteinander verbunden werden.

Neben der Mongo DB für die Backend Komponenten wird eine Postgres DB für WireCloud benötigt.

Weitere Informationen 🡪 https://wirecloud.readthedocs.io/en/stable/

* 1. Docker Infrastruktur Setup

Im ersten Schritt des Projektes wurde die geplante Infrastruktur mithilfe von Docker umgesetzt. Die Basis für die Infrastruktur lieferte das IPPR Projekt des Jahrganges 2015 vom WS 2016/17, in welchem die einzelnen Komponenten und ihr Zusammenspiel bereit evaluiert wurden. Einige Teile wurden von diesem Projekt übernommen, jedoch wurde darauf geachtet, dass die Docker Images am neusten Stand sind und dass wenn möglich Images von FIWARE verwendet werden, anstatt angepasster Images.

Im Laufe des Projektes wurde das Docker Compose File einige Male angepasst und überarbeitet. Auch einige neue Komponenten wurden hinzugefügt. Im Folgenden wird das finale Docker Compose File dargestellt.

version: '2'

services:

mongodb:

restart: on-failure

image: mongo:latest

hostname: mongodb

container\_name: mongodb

expose:

- "27017"

ports:

- "27017:27017"

command: --smallfiles

orion:

restart: on-failure

image: fiware/orion:latest

hostname: orion

container\_name: orion

links:

- mongodb

expose:

- "1026"

ports:

- "1026:1026"

command: -dbhost mongodb

cygnus:

restart: on-failure

image: telefonicaiot/fiware-cygnus

hostname: cygnus

container\_name: cygnus

links:

- mongodb:iot-mongo

expose:

- "8081"

- "5050"

ports:

- "8081:8081"

- "5050:5050"

volumes:

- "./cygnus\_config/agent.conf:/opt/apache-flume/conf/agent.conf:ro"

comet:

restart: on-failure

image: fiware/sth-comet

hostname: comet

container\_name: comet

links:

- mongodb

expose:

- "8666"

ports:

- "8666:8666"

volumes:

- "./comet\_config/config.js:/opt/sth/config.js:ro"

idas:

restart: on-failure

image: fiware/iotagent-ul:latest

hostname: idas

container\_name: idas

links:

- orion

- mongodb:mongo

expose:

- "7896"

- "4041"

ports:

- "7896:7896"

- "4041:4041"

nginx:

restart: on-failure

image: nginx:latest

hostname: nginx

container\_name: nginx

ports:

- "80:80"

- "443:443"

volumes:

- ./nginx\_config/nginx.conf:/etc/nginx/nginx.conf:ro

- ./nginx\_config/certs/:/etc/nginx/certs/:ro

- letsencrypt\_www:/var/www/letsencrypt/

volumes\_from:

- wirecloud

links:

- wirecloud:wirecloud

certbot:

image: certbot/certbot:latest

hostname: certbot

container\_name: certbot

volumes:

- ./nginx\_config/certs/:/etc/letsencrypt/

- letsencrypt\_www:/var/www/letsencrypt/

postgres:

restart: on-failure

image: postgres:latest

hostname: postgres

container\_name: postgres

volumes\_from:

- data

ports:

- "5432:5432"

data:

restart: on-failure

image: postgres:latest

hostname: data

container\_name: data

volumes:

- /var/lib/postgresql

command: /bin/true

wirecloud:

restart: on-failure

image: fiware/wirecloud:latest-composable

hostname: wirecloud

container\_name: wirecloud

volumes:

- ./wirecloud\_config/graziotthemev2/:/opt/wirecloud\_instance/graziotthemev2/:ro

- ./wirecloud\_config/settings.py:/opt/wirecloud\_instance/wirecloud\_instance/settings.py:ro

links:

- postgres:postgres

ngsi-proxy:

restart: on-failure

image: januschm/ngsi-proxy:latest

hostname: ngsi-proxy

container\_name: ngsi-proxy

expose:

- "3000"

ports:

- "3000:3000"

volumes:

letsencrypt\_www: ~

Abbildung : Docker Compose File

Neben den Docker Compose File werden für das Setup der lokalen Infrastruktur auch noch die Config Ordner mit den entsprechenden Config Dateien für Comet, Cygnus, Nginx und Wirecloud benötigt. Alle Dateien können aus dem GitHub Repository entnommen werden.

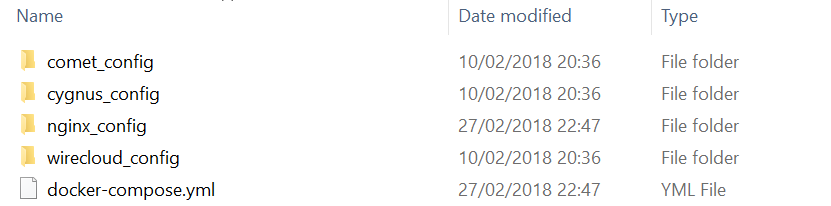


Abbildung : Order Struktur der Konfigurationsordner

Das Setup kann mit folgenden Befehl gestartet werden, vorausgesetzt Docker ist installiert und gestartet:

docker-compose up

Nachdem alle Images heruntergeladen und alle Container erstellt wurden kann mit folgenden Befehl überprüft werden ob alle Container erfolgreich gestartet wurden:

docker container ls

Wenn alle Container laufen, muss zuerst die WireCloud Datenbank initialisiert werden und ein Benutzer angelegt werden. Dies wird mithilfe von folgenden Befehlen durchgeführt:

docker exec -it wirecloud /docker-entrypoint.sh initdb

docker exec -it wirecloud /docker-entrypoint.sh createsuperuser

Danach ist es bereits möglich sich auf WireCloud einzuloggen. Dafür muss man lediglich WireCloud (localhost) im Browser öffnen.

* 1. FIWARE Lab Infrastruktur Setup

FIWARE Lab ist eine Cloud-Umgebung, die von FIWARE für Testszenarien zur Verfügung gestellt wird. Registrierte Benutzer können sich über den Online-Dienst virtuelle Maschinen einrichten, auf denen ausgewählte Serverkomponenten der FIWARE-Infrastruktur laufen, welche direkt aus dem Internet erreichbar sind. Dadurch können Anwendungsszenarien getestet werden, für die andernfalls teure Cloud-Dienste benötigt würden.

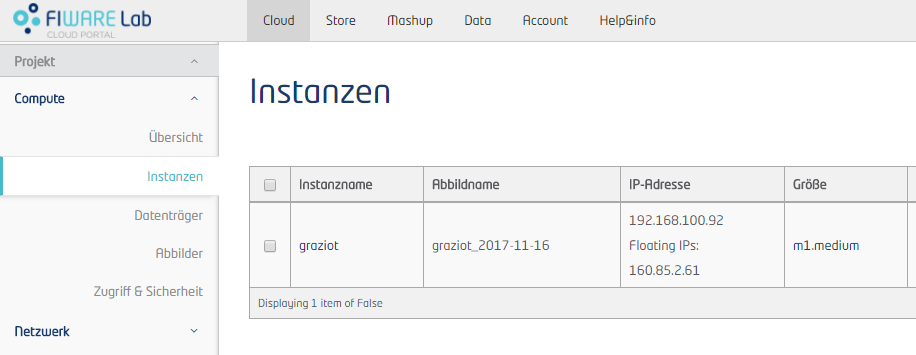


Abbildung : Instanzmenü von FIWARE Lab

Um eine FIWARE-Infrastruktur in FIWARE Lab einzurichten, wird zumindest eine Instanz benötigt. Diese kann im Abschnitt „Instanzen“ im FIWARE Lab-Menü erstellt werden. Als Basis für die neue Instanz wird ein vorgefertigtes Abbild ausgewählt, welches verschiedene Betriebssysteme und Konfigurationen enthalten kann. Für unsere Zwecke wurde ein Ubuntu-Abbild gewählt.

Während der Einrichtung wird neben verschiedenen Einstellungen mit Bezug auf Leistungsparameter und verfügbaren Speicher auch eine öffentliche IP-Adresse gesetzt, damit die Instanz aus dem Internet erreicht werden kann. Mithilfe des Private Keys, der ebenfalls bei der Einrichtung erstellt wird, kann im Anschluss die laufende Instanz mit einem geeigneten Programm über SSH gewartet werden.



Abbildung : Sicherheitsgruppen in FIWARE Lab

Um im weiteren Verlauf die reibungslose Kommunikation mit den Serverkomponenten der Instanz zu ermöglichen, müssen in der Sicherheitsgruppe der entsprechenden Instanz noch die benötigten Ports geöffnet werden.

Nach der erfolgreichen Einrichtung der Cloud-Instanz kann die Konfiguration der FIWARE-Serverkomponenten durchgeführt werden. Über eine SSH-Verbindung muss dafür zuerst über den Paketmanager der Instanz die aktuelle Version von Docker installiert werden. Danach müssen noch alle für die Docker-Infrastruktur benötigten Dateien (beschrieben im vorangegangenen Kapitel) auf die Instanz übertragen werden.

Das abschließende Starten der FIWARE-Komponenten kann über die SSH-Verbindung mit der in Kapitel 2.2 erklärten Prozedur durchgeführt werden. Ab diesem Zeitpunkt kann auf die FIWARE-Komponenten über das Internet zugegriffen werden.

* 1. Testdaten anlegen

Das Anlegen der Testdaten für die FIWARE-Infrastruktur erfolgt generell über die REST-Interfaces des IoT Agents und des Context Broker Servers. Um diesen Vorgang zu vereinfachen, wurden mehrere Python-Skripte erstellt, mit denen alle benötigten Datenbankobjekte angelegt werden können.

Die Funktionalitäten sind auf zwei Skripte aufgeteilt, die im anschließenden Abschnitt kurz erläutert werden. Beide Skripte sind auch im GitHub Repository zu finden.

