
# IoT4CPS – Trustworthy IoT for CPS

FFG - ICT of the Future
Project No. 863129

# Deliverable D5.5.2
# Lifecycle Data Management Prototype II

The IoT4CPS Consortium:
AIT – Austrian Institute of Technology GmbH
AVL – AVL List GmbH
DUK – Donau-Universität Krems
IFAT – Infineon Technologies Austria AG
JKU – JK Universität Linz / Institute for Pervasive Computing
JR – Joanneum Research Forschungsgesellschaft mbH
NOKIA – Nokia Solutions and Networks Österreich GmbH
NXP – NXP Semiconductors Austria GmbH
SBA – SBA Research GmbH
SRFG – Salzburg Research Forschungsgesellschaft
SCCH – Software Competence Center Hagenberg GmbH
SAGÖ – Siemens AG Österreich
TTTech – TTTech Computertechnik AG
IAIK – TU Graz / Institute for Applied Information Processing and Communications
ITI – TU Graz / Institute for Technical Informatics
TUW – TU Wien / Institute of Computer Engineering
XNET – X-Net Services GmbH



*For more information on this document or the IoT4CPS project, please contact:*
Mario Drobics, AIT Austrian Institute of Technology, mario.drobics@ait.ac.at

## Document Control

**Title:** Lifecycle Data Management Prototype
**Type:** Public
**Editor(s):** Felix Strohmeier (SRFG)
**E-mail:** felix.strohmeier@salzburgresearch.at
**Author(s):** Felix Strohmeier (SRFG), Christoph Schranz (SRFG), Violeta Damjanovic-Behrendt (SRFG)
**Doc ID:** D5.5.2

## Amendment History

| Version | Date | Author | Description/Comments |
|---|---|---|---|
| V0.1 | 17.10.2019 | Felix Strohmeier | Initial version prepared |
| V0.2 | 20.11.2019 | Felix Strohmeier, Christoph Schranz | Draft for project-internal QA |
| V0.3 | 27.11.2019 | Felix Strohmeier | Minor updates in Appendix A |
| V0.4 | 02.12.2019 | Arndt Bonitz | Integrated Review Comments from AIT |
| V0.5 | 03.12.2019 | Heribert Vallant | Integrated Review Comments from JR |
| V1.0 | 12.12.2019 | Felix Strohmeier | Final Version for Publication |
| | | | |
| | | | |
| | | | |
| | | | |



The IoT4CPS project is partially funded by the "ICT of the Future" Program of the FFG and the BMVIT.

Federal Ministry
Republic of Austria
Transport, Innovation
and Technology

FFG
Forschung wirkt.

# Content

# Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| CPS | Cyber-Physical System |
| CRUD | Create, Read, Update, Delete |
| DNS | Domain Name System |
| GOST | Go-SensorThings |
| JSON | JavaScript Object Notation |
| OAuth2.0 | OAuth 2.0 Authorization Framework |
| RAMI4.0 | Reference Architecture Model Industrie 4.0 |
| REST | Representational State Transfer |
| SSL | Secure Socket Layer |
| TLS | Transport Layer Security |
| URI | Universal Resource Identifier |
| URL | Universal Resource Locator |
| UUID | Universally Unique Identifier |

## Executive Summary

This deliverable documents the developments of the Lifecycle Data Management Prototype II. It presents the second iteration of the Digital Twin Platform that was created to connect “loosely coupled” components (client applications) to share data with third parties, keeping stakeholder control over subsets of the data by the clients through customisation. This concept has been implemented by the creation of a data-streaming platform around the scalable open-source framework “Apache Kafka”, in which each of the “data producer” to “data consumer” relation is defined by a separate communication “topic”. Kafka Streaming Applications, which can be configured with additional filter functions, connect publishers and subscribers with each other to exchange the contractually agreed data streams.

The source code of the prototype is released under a permissive open source license and can be found on the project-internal GitLab instance (https://git-service.ait.ac.at/im-IoT4CPS/WP5-lifecycle-mgmt). To provide any reader of this public deliverable access to the open source code, a fork on GitHub (https://github.com/iot-salzburg/panta_rhei) has been created that is public and updated regularly.

# 1. Introduction

This report contains the second iteration of a Digital Twin Platform prototype developed in the IoT4CPS project (as part of Work Package WP5 "IoT Lifecycle Management", within Task 5.5). Being the successor of public deliverable D5.5.1, published in July 2019, it contains the advances achieved during the Implementation period from July 2019 to October 2019.

The goal of the developed Digital Twin concept and prototype is to support scenarios, where several data suppliers can exchange their live data without additionally storing it in a central data collection point. At the destination, the data subsets will then again be set into the local context in order to be interpreted correctly. In this part of the prototype implementation, we concentrate on the aspect of sharing data streams and meta-data information between CPSs across boundaries of companies and administrative domains. This is especially important in all use cases of a multi-tenant Digital Twin Platform, where multiple systems of multiple companies are involved along the value chain. In the final prototype application, we want to demonstrate the life cycle data management of automotive components in use, with stakeholder control over subsets of the data and – through customisation – keeping compliance with different regulations regarding privacy and third-party usage of data. Therefore, we created a simple data model for meta-information about stakeholders including a management UI to support the multi-tenant access on sensor data streams.

The report on the final prototype (D5.5.3), including data analytics aspects will follow in June 2020.

# 2. Architecture Overview of the Prototype

In this section, we describe the layered architecture of our Digital Twin Platform prototype. An architectural overview is shown in Figure 1, which distinguishes between two types of clients of the platform (upper layer), the application run on a CPS and the end-user interacting with the system for administrative tasks. While the first one usually directly runs on an embedded system (e.g. within the connected car), the latter one enables any interaction through a user interface device, such as a web browser or a mobile application. The following sections describe the single layers of the architecture, from the bottom to the top.

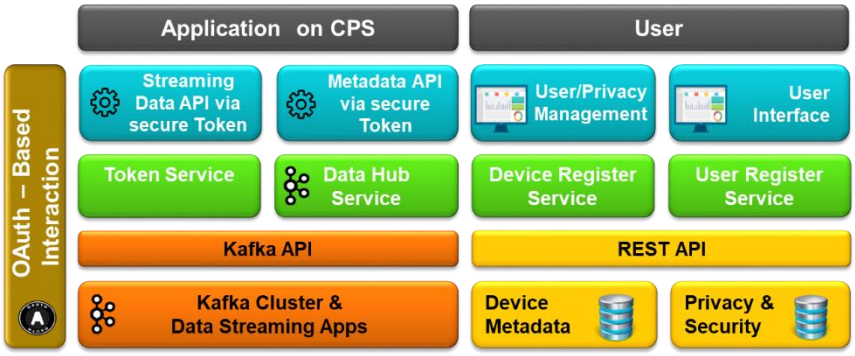


**Figure 1 – High-level Component Architecture**

## 2.1 Digital Twin Messaging and Data layer

### 2.1.1 Streaming Platform with Data Stream Apps (Apache Kafka)

Core functionalities required in Digital Twin Platforms are scalable data streaming and complex event processing, which has been implemented using Apache Kafka. In the demo setup, Kafka just runs on a single node. In production environments, however, Apache Kafka can and should be scaled out and distributed among a cluster of nodes for both performance and fault-tolerance reasons. Beneath the data streaming itself, Kafka also allows the creation of "Data Streaming Applications" using Kafka Streams[1], which can subscribe to

[1] Kafka Streams: https://kafka.apache.org/documentation/streams/

various source data streams, filter, process, or analyse them and return altered data streams back to the Kafka cluster.