**CreateObjects.py**

Dieses Skript dient dazu, Services, Entities und Devices für die FIWARE-Infrastruktur anzulegen. Das Python-Skript erwartet als erstes Argument das Kürzel des anzulegenden Objekts („srv“, „ent“ oder „dev“) und nimmt als zweiten Parameter den Pfad zu der Datei, welche die Objekteigenschaften im JSON-Format enthält. Zusätzliche Informationen, wie die IP-Adresse des entsprechenden Servers, sind in der Datei „config.ini“ hinterlegt.

**CreateSubscriptions.py**

Dieses Python-Skript dient dazu, Subscriptions für einzelne Attribute eines Entitys anzulegen. Bei der Ausführung erwartet das Skript als Parameter den Pfad zu der Datei, welche die Eigenschaften der anzulegenden Subscriptions im Format „<entity\_id>,<entity\_type >,<attribute>“ enthält. Auch dieses Skript liest zusätzliche Informationen aus der Datei „config.ini“ ein.

1. Dashboard

Eines der Hauptziele des Projektes war die Evaluierung und Implementierung von der FIWARE Komponente WireCloud als Dashboard Möglichkeit.

* 1. Wirecloud

WireCloud ist ein Teil der FIWARE Plattform und ermöglicht die Erstellung von Dashboards in Form einer Web Applikation. Somit stelle WireCloud die Frontend Komponente einer FIWARE Infrastruktur dar. Die Dashboards werden im WireCloud Kontext als Mashups bezeichnet und werden mit sogenannten Widgets und Operatoren aufgebaut. Diese Bausteine verfügen über Input sowie auch Output Endpunkte und können über diese miteinander verbunden werden. Im Folgenden werden Widgets und Operatoren, sowie WireCloud Themes näher beschrieben. Diese Beschreibung fasst die wichtigsten Punkte der offiziellen WireCloud Dokumentation [1] zusammen.

* + 1. Widgets

**Basis Konzepte**

Folgende Basis Konzepte sollten bei dem Design und der Entwicklung von Widgets beachtet werden:

* Widgets sollten kleine und wiederverwendbare Web Applikationen darstellen.
* Widgets sollten so generisch wie möglich sein, können aber in Ausnahmefällen auch spezifische, ad-hoc Lösungen darstellen.
* Widgets sollten an reale Probleme angepasst werden.
* Widgets sind hauptsächlich Elemente des Front Ends. Widgets können zwar direkt auf Backend Services zugreifen, diese Aufgabe sollte aber wenn möglich von Operatoren übernommen werden.
* Während der Entwicklung von Widgets kann jede Technologie, welche von Web Browsern akzeptiert wird verwendet werden.

**Aufbau von Widgets**

Widgets bestehen grundsätzlich aus den folgenden Komponenten:

*Beschreibung (config.xml)*

Das config.xml beinhaltet eine deklarative Beschreibung des Widgets. Unter anderen beinhaltet dies Beschreibung folgende Punkte:

* Name und Version
* Details (z. B. Name des Autors, E-Mail-Adresse, Referenzen zu Read Me Files, etc.)
* Definition von Input sowie Output Endpunkten
* Optionen für unterschiedliche Einstellungen
* Referenz zum index.html
* Rendering Einstellungen

Das File wir mithilfe der sogenannten Mashable Application Component Description Language (MACDL) geschrieben. Beispiel für config.xml Dateien können im Github Repository gefunden werden.

*Code*

Die zweite Komponente eines Widgets ist der eigentliche Code, welcher zum Beispiel aus HTML, JavaScript und CSS Dateien besteht. Diese Dateien beinhalten die Implementierung eines Widgets und können auch die sogenannte Widget API [2] verwenden.

*Statische Ressourcen*

Zu den statischen Ressourcen zählen z. B. Bilder, Dokumentation oder ähnliches.

**Packaging**

Alle Komponenten von einem Widget werden in ein zip File verpackt. Die Dateiendung wird jedoch von zip auf wgt geändert. Folgende Abbildung zeigt eine mögliche Struktur eines Widgets.



Abbildung : Struktur von WireCloud Widgets/Operatoren

Im Zuge des Projektes wurden außerdem npm und Bower als Package Manager sowie auch der Taskrunner Grunt verwendet um bestimmte Aufgaben wie das Laden von Komponenten, das Kompilieren und das Packaging zu vereinfachen. Beispiele für package.json, bower.json und gruntfile.js Dateien können im Github Repository gefunden werden.

* + 1. Operatoren

Operatoren verfügen über einen ähnlichen Aufbau wie Widgets. Der größte Unterschied zu Widgets liegt darin, dass Operatoren über keine visuellen Komponenten verfügen. Aus diesem Grund referenziert die config.xml Datei nicht auf ein index.html sondern auf die jeweiligen Scripts, welche den Code beinhalten.

Grundsätzlich werden Operatoren in drei Kategorien eingeteilt:

* **Data Source Operators** 🡪 stellen Informationen zur Verfügung, welche von anderen Widgets oder Operatoren verwendet werden.
* **Data Target Operators** 🡪 stellen Informationen zur Verfügung und verwenden diese um bestimmte Aufgaben durchzuführen.
* **Data Transformation Operators** 🡪 transformieren Daten, um sie für die Verwendung von Widgets und Operatoren aufzubereiten.
  + 1. Themes

**Allgemeines**

Wirecloud bietet auch die Möglichkeit neue Themes für das Dashboard zu erstellen oder Teile von bereits vorhanden Themes zu überschreiben. Dafür muss zuerst ein neuer Ordner mit folgenden Struktur erstellt werden:

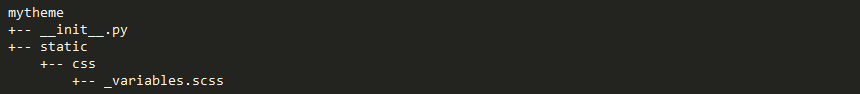


Abbildung : Struktur für WireCloud Themes

Laut der offiziellen WireCloud Dokumentation wird die \_\_init\_\_.py Datei für die Plug-in Architektur, welche von WireCloud verwendet wird benötigt. Die Datei leer sein, muss jedoch vorhanden sein.

Grundsätzlich leiten alle Themes vom Default WireCloud Theme ab, wenn ein anderes Theme als Basis verwendet werden sollte muss die folgende Zeile in der \_\_init\_\_.py Datei hinzugefügt werden:



Weitere Informationen welche Templates zur Verfügung stellen und welche Variablen überschrieben werden können, können in der WireCloud Themes Dokumenation [3] entnommen werden. Im folgenden Abschnitt liegt der Fokus auf den Einstellungen, welche getroffen werden müssen um ein angepasstes Theme zu verwenden.

**Ändern eines WireCloud Themes**

Folgende Anleitung beschreibt die Schritte, welche durchgeführt werden müssen um ein WireCloud Theme von einer lokalen Docker Umgebung zu ändern. Mit geringen Anpassungen können dieselben Schritte für eine Docker Umgebung auf einer FIWARE Lab Instanz verwendet.

*Ordner Struktur*

Wie bereits erwähnt muss zu Beginn die passende Ordner Struktur erstellt werden. Danach kann im defaultheme von WireCloud nach dem Teil, welcher überschrieben werden sollte gesucht werden. Das WireCloud Projekt kann von Github [4] geklont werden. In diesem Beispiel wird das Header Logo und einige Variablen überschrieben, wofür folgende Ordner Struktur erstellt wurde:

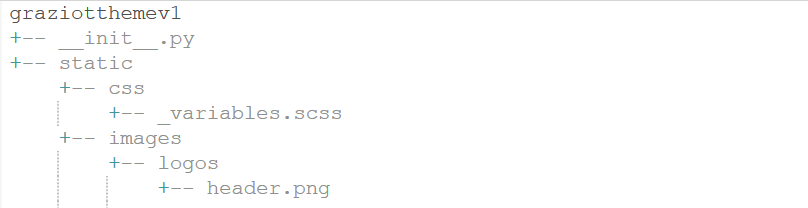


Abbildung : Struktur des Custom Themes für WireCloud

Wichtig dabei ist, dass die Datei oder das Bild, welches überschrieben werden sollte, den gleichen Dateinamen und Pfad wie im Default Theme hat.

Im nächsten Schritt wird der gesamte Ordner (graziotthemev1) auf die WireCloud Instanz in den Ordner /opt/wirecloud\_instance kopiert. Dies kann mit folgenden Docker Befehl durchgeführt werden:

docker cp graziotthemev1 wirecloud:/opt/wirecloud\_instance/graziotthemev1

Nach jeder neuen Änderung im Theme Ordner, muss der Ordner umbenannt werden, da WireCloud sonst keine Änderungen erkennt.

*Settings.py*

Danach muss die settings.py Datei, welche sich unter /opt/wirecloud\_instance/wirecloud\_instance befindet bearbeitet werden, so dass das neue Theme referenziert wird. Dafür muss die folgende Zeile geändert werden:

THEME\_ACTIVE = "graziotthemev1"

*Laden des neues Themes*

Um das neue Thema zu laden, müssen alle Docker Container, welche für WireCloud relevant sind neugestartet werden. Am sichersten ist es jedoch gleich alle FIWARE Container zu stoppen und wieder zu starten. Dies kann entweder mit den folgenden Docker Befehlen durchgeführt werden:

docker stop @(docker ps -aq)

docker start @(docker ps -aq)

Alternativ können auch Docker Compose Befehle durchgeführt werden, dafür muss das Terminal im selben Pfad geöffnet werden, in welchen sich auch die docker-compose.yml Datei befindet:

docker-compose stop

docker-compose start

Nach dem Restart sollte überprüft werden ob alle Container erfolgreich gestartet werden konnten. Dies geschieht mit dem Befehl:

docker container ls

Wenn alle Container laufen, muss man sich auf die WireCloud Instanz verbinden und alle statischen Ressourcen neu laden. Dies geschieht mithilfe von folgenden Befehlen:

docker exec -it wirecloud bash

python manage.py collectstatic

*Ordner mounten*

Werden häufig Änderungen am Theme vorgenommen, ist es sinnvoll die betroffenen Ordner mithilfe der docker-compose.yml Datei zu mounten. Die entsprechende Konfiguration dafür in der docker-compose.yml Datei sieht folgendermaßen aus:

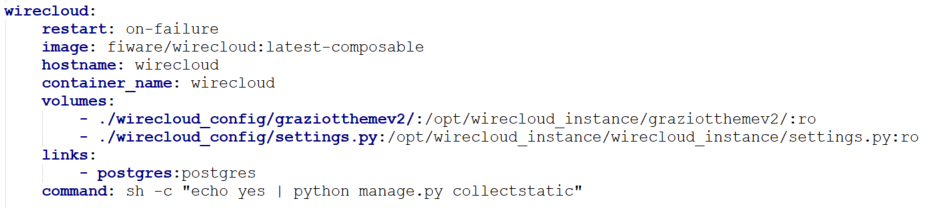


Abbildung : Docker Compose Konfiguration für die WireCloud Komponente

Es wurde nicht getestet ob für diese Einstellung nach einer Änderung des Themes es ausreichend ist, die Container neu zu starten oder ob bei jeder Änderung ein docker-compose up ausgeführt werden muss.