### 2.1.2 Device Metadata (SensorThings)

Sensors usually have specific metadata, such as the type of observation, the observation property, unit of measure or any other description of the sensor devices (things) itself. Instead of conveying that data in each and every data packet delivered for a measurement, this information is managed using an external SensorThings server. A SensorThings server provides a SensorThings API[2] as defined by the Open Geospatial Consortium (OGC). For the prototype implementation we use a GOST SensorThings server[3], running inside three Docker[4] containers (one for the database, one for the service API and one for the dashboard).

### 2.1.3 Digital Twin Platform Identity Data Model

For managing the basic data within the Digital Twin Platform prototype it requires a simple data model for creating relations between the single entities. In the data model we define users, companies, clients, streams and systems. “Systems” is a general term that we use here for grouping single CPSs and service applications that serve for a specific purpose, e.g., a weather service including weather stations. For identification and structuring of multiple systems, we propose to use a hierarchical approach according to the RAMI4.0 reference model, which defines “workcenters” and “stations” below each organisation (or company). In our prototype, this substructure model is composed of simple strings using the dot-notation know from DNS. A system, which is owned by a single company, can have multiple clients and multiple data streams. According to the model, a data stream connects exactly one source to one target system. However, using the data stream implementation as described in more detail in section 3, flexible many-to-many communication streams are possible.

In this prototype, for simplicity a local PostgreSQL[5] database was used. A production-grade system can also include more advanced user and identity management, such as OAuth2.0[6]-based authorization servers. The physical data model is shown below (Figure 2).

[2] SensorThings: https://github.com/opengeospatial/sensorthings

[3] GOST (Go-SensorThings) is an IoT Platform written in Golang (Go): https://github.com/gost/server

[4] GOST: https://www.gostserver.xyz/tutorials-installation-docker/

[5] PostgreSQL: https://www.postgresql.org/

[6] The OAuth 2.0 Authorization Framework, IETF RFC6749, RFC8252

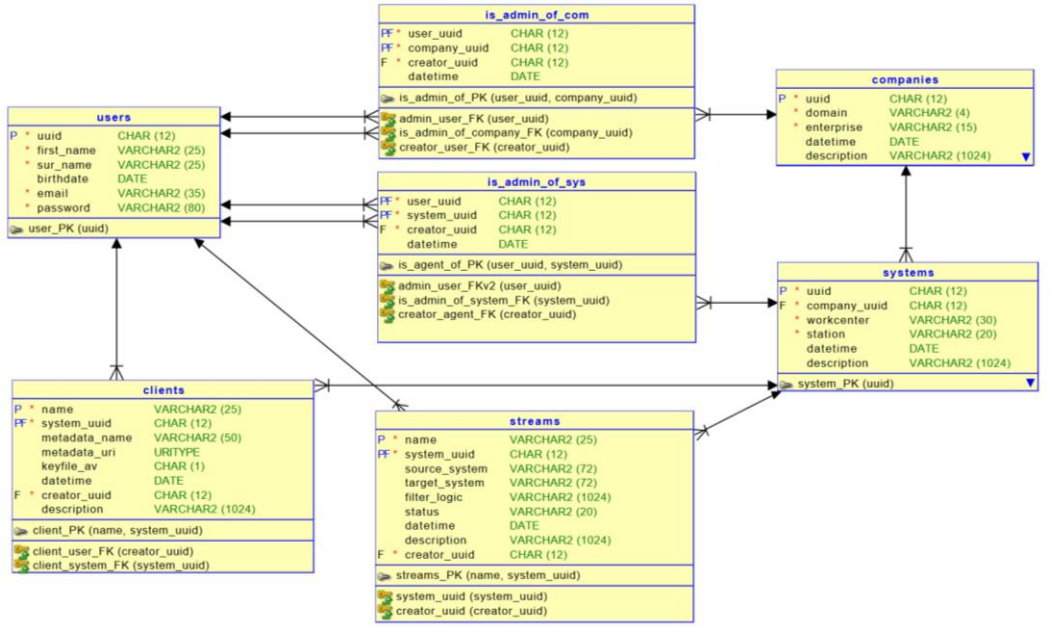


**Figure 2 – System Data Model**

Note that in our data model we use a n:m connection between companies and users, instead of the usual 1:n relationship. This enables the users to manage multiple companies, which is useful in cases where e.g. an IT-Service company should manage data streams for multiple other companies without any IT personnel available. The same is true for the relation between users and systems, i.e., one user can manage multiple systems and one system can be managed by different users.

## 2.2 Digital Twin Service layer

The service layer provides controlled access to the database and implements the standards CRUD operations on companies, users, systems, clients and streams. In the prototype, the service layer is implemented in Python and Flask[7].

## 2.3 Application and User Interface layer

As already mentioned, the Digital Twin Platform provides separate interfaces for the users and applications running on CPSs. The user interface provides simple management functionality (list, add, show, delete) for companies, systems, data streams and the users itself (including registration, login). The API for the CPS is implemented in Apache Kafka, clients can either use the Kafka REST API, or directly implement a Kafka consumer and / or producer. Sample consuming and producing client applications implemented in Python are provided together with the platform.

# 3. Demo Use Case

In this section, we describe the main flow of events using screenshots of the prototype according to a demo use case about connected cars. In particular, the use case involves connected cars of a car rental company located in Iceland, where cars are enabled to exchange weather and temperature information with each other and with a central weather service. Before we discuss on the use of the platform in detail, we describe the data flow in the platform shown in Figure 3.

[7] https://palletsprojects.com/p/flask/, https://github.com/pallets/flask

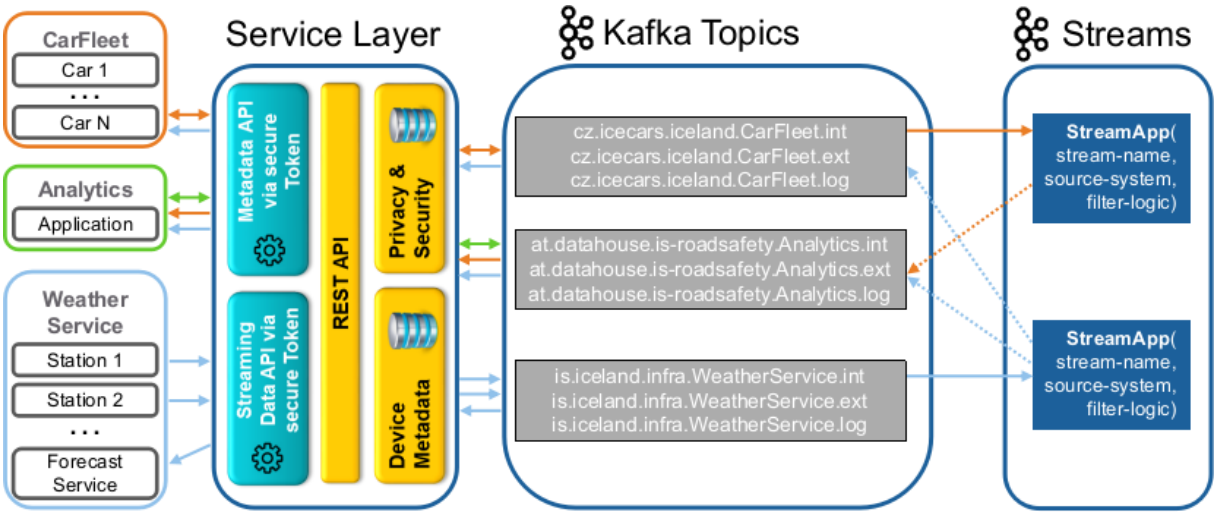


**Figure 3 – Data Streaming between CPSs**

On the left side, the client applications are shown, which are implemented directly in the cars, in some analytics backend or at a weather service provider. These applications can send (produce) single data objects to or receive (consume) them from the streaming platform via the service layer. The service layer handles access control and additionally provides the metadata for the single data objects on request. This can be for example meta-data information on the sensor, such as the unit of measurement or accuracy of the measured values.
For each system, three separate Kafka topics with a name and three different suffixes are created (“.int”, “.log” and “.ext”). Once the service layer has been passed, each stream is published to its own topic name using either the internal (“.int”) or the logging (“.log”) suffix. Internal topics are used for the communication of applications and CPSs within a single system. Furthermore, a streaming app can subscribe on an internal topic, combine several data streams to a new one and implement filter rules, e.g. to only get warnings if some predefined thresholds are passed. In contrast to communication between clients within a single system, the resulting stream of a stream application is finally forwarded to an external topic of another target system. Therefore, only stream apps are permitted to publish on external (“.ext”) topics. This means the “ext”-topics of one system store data from other (external) systems rather than their own data.
The resulting stream will be sent back through the service layer and can be consumed by any authorised consumer client application that subscribe to that data stream.