* 1. Vorhandene Widgets und Operatoren

Im Laufe des Projektes wurden eine Vielzahl an vorhanden Widgets und Operatoren verwendet, welche im FIWARE Store [5] sowie auch dem WireCloud Widget Github Repository [6] gefunden werden können. Viele Widgets wurden für die Anforderungen des Projektes leicht modifiziert. Im Folgenden werden die wichtigsten Widgets und ihre Verwendung näher beschrieben.

* + 1. NGSI Source Operator

Der NGSI Source Operator [7] ermöglicht die Verwendung des Orion Context Brokers als Datenquelle. Dabei wird eine Subskription erstellt, um Benachrichtigen zu erhalten, sobald ausgewählte Attribute einer Entität upgedated werden. Der Operator kann ein oder mehrere Entitäten hintereinander zur Verfügung stellen. Informationen über die grundlegenden Einstellungen des Operators und die Input sowie Output Endpunkte können aus dem User Guide von dem GitHub Repository entnommen werden.

Wichtig zu erwähnen ist, dass für die Anforderungen des Projektes der NGSI Source Operator um einen Reload Input Endpunkt erweitert wurde. Der Endpunkt dient dazu, einen Reload der NGSI Source auszulösen.

* + 1. NGSI Entity To PoI Operator

Der NGSI Entity to PoI Operator [8] kann mit dem NGSI Source Operator verbunden werden. Die Aufgabe des Operators ist die Umwandlung einer Entität in einen Point of Interest (PoI), welcher auf einer Map dargestellt werden kann. Neben dem Attribut, welches über die Koordinaten verfügt kann auch ein Icon in den Einstellungen ausgewählt werden. Grundsätzlich bietet das Widget die Option einen Link zu einem Icon anzugeben. Für das Projekt wurden außerdem statische Ressourcen für die Icons hinzugefügt. Für die jeweiligen Icons müssen die Keywords **lamp**, **tram** oder **bus** eingegeben werden.

* + 1. NGSI Browser Widget

Ein Widget, welches ebenfalls den Orion Context Broker als Datenquelle verwendet ist der NGSI Browser [9]. Im Gegensatz zur NGSI Source werden hier Veränderungen aber nicht sofort angezeigt, um neue Daten zu erhalten, muss das Widget neu geladen werden. Mit dem NGSI Browser können nicht nur alle Entitäten oder ausgewählte Entitäten in Form einer Liste angezeigt werden, sondern auch bearbeitet oder gelöscht werden. Außerdem können auch neue hinzugefügt werden. Für das Bearbeiten und das Hinzufügen wird jedoch das JSON Editor Widget benötigt. Welches im nächsten Unterkapitel näher beschrieben wird.

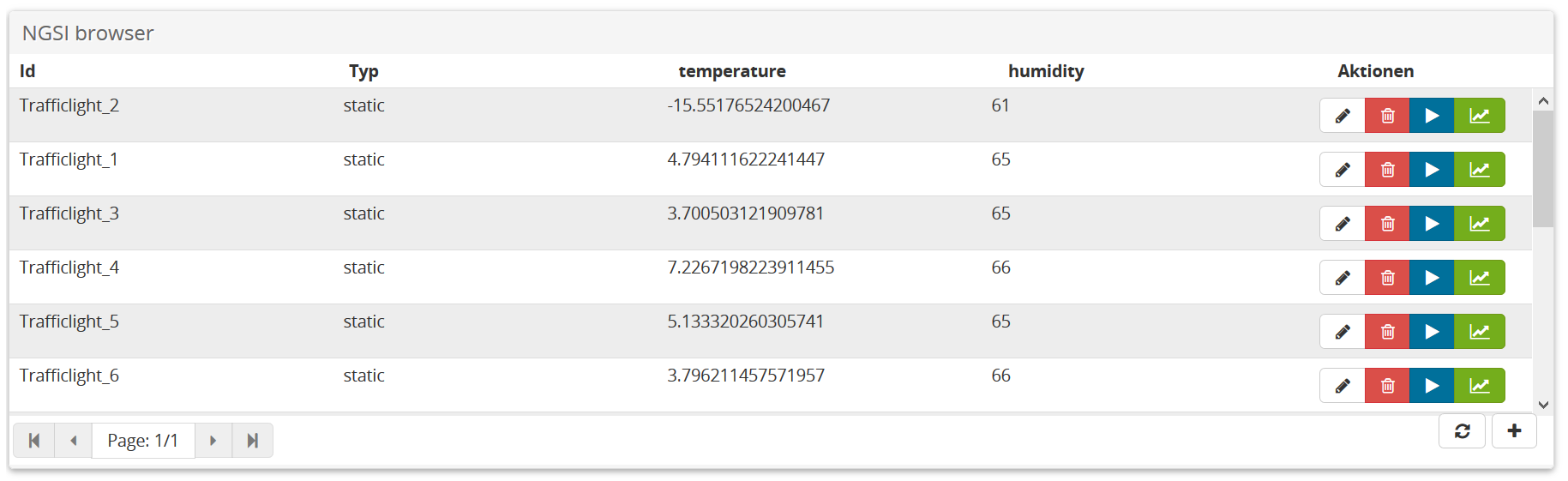


Abbildung : NGSI Browser Widget

In unserer modifizierten Version des NGSI Browsers wurde ein Button für Diagramme hinzugefügt, sowie auch ein Refresh Button um ein schnelles Reloaden zu ermöglichen. Bei einem Klick auf den Diagramm Button wird die jeweilige Entität an den Output Endpunkt gesendet und kann somit von anderen Widgets und Operatoren wieder als Input verwendet werden. Weitere Informationen über die Einstellungen und die Input und Output Endpunkte können dem GitHub Repository entnommen werden.

Im Zuge des Projektes wird der NGSI Browser vor allem als Source für die Diagramm Darstellung verwendet.

* + 1. JSON Editor Widget

Das JSON Editor Widget, ist ein Widget, welches dazu verwendet wird, um Entitäten hinzuzufügen oder zu bearbeiten. Es wird direkt vom NGSI Browser oder ähnlichen Widgets aufgerufen und muss somit nicht mit in das Wiring eingebunden werden.

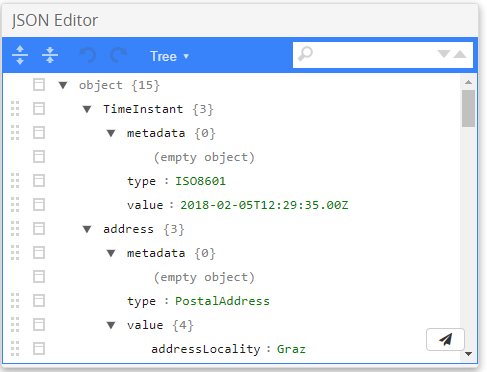


Abbildung : JSON Editor Widget

* + 1. STH Source Operator

Der STH Source Operator ist ein Operator, welcher das Abfragen von historischen Werten von Entitäten ermöglicht. Für diese Abfragen wird die Short Term Historict Komponente, genannt Coment, benötigt. Über einen Input Entpunkt wird eine Entität, zum Beispiel von einer NGSI Source oder dem NGSI Browser, an den STH Source Operator gesendet. In den Einstellungen kann festegelegt werden für welches Attribut die Abfrage erfolgen sollte und wieviele Werte abegfragt werden sollten. Der Operator verfügt über zwei Output Endpunkte. Einerseit werden die Attributwerte gesendet und anderseits die jeweiligen Zeitstempel. Diese Werte können danach von anderen Operatoren zum Beispiel für das Google Charts Widget aufbereitet werden.

Der STH Source Operator wurde für die Anforderungen des Projektes leicht angepasst. Außerdem diente der Operator als Basis für unterschiedliche andere Short Term Historic Operatoren um unterschiedliche Anforderungen zu erfüllen.

Wichtig zu beachten gilt, dass die Github Doku für den STH Source Operator nicht mit der aktuellen Version 1.0 übereinstimmt.