## 3.1 Welcome screen and user registration

After the service has been started, some users need to be registered to be able to use the platform prototype itself. As shown in Figure 4, the welcome screen contains the basic steps required to start a data stream, an exemplary illustration of the data flows and the link to its source code repository.


**Figure 4 – Welcome Screen and Introduction**

From the welcome screen, the user can use the “register” button for self-registration to the system as shown in Figure 5. If the user is already registered, he or she can directly navigate to the login.

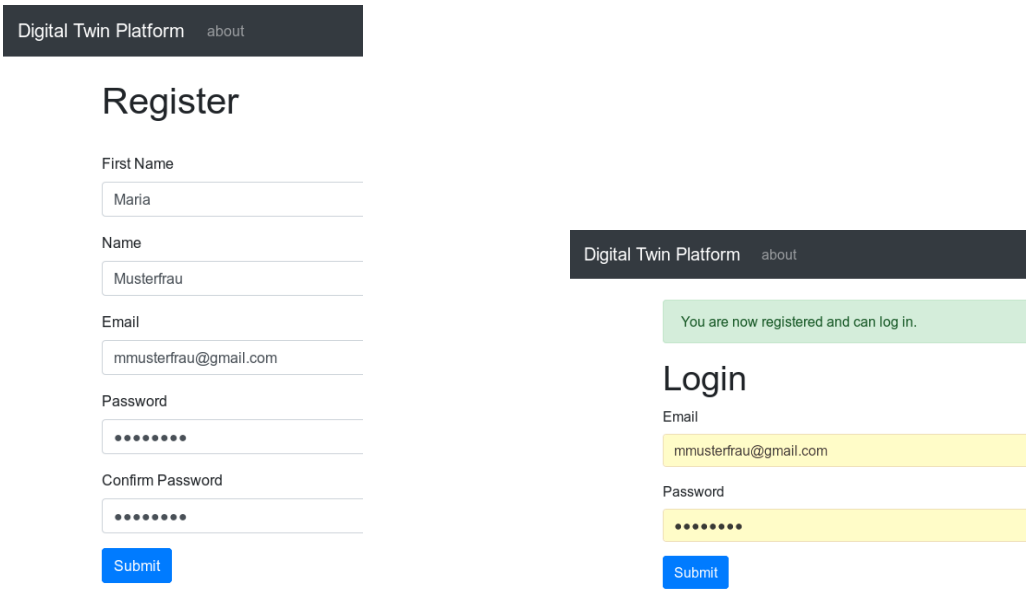


**Figure 5 – Self-Registration and Platform Login**

Once the user is registered and logged in, the user will see an empty dashboard (Figure 6), listing companies, systems, client applications and data streams accessible by the current user. The user has now the option to add new instances, which is described in more detail in the next section.

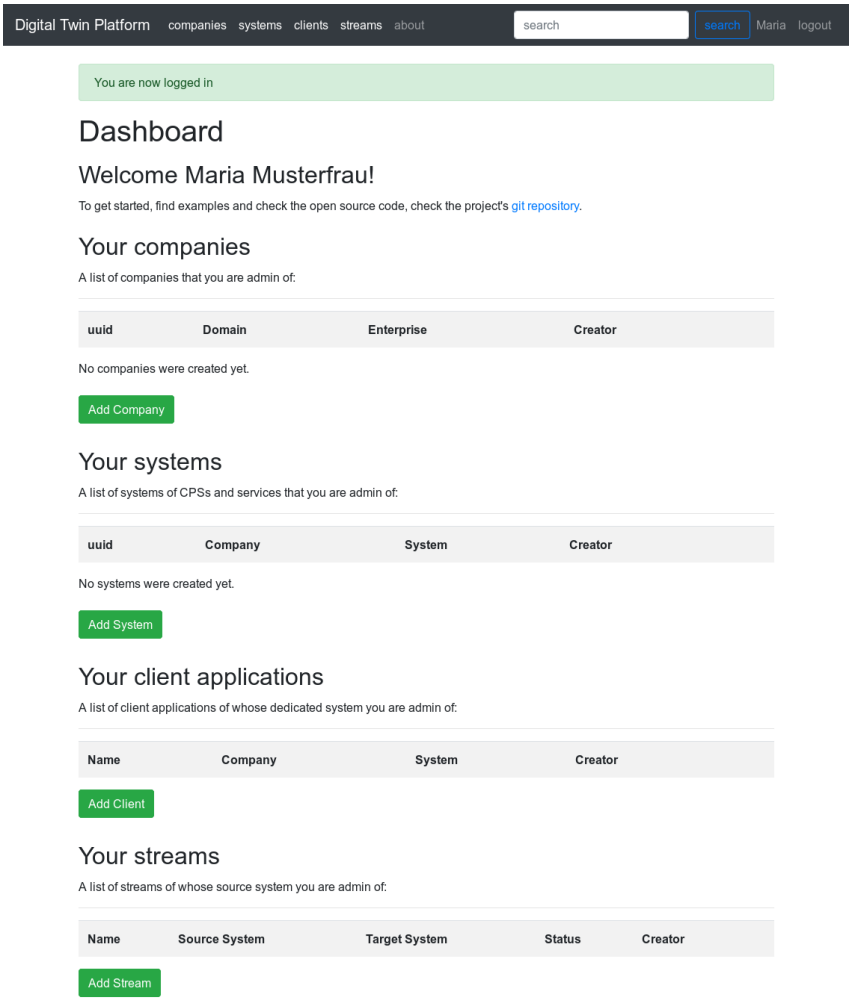


**Figure 6 – Initial, empty Dashboard**

## 3.2 Company and Systems Management

In this section we show, how to manage companies and systems. In the presented use case the company, users and systems for the weather service is presented.
For the full demo scenario, a second company for the car rental service (car fleet) needs to be created, including corresponding users and systems in order to exchange datastreams between multiple systems.

### 3.2.1 Companies

The first step is to register a new company, which will be the owner of the cyber-physical systems created later. As shown in the screenshot in Figure 7 below, for demonstration purposes, a company is simply identified by a top-level domain, a short name and an optional description.