* + 1. Google Charts Widget

Das Google Charts Widget bietet die Möglichkeit Diagramme mithilfe von Google Charts darzustellen. Das Widget verfügt über einen Input Endpunkt genannt „Data in“. Dieser Endpunkt erwarten Daten im JSON Format mit den folgenden Attributen:

* Type (String) 🡪 Name der Visualisierungsklasse für das Diagramm
* Options (Objekt) 🡪 Optionen wie Höhe, Breite, Hintergrundfarben, etc.
* Data 🡪 Daten für das Diagramm

Beispiele für „Data in“ JSON Objekte werden im folgenden dargestellt:

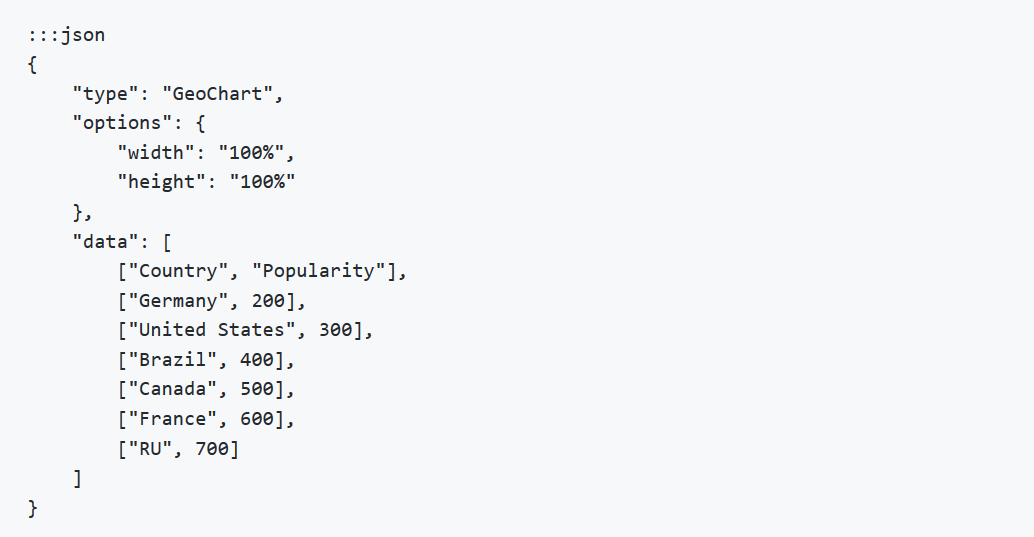


Abbildung : JSON Objekt für eine GeoChart Darstellung mit dem Google Chart Widget

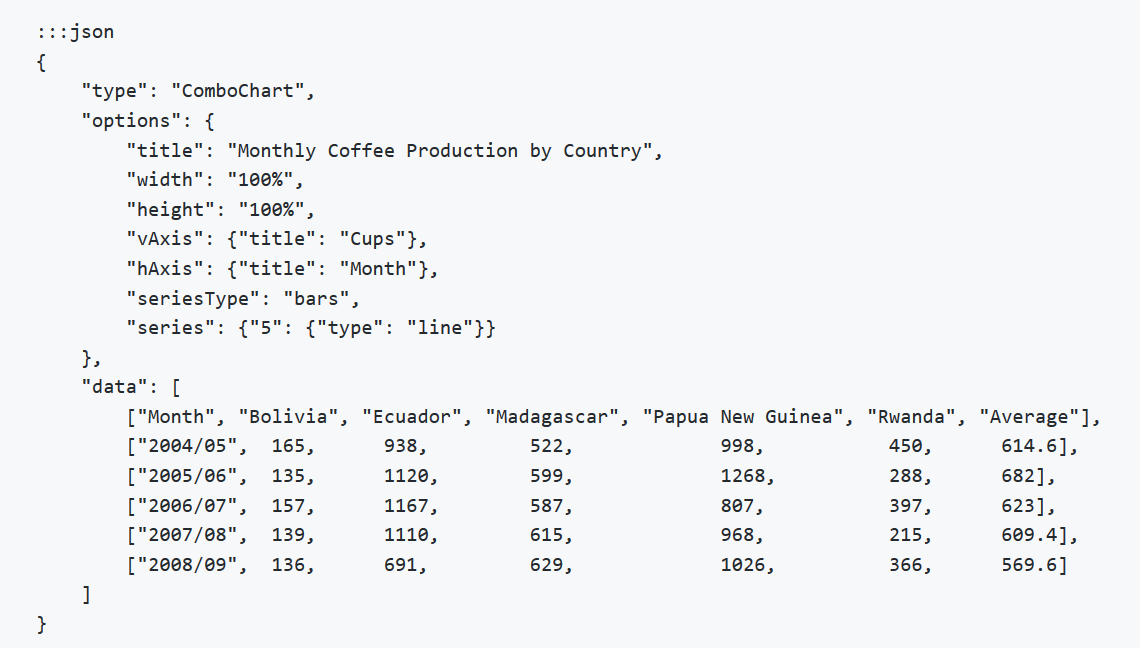


Abbildung : JSON Objekt für eine ComboChart Darstellung mit dem Google Chart Widget

Das Widget verfügt auch über einen Output Endpunkt genannt „Selected Data“. Welcher im Zuge des Projektes aber nicht getestet wurde. Informationen zum Output Endpunkt können am GitHub Respository gefunden werden.

Das beschriebene Widget wurde im Zuge des Projektes nur minimal durch um die Formatierung von Werten erweitert.

* 1. Custom Widgets und Operatoren

Neben den bereits vorhandenen Widgets und Operatoren wurden auch eine Vielzahl an eigenen Widgets und Operatoren erstellt um die Anforderungen des Projektes zu erfüllen. Als Grundlage dafür dienten in vielen Fällen die vorhandenen Widgets und Operatoren.

Im Folgenden werden die wichtigsten Widgets und Operatoren näher beschrieben. Grundlegende Informationen zu dem Widget können auch der userguide.md Datei im Doc Ordner des jeweiligen Widgets entnommen werden.

* + 1. Attribute Selector Widget

Das Attribute Selector Widget ist ein Widget, welches es ermöglicht ein Attribut aus in den Einstellungen definierten Liste auszuwählen. Das Widget wird auf der Input Seite mit dem NGSI Source Operator verbunden und auf der Output Seite mit dem STH Multiple Entities Source Operator, wie auf folgender Grafik dargestellt wird.

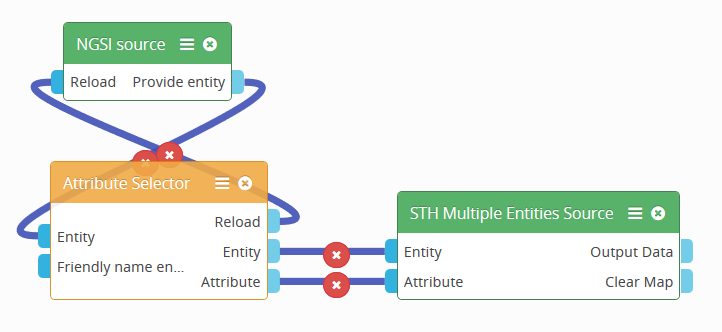


Abbildung 13: Attribute Selector Widget Wiring

Wie bereits erwähnt ist die Hauptaufgabe des Widgets das Auswählen von einem Attribut, dieses Attribut wird dann dem nächsten Operator als Input übergeben. Da die NGSI Source nur neue Werte an den Attribute Selector sendet, wenn Entitäten neue Werte erhalten und nicht wenn das Attribut im Attribut Selector geändert wird, wurde der Relaod Output Endpunkt hinzugefügt. Mit diesem Endpunkt wird ein Reload der NGSI Source ausgelöst. Erst wenn eine Entität an den Attribute Selector gesendet wird, wird das aktuell ausgewählte Attribut sowie auch die Entität an die Output Endpunkte für die weitere Verwendung gesendet.

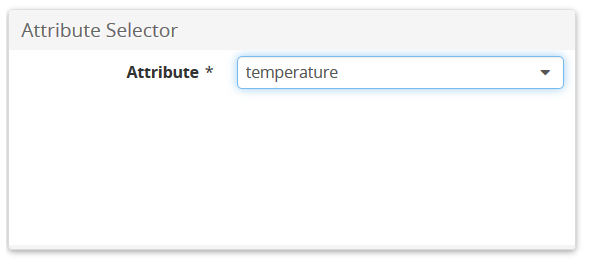


Abbildung : Attribute Selector Widget

* + 1. STH Multiple Entities Source Operator

Der STH Multiple Entities Source Operator ist ein WireCloud Operator, welcher es ermöglicht historische Daten von bestimmten Attributen von Entitäten mithilfe von Comet abzufragen. Der Operator basiert auf dem STH Source Operator und wurde an unsere Anforderungen angepasst.

Der Operator verfügt über die folgenden Input und Output Endpunkte:

* **Input** 
  + **Entity** 🡪 Für welche Entität die historischen Daten abgerufen werden.
  + **Attribute** 🡪 Für welches Attribut die historischen Daten abgerufen werden.
* **Output** 
  + **Output Data** 🡪 Ein JSON Objekt welches die abgefragten Werte, die dazugehörigen Timestamps, die Entität und das Attribut beinhaltet.
  + **Clear Map** 🡪 Ein String mit der Entity ID und einer Nachricht, dass keine Daten gefunden wurden. Kann verwendet werden um eine Karte zu leeren, vorausgesetzt das Karten Widget hat den dazugehörigen Input Endpunkt.

In den Einstellungen des Operators müssen die Server Einstellungen für Comet und ähnliches, sowie die Anzahl an Werten die abgefragt werden sollten eingetragen werden.

Der Grund dafür, dass die abgefragten Werte als JSON Objekt weitergegeben werden und nicht als extra Output Punkte liegt darin, dass die Daten gleichzeitig gesendet werden und fehlerfrei im nächsten Operator weiterverarbeitet werden können. Im Zuge des Projektes wurde als nächster Operator der STH Location Operator verwendet, wie auf folgender Abbildung dargestellt wird.

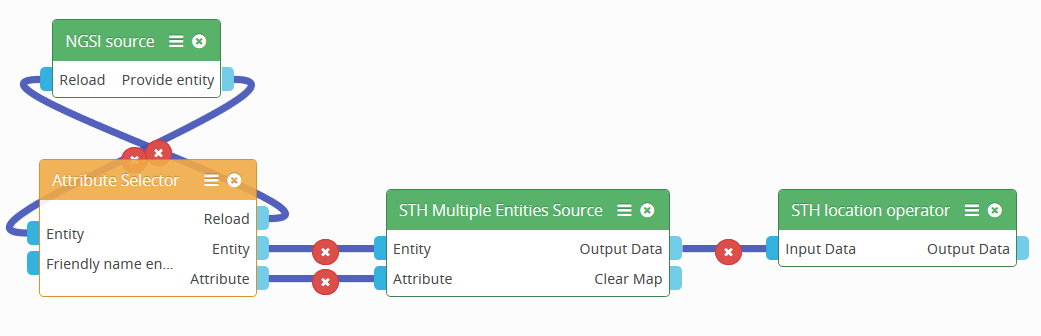


Abbildung 15: STH Multiple Entities Source Operator Wiring

* + 1. STH Location Operator

Aufgrund der Einschränkungen von Comet, dass historische Werte jeweils nur für ein Attribut abgefragt werden konnten und im Zuge des Projektes auch die Koordinaten für die Darstellung auf einer Heatmap notwendig waren, wurde der STH Location Operator entwickelt. Der STH Location Operator bekommt als Input die Daten des STH Multiple Entities Source Operator, macht eine Abfrage für die Koordinationswerte und fügt die Daten danach entsprechend zusammen.