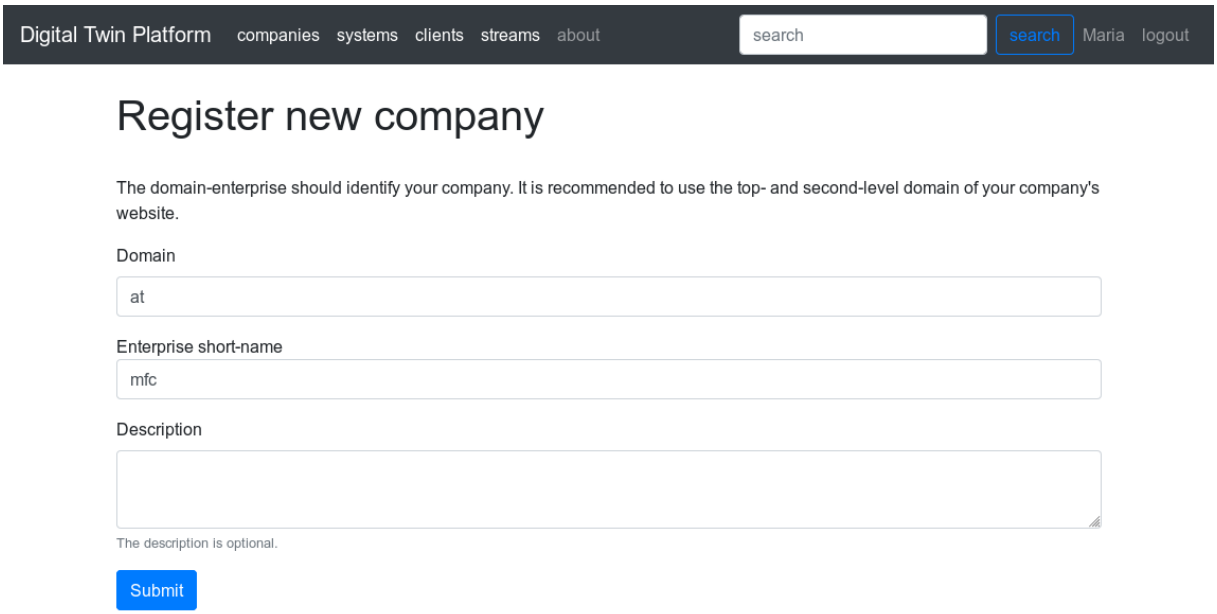


**Figure 7 – Register / create companies**

A list of all companies that can be managed by the current user is shown in Figure 8, which currently contains exactly the just registered company. The blue button “manage company” leads to the next screen (Figure 9), in which other company admins and systems can be added.

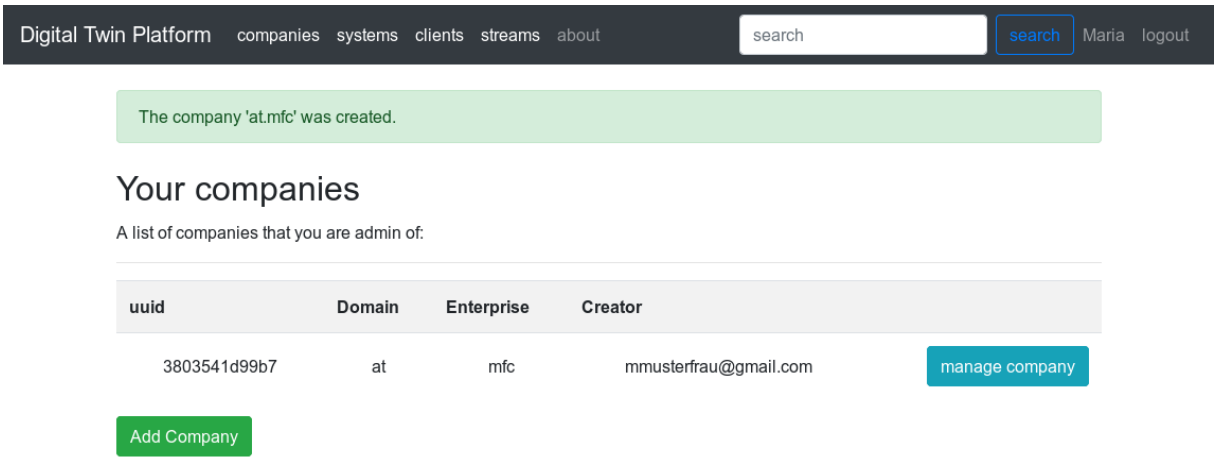


**Figure 8 – List companies**

A company admin has the permission to create and delete systems as well as other company admins. Admins can also delete the selected company.

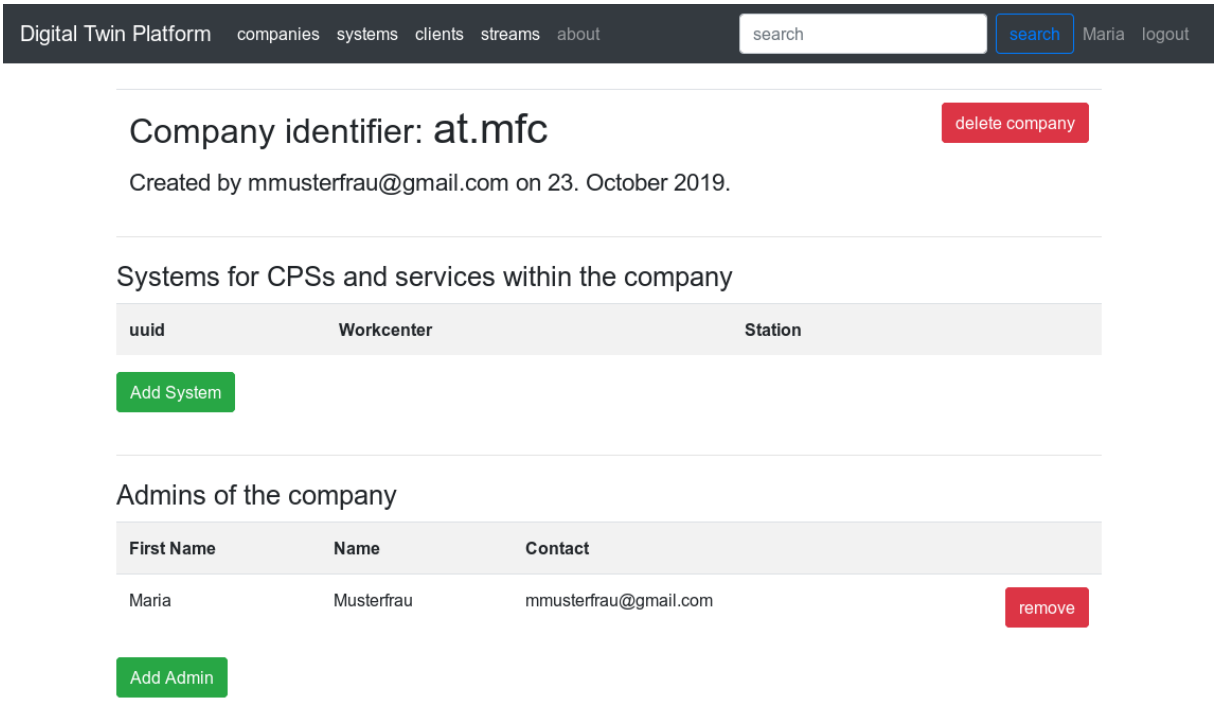


**Figure 9 – Show companies**

Systems for CPSs and services within the company can be added using the green “Add System” button. How systems can be defined is described in the next section.

### 3.2.2 Systems

In our context, a system is a logical entity that groups together multiple applications and CPSs, which serve a common purpose. The user can provide its own system identifiers using a unique combination of a workcenter short-name and station name within the related company. Still, each system will get universally unique identifier (UUID) for globally unique identification. The notations of workcenter and station are taken from the RAMI 4.0 reference model. They allow the creation of a logical hierarchical structure within a company (or enterprise). Having that, system instances can again be assigned and grouped into such structure. The screen on how to add new systems to the company on the platform is shown in Figure 10 with the example of the weather service.