In den Einstellungen kann der jeweilige Attributname für das Attribut, welches die Koordinaten beinhaltet eingegeben werden z. B. Location oder Position. Das Attribut sollte vom Typ geo:point sein.

**Input Daten (JSON Object):**

* Array von Werten (dataseries)
* Array von Timestamps zu den Werten (timestamps)
* Entität (entity)
* Attribut (attribute)

**Output Daten (JSON Object):**

* Array von Werten (dataseries)
* Array von Koordinaten von Werten (dataseriesLocation)
* Array von Timestamps zu den Werten (timestamps)
* Entität (entity)
* ID der Entität (entityId)
* Attribut (attribute)

Für die Weiterverarbeitung der Daten und damit diese auf einer Karte angezeigt werden können, wurde der STH Data to PoI Operator entwickelt, welcher mit dem STH Location Operator verbunden werden kann.

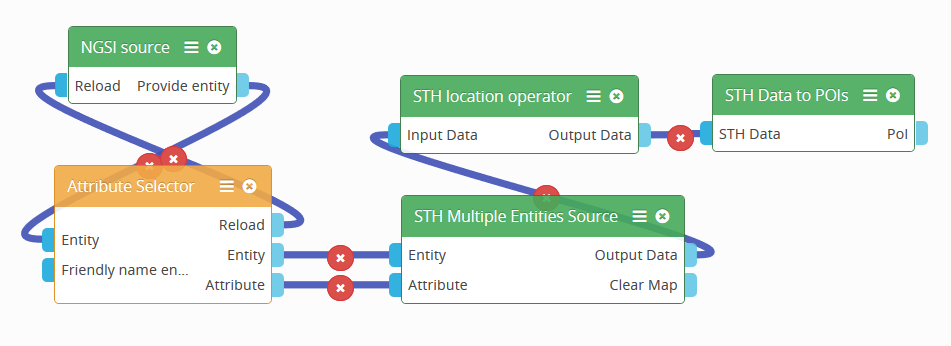


Abbildung : STH Location Operator Wiring

* + 1. STH Data to PoIs Operator

Ähnlich wie der bereits beschriebene Entity to PoI Operator, verhält sich auch der STH Data to PoI Operator. Dem Input Endpunkt wird ein JSON Objekt mit folgenden Attributen übergeben:

* Array von Werten (dataseries)
* Array von Koordinaten von Werten (dataseriesLocation)
* Array von Timestamps zu den Werten (timestamps)
* Entität (entity)
* ID der Entität (entityId)
* Attribut (attribute)

Für jeden Wert in dem Array dataseries wird daraufhin ein PoI erstellt. Ein PoI ist ebenfalls ein JSON Objekt bestehend aus folgenden Attributen:

* **Id (id)** 🡪 die ID der Entität
* **Tooltip Text (tooltip)** 🡪 die ID der Entität
* **Entität (data)** 🡪 Entität
* **Info Window (infoWindow)** 🡪 Informationen welche im Popup auf der Karte angezeigt werden
* **Aktuelle Position (currentLocation)** 🡪 Koordinaten für das PoI
* **Aktuellen Wert (currentValue)** 🡪 Wert des Attributes
* **Id für ein Set of PoIs (poiSetIdentifier)** 🡪 eine ID welche für das Array von Werten vergeben wird und einen Zusammenhang der PoIs für dieses Array schafft

Das PoI kann im nächsten Schritt dann auf einer Karte, im Fall des Projektes in Form einer Heatmap, dargestellt werden. Außerdem wurde im Zuge des Projektes ein PoI Beuatifier entwickelt.

* + 1. PoI Beautifier Operator

Der PoI Beautifier dient als Ergänzung für Operatoren, die aus verschiedenen Sensorinformationen von Entitäten PoIs für die Darstellung auf einer Karte erzeugen. In den Einstellungen des Operators kann festgelegt werden, welche Attribute der anzuzeigenden Entität im Popup-Fenster auf der Karte zu sehen sein sollen.

Zusätzlich können auch zusammengesetzte Attribute (wie zB. Adressen) entsprechend angepasst und Messwerte mit den dazugehörigen Einheiten versehen werden. Mithilfe eines „Friendly name entities“, das über den entsprechenden Input Endpunkt übergeben wird, können die im Info Window anzuzeigenden Attribute mit selbst gewählten Anzeigenamen dargestellt werden.

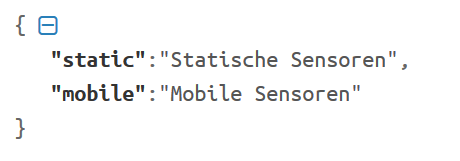
Soll der PoI für einen konkreten Messpunkt erzeugt werden, kann auch das Format des Zeitstempels eingestellt werden. Der Name des Messwertattributs wird dabei über den Input Endpunkt „Additional attributes“ übergeben.

* + 1. OpenLayers Map Widget

Das OpenLayers Map Widget basiert auf dem OpenLayer Map Widget von WireCloud und wurde für die Anforderungen des Projektes etwas bearbeitet und um eine Heatmap erweitert.

Das Widget besteht aus einem Basis Layer, der in unterschiedlichen Varianten zur Verfügung steht. Diese Basis Layer Optionen für von <https://basemap.at/> bezogen. In den Einstellungen des Widgets kann die initiale Position, der initiale Zoom Level, der minimale Zoom Level sowie auch Layer Optionen angegeben werden.

Bei den Layer Optionen wird ein JSON String im folgenden Format erwartet:



Das bedeutet, dass bei der Initialisierung des Widgets zwei leere Layer angelegt werden für die zwei Kategorien von Entitäten (static und mobile). Die PoIs werden danach je nach ihrem Typ dem jeweiligen Layer zugeordnet. Außerdem wird eine Checkbox für jeden Layer angelegt um den Layer auf der Karte danach ein oder auszublenden.

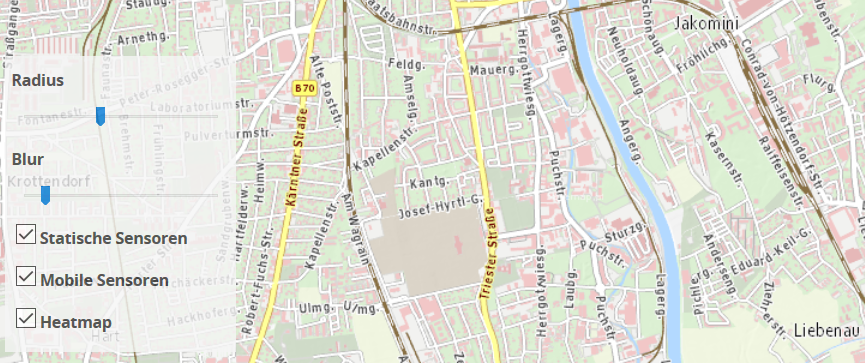


Abbildung : OpenLayers Map Widget Einstellungen

Wie man auf der Abbildung gut erkennen kann gibt es noch weitere Einstellungen, welche spezifisch für die Heatmap sind. Wenn der Input Endpunkt für die Heatmap PoIs verwendet wird, werden diese Einstellungen angezeigt. Auch für die Heatmap wird ein eigener Layer definiert, welcher je nach Belieben angezeigt oder nicht angezeigt wird. Die Slider für Radius und Blur sind ebenfalls für die Heatmap. Mithilfe von diesen Slider kann der Radius der Heatmap PoIs vergrößert/verkleinert werden sowie auch der Blur verändert werden.

Das OpenLayers Map Widget verfügt über die folgenden Input Endpunkte:

* **Insert/Update PoI** 🡪 Für die Darstellung von PoIs
* **Heatmap PoIs** 🡪 Für die Darstellung von historischen PoIs
* **Min Attribute Value** 🡪 Min Wert für die Berechnung der Gewichtung für Heatmap PoIs
* **Max Attribute Value** 🡪 Max Wert für die Berechnung der Gewichtung für Heatmap PoIs
* **Clear** 🡪 Zum Entfernen von Heatmap PoIs für eine bestimmte ID

Für die Bereitstellung der Werte für Min Attribute Value und Max Attribute Value kann der Min/Max Selector Operator verwendet werden.

* + 1. Min/Max Selector Operator

Der Min/Max Selector Operator ist ein Operator, welcher verwendet werden kann um Min/Max Werte an die OpenLayers Map weiterzugeben. Die Min/Max Werte werden von dem Map Widget zur Berechnung der Gewichtung für die Heatmap Punkte benötigt. Lauter der Dokumentation von OpenLayers für die Heatmap (<https://openlayers.org/en/latest/apidoc/ol.layer.Heatmap.html>) kann die Gewichtung von 0 bis 1 definiert werden. Die Berechnung der Gewichtung wird im Map Widget vorgenommen.

In den Einstellungen des Min/Max Selector Operator müssen die Min/Max Werte mithilfe von folgenden Schema definiert werden:

temperature=-20/40, humidity=0/100, pm2\_5=0/25, pm10=0/50

Mehrere Werte für unterschiedliche Attribute werden mit einem Beistrich getrennt. Eine Definition setzt sich aus dem Attributnamen, dem Minimum Wert und dem Maximum Wert zusammen. Wobei Minimum und Maximum mit einem Slash getrennt werden. Je nachdem welches Attribut dem Input Endpunkt übergeben wird, werden die jeweiligen Werte an die zwei Output Endpunkte gesendet. Das Attribut für den Input Endpunkt kann zum Beispiel vom Attribut Selektor kommen, wie auf folgender Abbildung dargestellt wird.