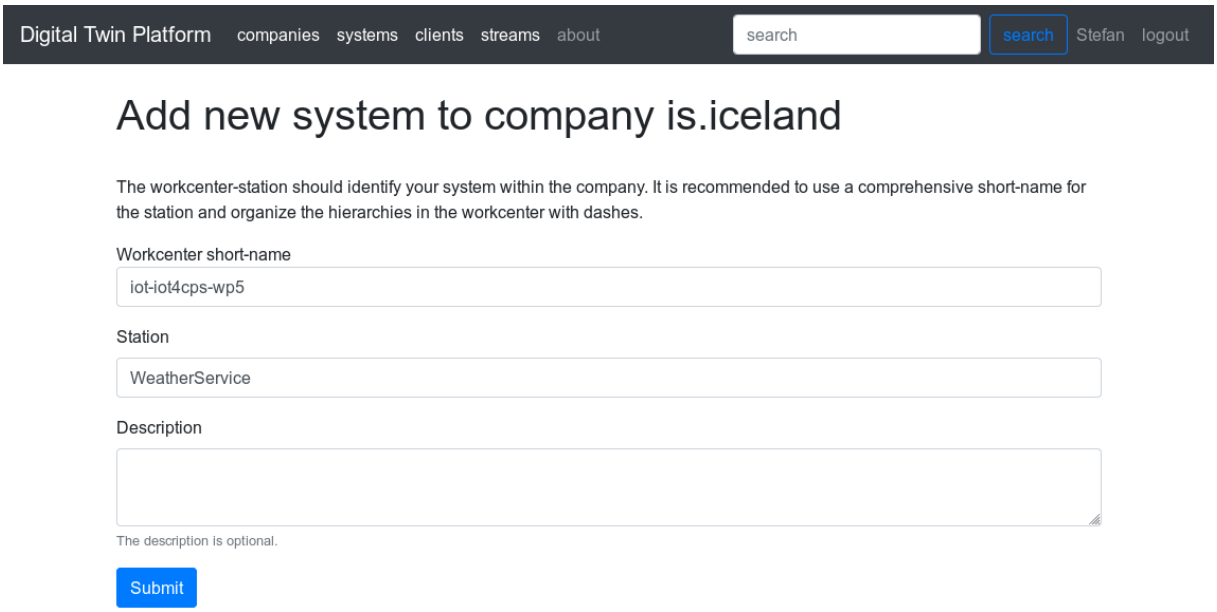


**Figure 10 – Add systems to companies**

Once a system is created, it has several options that need to be further defined. The overview of a newly created system is shown in Figure 11, which can contain client applications, stream applications and system administrators. Similar to companies, systems also have dedicated administrators. System administrators have the permission to register and manage client applications and streaming applications that are assigned to this system. One company administrator can assign multiple system administrators for managing their systems.
Client applications can be created by using the “Add Client” button and are used to produce data to and consume data from the Digital Twin Platform. They are described in more detail in the next section.
Streaming applications can be created by using the “Add Stream” button and are used to connect the selected system to another system for data exchange. While the selected system is the source system, the other system will be shown as target system.
Additional system administrators can be invited or assigned by using the “Add Admin” button.
All added clients, streams, admins or even the whole system can be deleted using the corresponding “delete system” buttons.

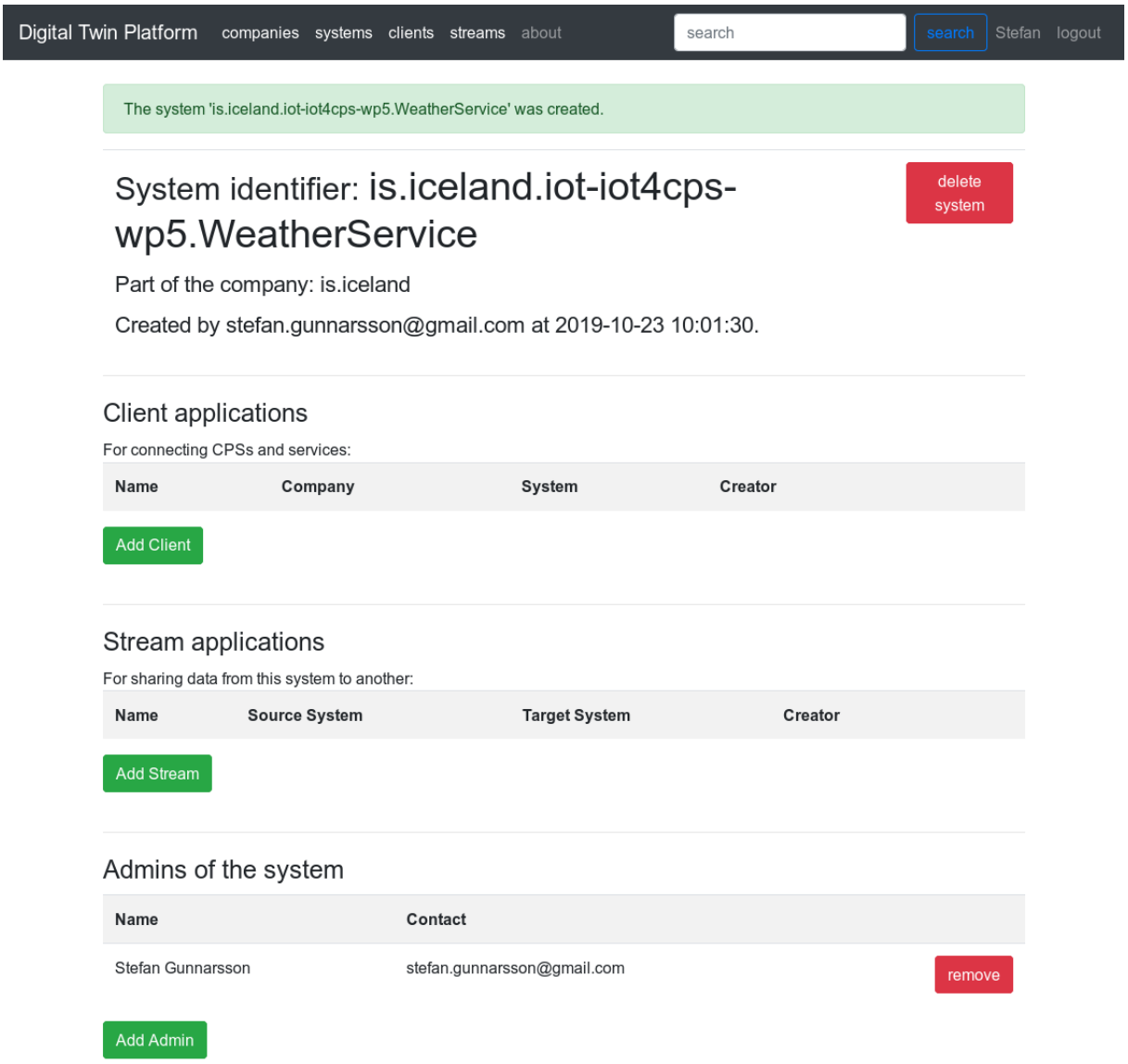


**Figure 11 – Show system details (newly registered system)**

## 3.3 Client Applications

A client application denotes a piece of software, deployed as part of a service or connected device (“thing”), which is intended to communicate with other services or devices. In the use case example, this would be a connected car, which provides its own measured temperatures including spatiotemporal information to other cars. In the same time, it may receive external temperature data from other cars or weather stations for the next few hundred meters along the driving path to increase the safety of the driver.

Before the client application can send or receive data, it needs to be registered within a previously defined “system”. This can be done using the “Add Client” button. The registration screen for the clients is shown in Figure 12 for the system “is.iceland.iot-iot4cps-wp5.WeatherService”.