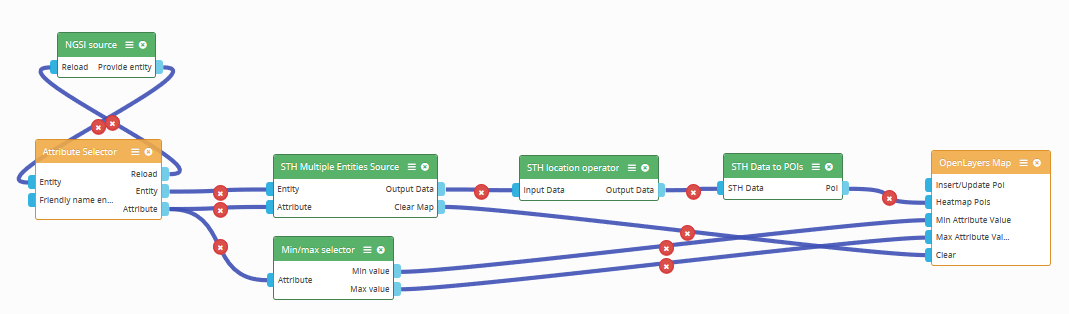


Abbildung : Min/Max Selector Wiring

* + 1. Heatmap Input Widget

Für das Abrufen von historischen Werten über einen bestimmten Zeitraum und der darauffolgenden Darstellung der abgerufenen Werte auf einer Karte in der Form einer Heatmap wird das Heatmap Input Widget benötigt. Das Widget besteht aus einem Formular mit folgenden Feldern:

* Attribute Name
* Date From
* Date To
* Max Values Per Entity

Des Weiteren beinhaltet das Widget einen Slider über welchen das Start- sowie auch das Enddatum definiert werden können. Der Zeitraum des Sliders ist eine Woche.

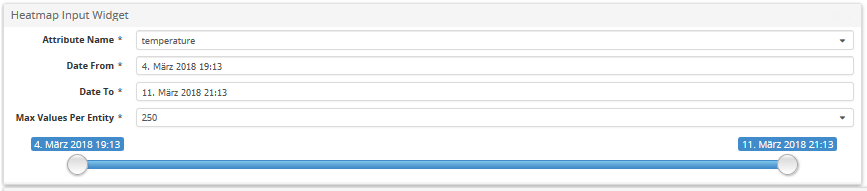


Abbildung : Heatmap Input Widget

Folgende Einstellungen müssen/können für das Widget gesetzt werden:

* NGSI Server 🡪 URL des Orion Context Brokers
* Tenant 🡪 Tenant für die Verbindung zum Context Broker
* Service Path 🡪 Service Pfad für die Verbindung zum Context Broker
* NGSI entity types 🡪 Liste von Entitätstypen, welche angezeigt werden sollten
* Attributes to display 🡪 Liste der Attribute, welche zur Auswahl stehen sollten

Anhand der eingestellten Filter für die Entitätstypen und die Attribute werden die jeweiligen zur Verfügung stehenden Attribute im Drop Down Feld angezeigt. Sobald ein Feld geändert wird werden die Werte aus den Formularfeldern an die jeweiligen Output Endpunkte gesendet. Das Widget verfügt über folgende Output Endpunkte:

* Entities 🡪 Alle Entitäten die den eingestellten Typ entsprechen
* Attribute 🡪 Das ausgewählte Attribut
* Date From 🡪 Das ausgewählte Startdatum
* Date To 🡪 Das ausgewählte Enddatum
* Max Values 🡪 Maximale Anzahl an Werten die abgefragt werden sollten

Für die Abfrage der historischen Werte wurde ein kompatibler Operator entwickelt, welcher über die entsprechenden Input Endpunkte verfügt, wie folgende Abbildung zeigt.

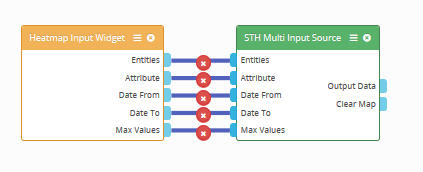


Abbildung : Heatmap Input Widget Wiring

* + 1. STH Multi Input Source Operator

Der STH Multi Input Source Operator wurde auf Basis des STH Source Operators entwickelt, wie auch schon die anderen STH Operators. Er verfügt über die passenden Input Endpunkte zu den Output Endpunkten des Heatmap Input Widgets und führt die Abfrage für die historischen Werte durch. In den Einstellungen kann die URL für den Comet sowie der Tenant und der Service Pfad gesetzt werden.

Da mehrere Entitäten vom Heatmap Inpute Widget gesendet werden, muss im Operator eine Abfrage für jede Entität vorgenommen werden. Nach jeder erfolgreichen Abfrage werden die Daten an den „Output Data“ Endpunkt gesendet. Werden keine Daten gefunden wird ein Signal an den „Clear Map“ Endpunkt gesendet. Dieser Endpunkt kann mit einer Map, mit dem entsprechenden Input Endpunkt, verbunden werden.

Bei den Output Daten handelt es sich gleich wie beim STH Multiple Entities Source Operator um ein JSON Objekt welches die abgefragten Werte, die dazugehörigen Timestamps, die Entität und das Attribut beinhaltet.

Der STH Multi Input Source Operator kann ebenfalls mit dem STH Location Operator und den darauffolgenden Operatoren verbunden werden. Dabei könnte sich folgendes Wiring ergeben:

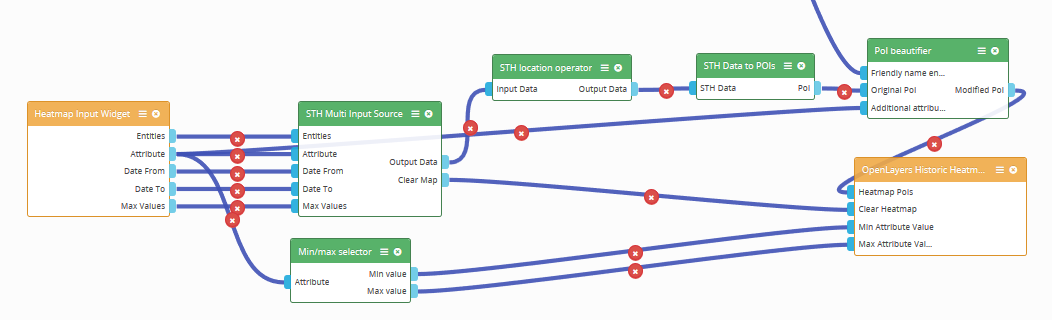


Abbildung : STH Multi Input Source Operator Wiring

* + 1. OpenLayers Historic Heatmap Widget

Alternativ zum OpenLayers Map Widget wurde das OpenLayers Historic Heatmap Widget entwickelt. Dabei handelt es sich um eine abgespeckte Version des OpenLayers Map Widget. Das bedeutet, dass Widget verfügt lediglich über die Heatmap Funktion zur Darstellung von historischen Daten. Ein weiterer Unterschied zum OpenLayers Map Widget ist, dass sobald ein Clear Signal beim „Clear Heatmap“ Input Endpunkt angelangt, alle Heatmap Punkte gelöscht werden und nicht nur die einer Entität.

* + 1. Chart Input Widget

Zur Abfrage von historischen Werten und der darauffolgenden Darstellung in einem Diagramm wurde das Chart Input Widget entwickelt. Dieses Widget besteht ähnlich wie das Heatmap Input Widget aus einem Formular mit den folgenden Feldern:

* Entity Id 🡪 Id der Entität für welche die Werte abgefragt werden
* Attribute Name 🡪 Der Name des Attributes für welches die Werte abgefragt werden
* Date Range 🡪 Zeitraum für welche die Werte abgefragt werden (auswählbar über einen DateTimePicker)

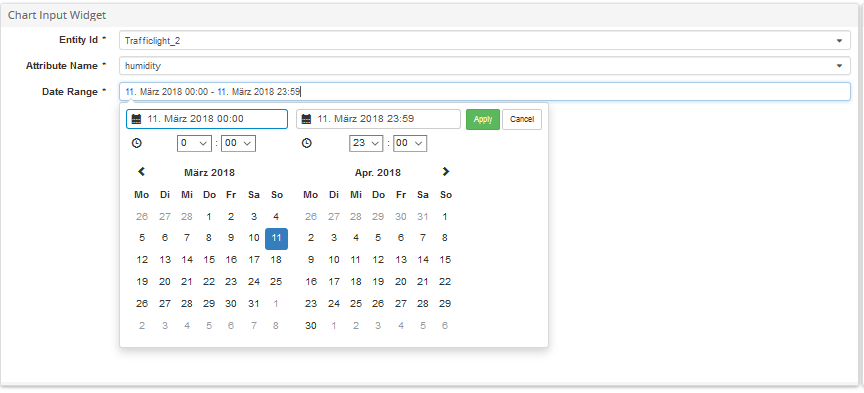


Abbildung : Chart Input Widget

Bei diesem Widget werden die Werte erst dann an den „Output Data“ Endpunkt gesendet, wenn der Accept Button geklickt wird. Bei den Output Daten handelt es sich wieder um ein JSON Objekt mit folgenden Attributen:

* entity 🡪 ID der Entität
* attribute 🡪 Ausgewähltes Attribut
* unit 🡪 Unit des Attributes (Leer, wenn in den Einstellungen nichts definiert wird)
* startDate 🡪 Ausgewähltes Startdatum
* endDate 🡪 Ausgewähltes Enddatum

Die Einstellungen sind ähnlich wie beim Heatmap Input Widget. Der einzige Unterschied liegt darin, dass beim Chart Input Widget noch die jeweiligen Attribute angegeben werden können, welche die Unit Werte enthalten. Dies geschieht mit dem selben Schema wie beim PoI Beuatifier Operator:

temperature=temp\_sensor\_unit,humidity=humid\_sensor\_unit,pm2\_5=air\_sensor\_unit,pm10=air\_sensor\_unit

Die Unit Werte werden für die Formatierung der Werte für das Diagramm benötigt. Das Widget verfügt auch über einen Input Endpunkt welcher „Message“ bezeichnet wird. Das bedeutet, dass eine Info- oder Fehlermeldung angezeigt werden kann. Bei der Message handelt es sich um ein JSON Objekt mit den Attributen:

* type 🡪 info oder error
* text 🡪 Die Meldung, welche angezeigt werden sollte

Für die Abfrage der historischen Werte wird natürlich wieder ein STH Operator benötigt, welcher mit dem Input Widget verbunden werden kann.