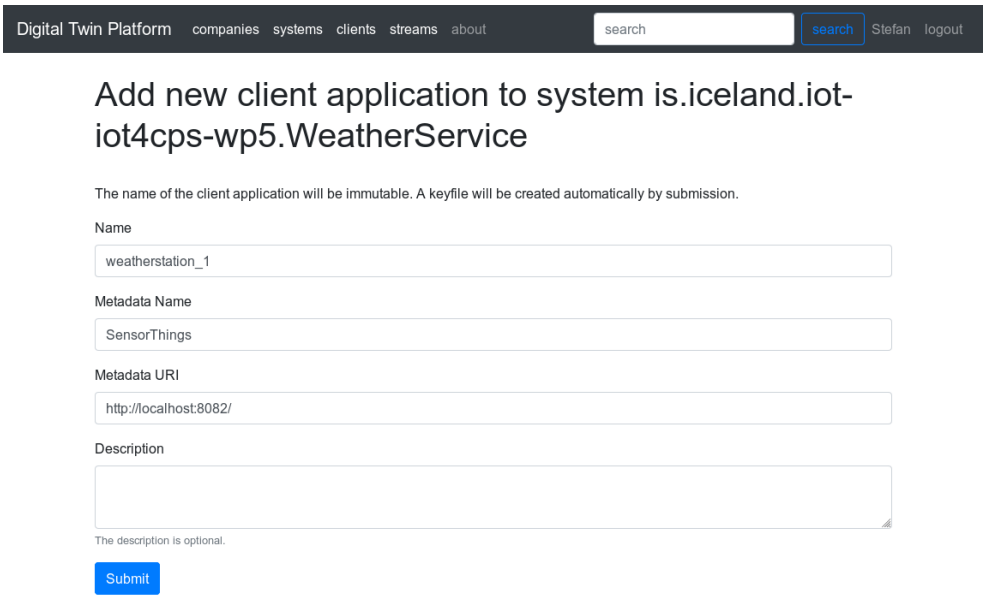


**Figure 12 – Register client applications within systems**

When registering such a client application, its name must match with the unique dedicated system entry in the Digital Twin Platform. Moreover, a name and URI for metadata description has to be provided, as shown in the configuration of the client in Figure 13 below. In our example, we create a client application for a single weather station (“weatherstation_1”). Note that in a production scenario a high number of clients can be added programmatically by directly using the REST-API.

Once a client is created, the screen from Figure 13 will be shown to the user. In addition to the data entered above, two important information elements are presented. First, a short JSON configuration data structure that can be used to create the client application itself by copy & paste. Second, if a client application is registered, a SSL-key will be generated that can also be used by the application. However, the usage of this key by the client application is not implemented in the current examples but will be added to the third prototype implementation (D5.5.3).

All access for clients can be revoked by deregistration of the client in the Digital Twin Platform. This can be achieved by using the “delete client” button.

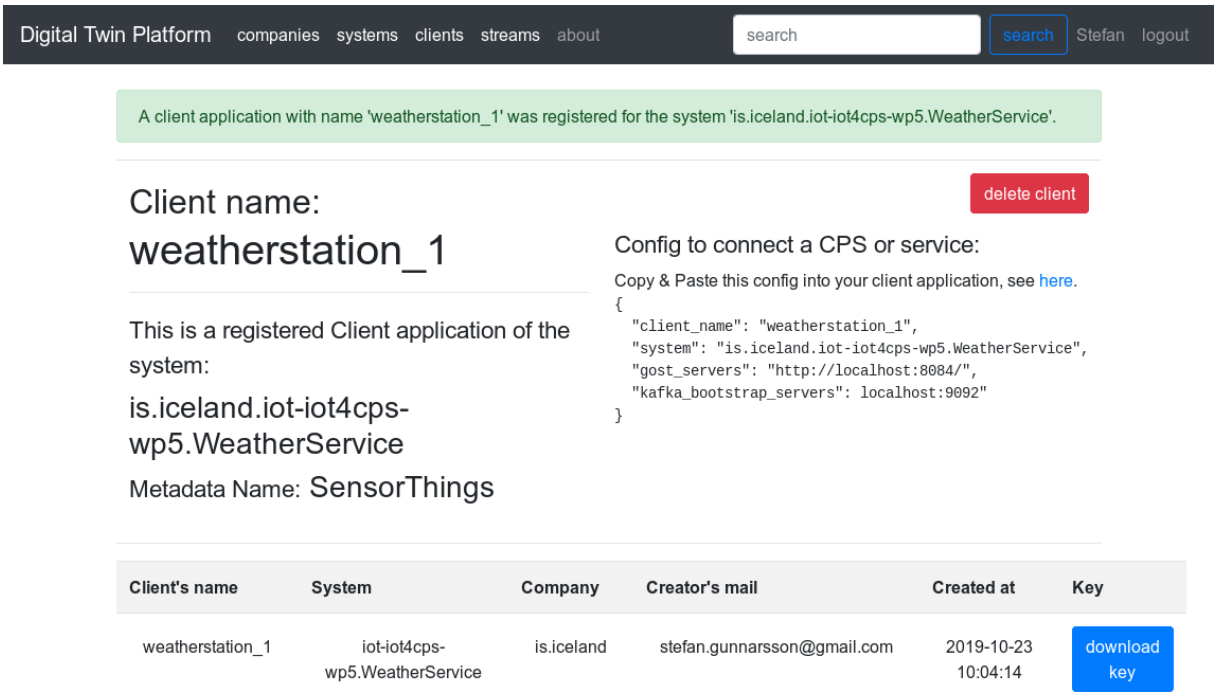


**Figure 13 – Manage client applications within systems**

For the simple creation of an example producer or consumer application in Python, a "DigitalTwinClient" class is provided in the source code repository, where only the shown config object (Figure 14) needs to be passed as the only constructor parameter.

```
config = {"client_name": "car_1",
          "system": "cz.icecars.iot-iot4cps-wp5.CarFleet",
          "gost_servers": "localhost:8084",
          "kafka_bootstrap_servers": "localhost:9092"}
```

**Figure 14 – Client Configuration**

Beneath client_name and system id, the client needs a kafka_bootstrap_server to connect as publisher or subscriber, and an address to a gost_server (SensorThings), which provides the meta-data information for the sensor data flow (as described in Section 2.1.2). A whole example source code for such client application, which shows how to produce and consume data to the platform, is also listed in Appendix B: Client Applications.

## 3.4 Streaming Applications

A streaming application enables the communication between a source and a target system. Once deployed, it subscribes the internal topic of the specified source system, combines several data streams to a new one and implement filter rules, e.g. to only get warnings if some predefined thresholds are passed. The resulting stream is then forwarded to the external topic of the target system. Hereby, these external topics are used to receive data from, and only from, streaming applications.

In order to create a new stream application within a given source system, a unique name for the stream and a target system is mandatory. In our example, the source system is the weather service and the target system will be the whole car fleet of the rental company. Therefore, the stream is simply named "weather2cars".

This stream will forward data from the source system to the target system. Additionally, a filter logic can be defined, which can be considered as a description language for selecting and filtering time-series datastreams.

The default value is an empty clause like shown in Figure 15, what implies that any data from the source system is forwarded to the defined target system. Within Prototype II, only this trivial default case is implemented.