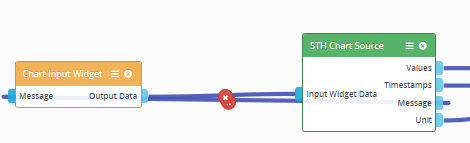


Abbildung : Chart Input Widget Wiring

* + 1. STH Source Chart Operator

Der STH Source Chart Operator basiert wie alle anderen STH Operators auch auf dem STH Source Operator von FIWARE. Der Input Endpunkt nimmt die Daten vom Chart Input Widget entgegen und macht danach die Abfrage an den Comet. Werden Daten gefunden, werden diese an die jeweiligen Output Endpunkte gesendet. Außerdem wird die Unit weitergesendet, das diese für die Formatierung benötigt wird und eine Message an das Chart Input Widget zurück gesendet. Werden keine Daten gefunden wird eine Fehlermeldung an das Chart Input Widget gesendet. Wie auch bereits bei den anderen STH Operatoren kann die URL für den Comet, der Tenant und der Service Pfad in den Einstellungen festgelegt werden.

* + 1. Google Chart Operator

Bevor die Daten an das Google Chart Widget gesendet werden können müssen diese noch in das richtige Format gebracht werden. Dies geschieht mithilfe des Google Chart Operators. Dieser nimmt die Daten von der STH Chart Source entgegen und fügt diese zu einem JSON Objekt zusammen, welches für das Google Chart Widget benötigt wird.

In den Einstellungen des Operators können folgende Konfigurationen vorgenommen werden:

* Chart Type 🡪 Auswahl des Diagrammtyps
* Chart Title 🡪 Titel des Diagramms
* X-Axis Label 🡪 Beschriftung der X-Achse
* Y-Axis Label 🡪 Beschriftung der Y-Achse
* Unit 🡪 Einheit für die Beschriftung
* Custom Chart Options 🡪 JSON Objekt mit benutzerdefinierten Werten (Optional), Informationen welche Einstellungen in den Optionen getroffen werden können in der Google Charts Doku gefunden werden.

Die aufbereiteten Daten werden danach an das Google Charts Widget von FIWARE gesendet. Daraus ergibt sich folgendes Wiring:

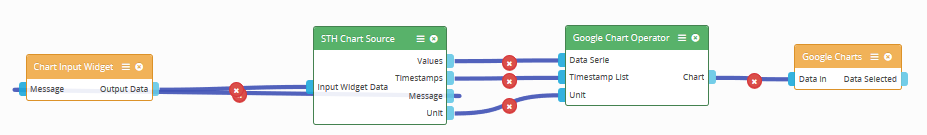


Abbildung : Google Chart Operator Wiring

* + 1. Gauge Chart Operator

Der Gauge Chart Operator dient dazu, ein bestimmtes Attribut einer Entität auszulesen und den entsprechenden Wert mit passenden Darstellungsoptionen an das Google Charts Widget weiterzuleiten. In den Einstellungen des Gauge Chart Operators kann neben dem darzustellenden Attribut auch die dazugehörige Einheit definiert werden. Darüber hinaus können noch die Darstellungsoptionen im JSON-Format übergeben werden, die Parameter wie Minimal- und Maximalwert oder die Farbgebung des Google Chart Widgets umfassen.

* + 1. IDAS & ORION REST Calls Operatoren

Die IDAS und ORION REST Calls Operatoren beinhalten die Logik für die Kommunikation mit dem IoT Agent bzw. dem Context broker Server. Diese Operatoren werden benötigt, um über Wirecloud Services, Entitäten, Devices und Subscriptions zu erstellen, zu bearbeiten oder zu löschen.

Für dieses Projekt kamen die beiden Operatoren ausschließlich in Verbindung mit den Service, Device, Entity und Subscription Manager Widgets zum Einsatz, wobei der jeweils benötigte Operator nicht ins Wiring integriert, sondern direkt aus dem entsprechenden Widget aufgerufen wird.

* + 1. Service/Entity/Device/Subscription Manager Widgets

Um über das Wirecloud-Portal auch alle relevanten Datenbankobjekte komfortabel verwalten zu können, wurden die Service, Entity, Device und Subscription Manager Widgets entwickelt. Diese vier Widgets präsentieren eine Auflistung sämtlicher Objekte des entsprechenden Typs inklusive der jeweils relevanten Eigenschaften. Für alle Entitäten werden so beispielsweise neben der Entity-ID auch der Entitätstyp und die enthaltenen Attribute angezeigt (zu sehen in der nachfolgenden Abbildung).

Über die zwei Buttons neben den jeweiligen Einträgen können die entsprechenden Objekte gelöscht oder bearbeitet werden. Bei der Bearbeitung eines bestehenden Objekts wird zuerst das in Kapitel 3.2.4 vorgestellte JSON Editor Widget gestartet, in dem sämtliche Objekteigenschaften im JSON-Format angezeigt und geändert werden können.

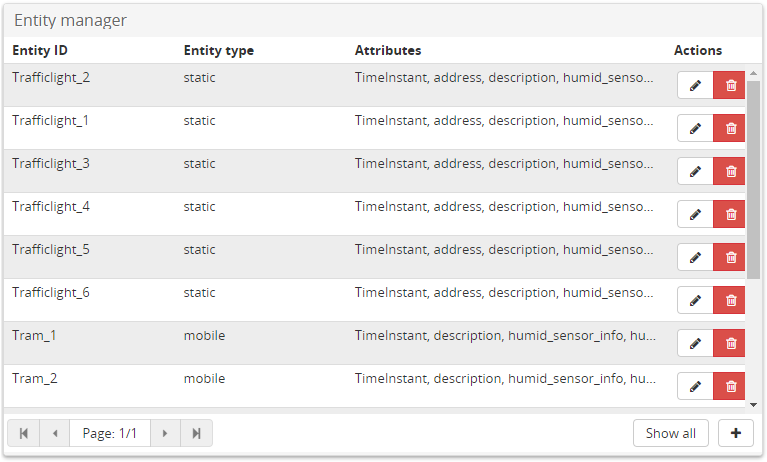


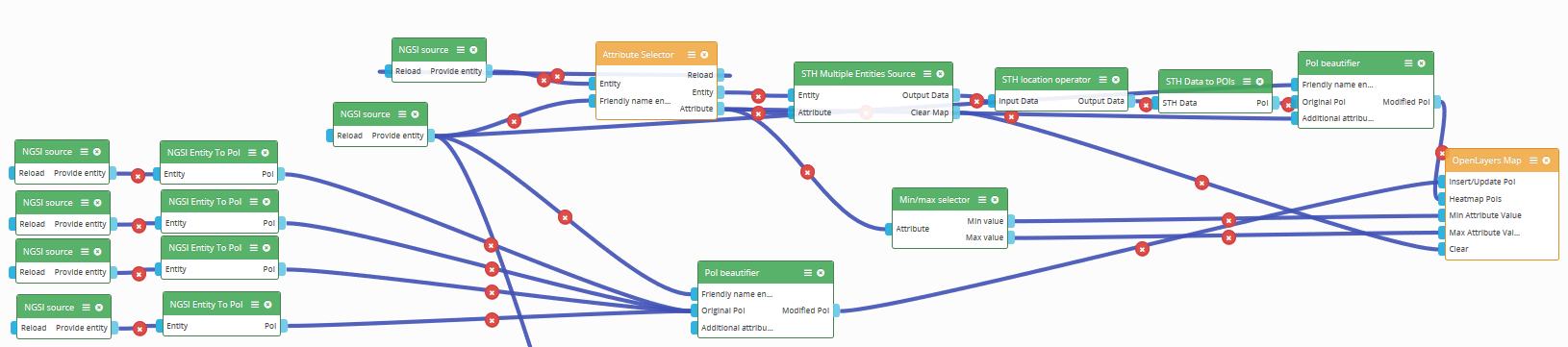
Abbildung : Entity Manager Widget

Mithilfe des „Plus“-Buttons im rechten unteren Bereich des Manager Widgets kann ein neues Objekt angelegt werden. Hierfür wird auch erst das JSON Editor Widget mit einer leeren Objektvorlage gestartet, welche in weiterer Folge befüllt werden kann.

In den Einstellungen des Manager Widgets können die Funktionen für das Hinzufügen, Bearbeiten und Löschen von Objekten je nach Bedarf separat aktiviert und deaktiviert werden. Weiters ist es möglich, die vier Manager Widgets im Wiring (in der Reihenfolge Service, Entity, Device und schließlich Subscription Manager) miteinander zu verbinden, wodurch die relevanten Parameter eines in der Liste ausgewählten Objekts (zB. Entity-ID und Entitätstyp bei Entitäten) automatisch als Filter an das nächste Manager Widget weitergeleitet werden. Hierfür muss in den Einstellungen zusätzlich die Option „Automatically forward filter information“ aktiviert sein. Sind die vier Manager Widgets nicht miteinander verknüpft, müssen Optionen wie „NGSI tenant/service“ und „Service path“ manuell gesetzt werden.