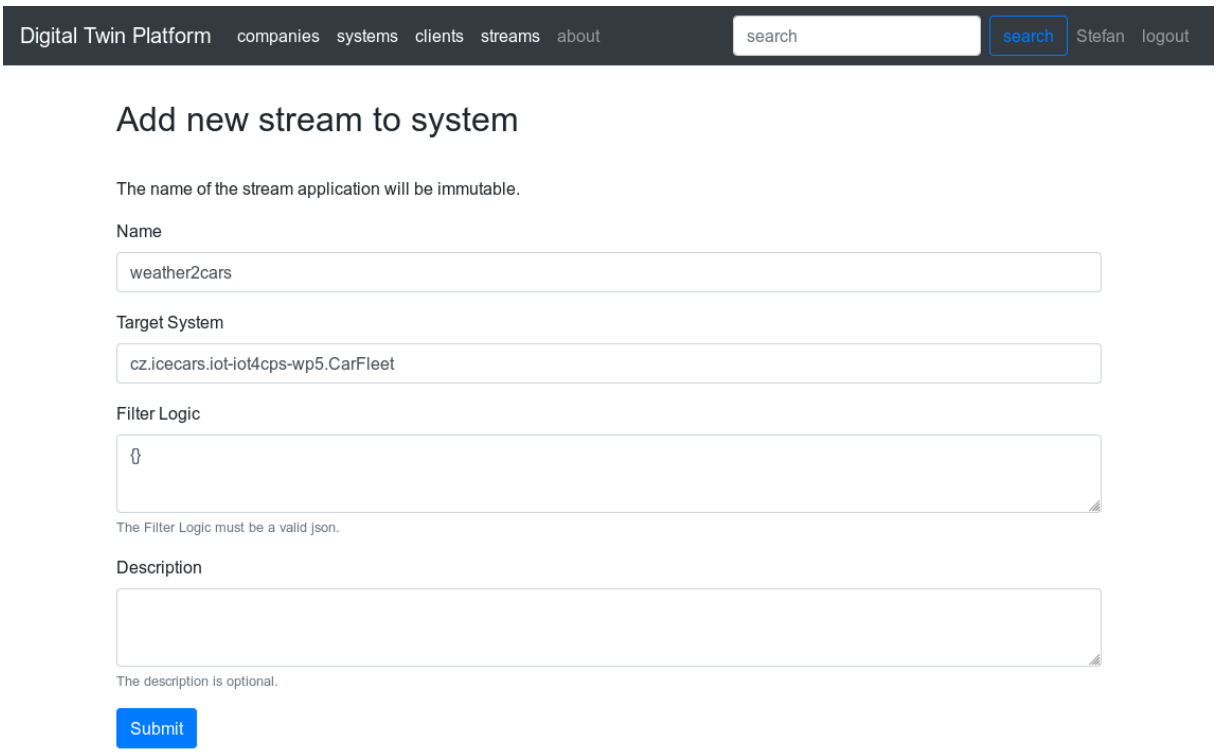


**Figure 15 - Creating a new stream application.**

In Figure 16, the view of the streaming application “weather2cars” is shown. As no filter logic is provided, any data point from the source system “is.iceland.iot-iot4cps-wp5.WeatherService” is forwarded to the target system. As depicted by the icon in the column “Status”, the streaming application started up and is running without errors in the moment this snapshot was created. Within this view, the stream can be also stopped, restarted and deleted with its configuration.

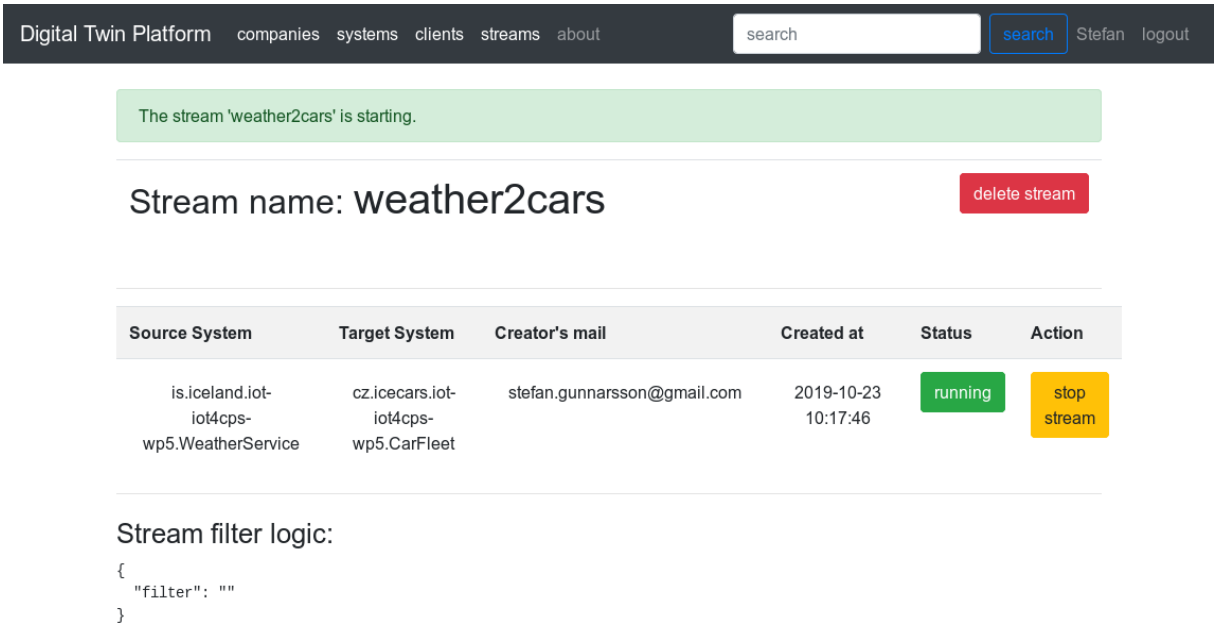


**Figure 16 - View of the streaming application "weather2cars".**

Finally, Figure 17 shows the finally created system of our demo scenario including three client applications for connecting CPSs with e.g. the forecast service, two streaming applications and two system administrators. One streaming applications is to distribute the weather information from the stations to the cars, while the other one will send all weather information to a central road analytics service for long term investigation of the weather conditions.