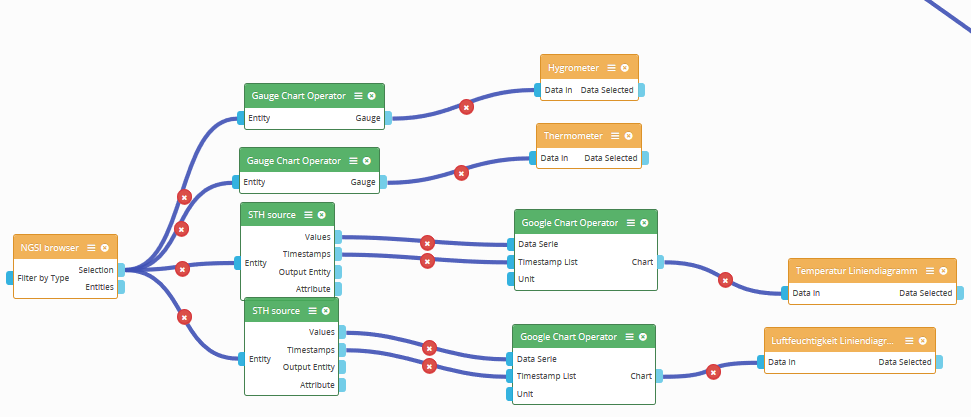
* 1. Use Cases
     1. Darstellung von Entitäten auf einer Karte



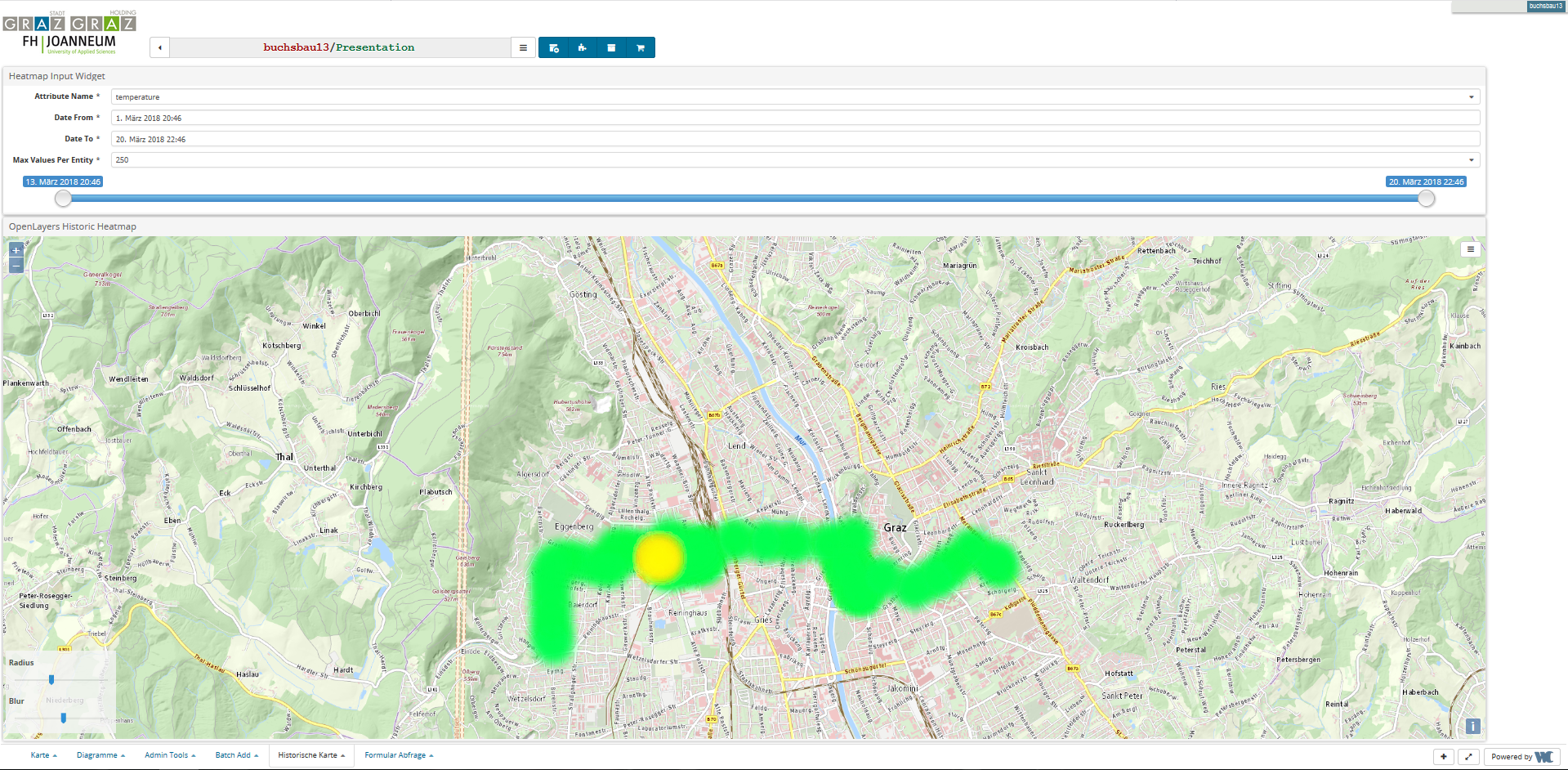


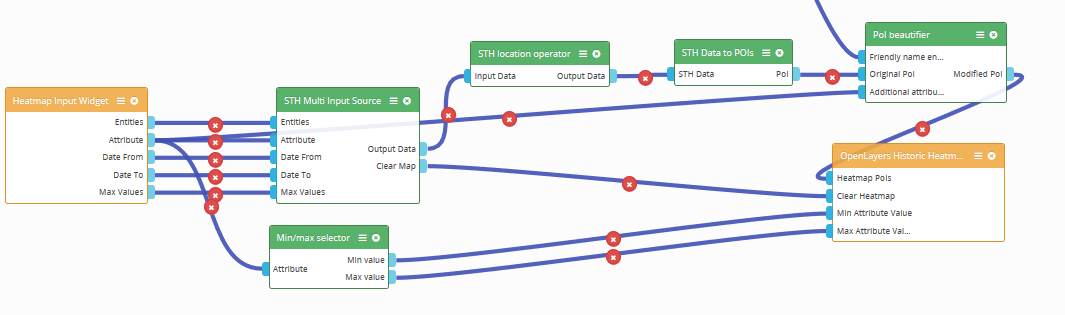
* + 1. Diagramme über die letzten Werte





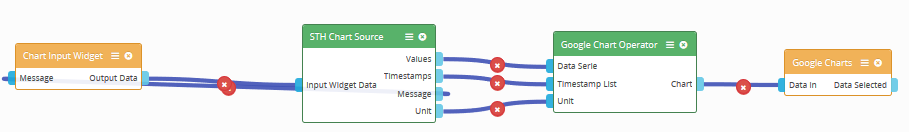
* + 1. Darstellung von historischen Werte mit einer Heatmap





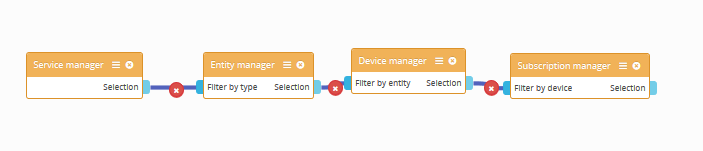
* + 1. Formular Abfrage von historischen Werten





* + 1. Administrations Tools





1. Security

Um die Sicherheit der FIWARE-Infrastruktur zu gewährleisten, werden von FIWARE selbst mehrere Softwarekomponenten bereitgestellt, mit denen sich die Kommunikation zwischen Sensoren und Server sowie zwischen den verschiedenen Serverkomponenten absichern lässt. Da der Sicherheitsaspekt der Plattform bei diesem Proof of Concept eher von geringer Bedeutung war, wurde die Implementierung der Sicherheitsfeatures in diesem Projekt ausgelassen.

Um allerdings zumindest ein Minimum an Sicherheit garantieren zu können, wurde das Wirecloud-Portal auf eine reine HTTPS-Verbindung umgestellt. Hierfür wurde zuerst eine kostenlose Subdomain über den FreeDNS-Dienst für die öffentliche IP-Adresse unserer FIWARE Lab-Instanz eingerichtet. Mit der neu angelegten Domain wurde nun mithilfe von Let’s Encrypt ein gültiges Zertifikat erstellt, das für die Absicherung der HTTPS-Verbindung eingesetzt wurde.

Um den Zertifikatserstellungsprozess möglichst einfach zu halten, wurde in die Docker-Konfiguration ein weiterer Container aufgenommen, in dem das Programm „certbot“ von Let’s Encrypt eingerichtet wurde. Da für das Wirecloud-Portal bereits eine Instanz des nginx-Webservers vorhanden war, brauchte nur dessen Konfiguration entsprechend für HTTPS angepasst zu werden. Die vollständige Einrichtungsprozedur kann unter <https://devsidestory.com/lets-encrypt-with-docker/> nachgelesen werden.

1. Raspberry Pi Sensoraufbau
2. Fazit
3. Linksammlung

|  |  |
| --- | --- |
| [1] | „WireCloud Dokumentation,“ [Online]. Available: https://wirecloud.readthedocs.io/en/latest/. |
| [2] | „Widget API,“ [Online]. Available: https://wirecloud.readthedocs.io/en/latest/widgetapi/widgetapi/. |
| [3] | „WireCloud Themes,“ [Online]. Available: https://wirecloud.readthedocs.io/en/latest/development/platform/themes/. |
| [4] | „Github WireCloud,“ [Online]. Available: https://github.com/Wirecloud/wirecloud. |
| [5] | „FIWARE Store,“ [Online]. Available: https://store.lab.fiware.org/#/offering. |
| [6] | „Github Wirecloud Widgets,“ [Online]. Available: https://github.com/wirecloud-fiware. |
| [7] | „Github NGSI Source,“ [Online]. Available: https://github.com/wirecloud-fiware/ngsi-source-operator. |
| [8] | „Github Entity2Poi,“ [Online]. Available: https://github.com/wirecloud-fiware/ngsi-datamodel2poi-operator. |
| [9] | „Github NGSI Browser,“ [Online]. Available: https://github.com/wirecloud-fiware/ngsi-browser-widget. |