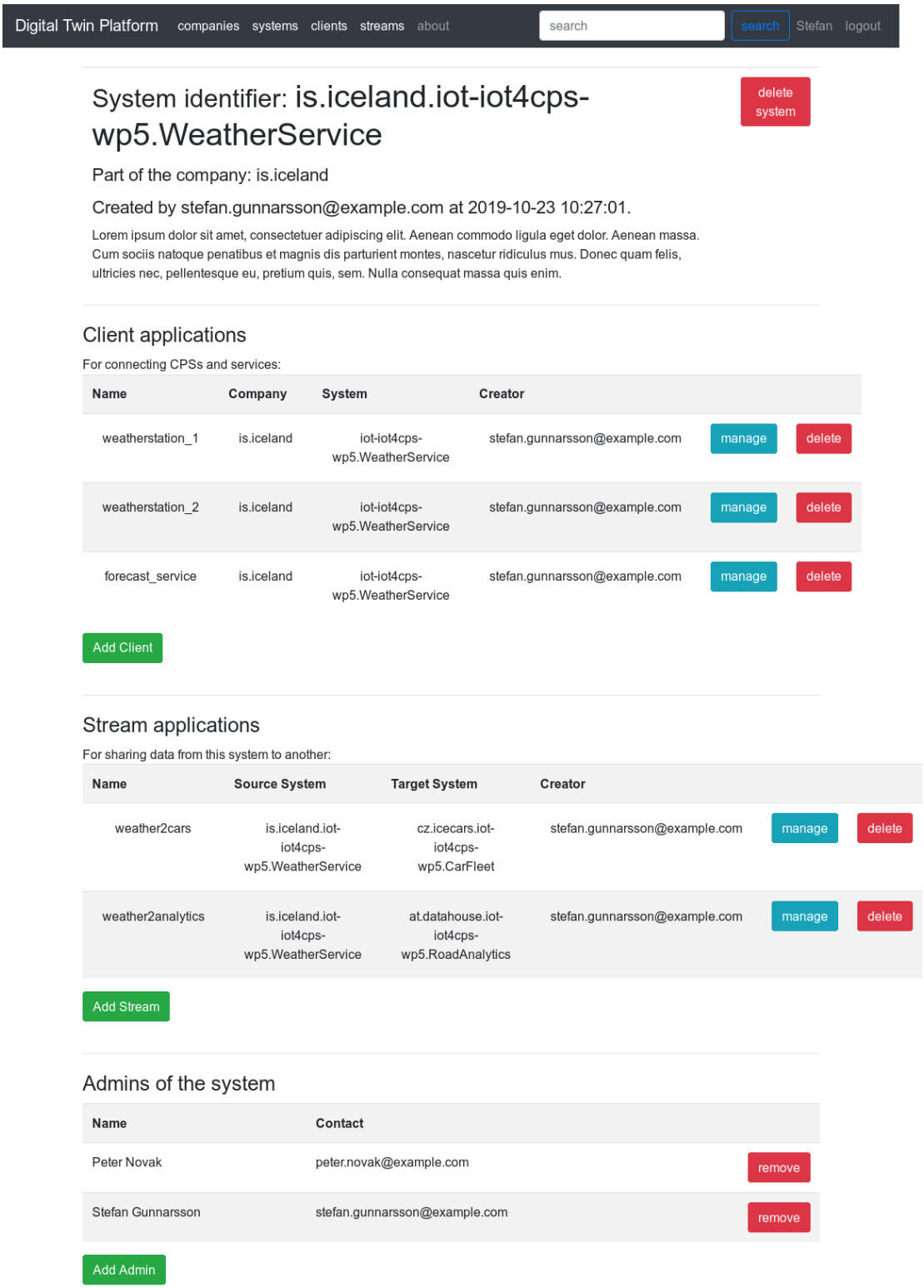


**Figure 17 - View on the a system including three client applications and two streaming applications (using dummy text for system description)**

## 3.5 Monitoring and analysing data streams

In most use cases, monitoring and analysing of collected data is a substantial feature of a digital twin platform. Therefore, we also included an analytics tool stack in the source code, including well-accepted third-party open source components that:

- Retrieve and store data,
- Visualize data and
- Provide an interactive analytics environment for use-case specific analyses.

In detail, the Elastic Stack[8] is used for storing the time-series data and Grafana[9] is used for the visualization, as this combination is well suited for metrical data and is still flexible enough to embed HTML snippets or interactive 3D graphics using plugins. Data analytics can be deployed in Jupyter[10] notebooks, which are browser applications that run code of various languages and use the rich data science packages provided by the Anaconda[11] project. Figure 18 gives an overview of the tool stack, where the red arrows depict the direction of the data flow.
To minimize the effort for the setup and help the developer to focus on his or her main task, each component is "dockerized", i.e., the installation process including some provisioned configuration is set in a Dockerfile[12] that can be deployed using a single command. More information about the setup can be found in Appendix A and the referred repository.

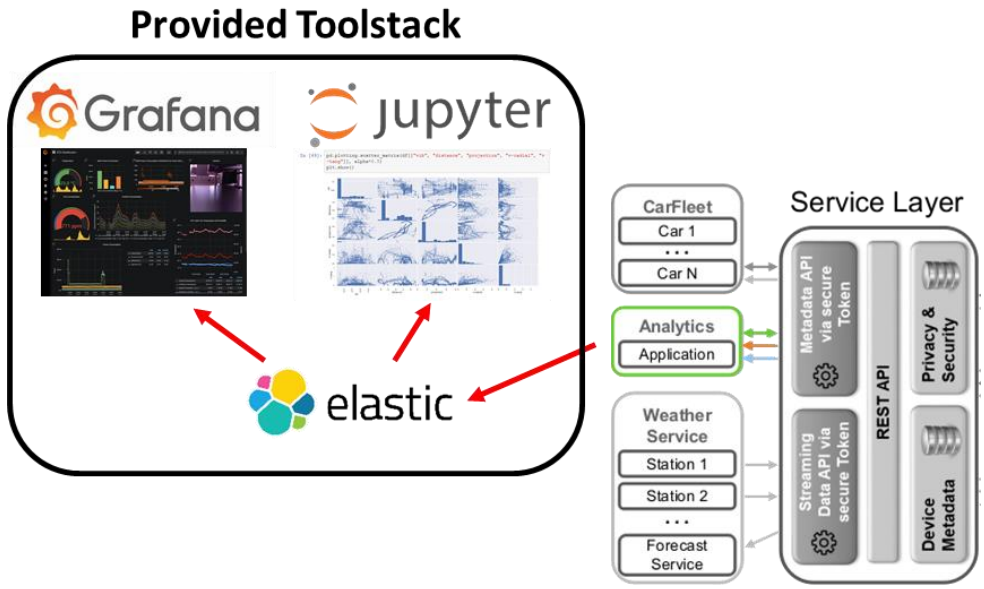


**Figure 18 - The Analytics sample code provides a Data Science toolstack.**

[8] Elastic Stack: https://www.elastic.co/
[9] Grafana: https://grafana.com/
[10] Jupyter Project: https://jupyter.org/
[11] Anaconda Project: https://www.anaconda.com/
[12] Docker: https://www.docker.com/

# 4. Source Code and Current Status

The source code of the platform can be found on a GitLab instance that is hosted by AIT[13]. To provide any reader of this public deliverable access to the open source code, a fork[14] on GitHub has been created that is public and updated regularly.

Table 1 shows the milestones and the current achievements. The major effort in the next months will be needed for the milestones “security” and “stream apps”.

**Table 1: Current status per milestone.**

| Milestone | Status |
|---|---|
| Use-Case | Finished |
| Architecture | Finished |
| Streaming Capability | Finished |
| Semantic | finished for the use-case, could be extended for actuators |
| Demo-Clients | Finished |
| Platform UI | Finished, some UI/UX improvements are possible |
| Security | Finished to secure the platform;<br>secure API for the platform;<br>communication via SSL/TLS via a given key |
| Stream apps | Finished the UI;<br>basic deployment is finished;<br>design a filter logic language;<br>parsing the filter logic |
| Dissemination | Document new features |

# 5. Conclusion

This deliverable documents the status of the Digital Twin Platform Prototype by November 2019. Based on a proof-of-concept use case it was demonstrated, that the platform already enables user and company registration, and that users can create client applications for exchanging data via data streaming applications. Currently, the data model of the data exchanged is still limited and will be targeted in the next iteration of the implementation, together with the finalisation of the platform security and the streaming applications.

The final implementation of the prototype will be released in the last quarter of the project in August 2020.

[13] Gitlab WP5: https://git-service.ait.ac.at/im-IoT4CPS/WP5-lifecycle-mgmt

[14] Fork on Github: https://github.com/iot-salzburg/panta_rhei

# Appendix A. Installation Guide

In this appendix, we provide the current snapshot version of the installation guide. An updated version will be continuously available within the project-internal GitLab repository[15]. The final version will also be published on GitHub. The installation guide includes five subtopics:

- Setup of the Messaging Layer
- Starting of Demo Applications
- Tracking to see what happens behind the scenes
- Deployment on a cluster
- Platform UI

## Setup Messaging Layer

### 1) Requirements

- Install Docker[16] version **1.10.0+**
- Install Docker Compose[17] version **1.6.0+**
- Clone the WP5 GitLab repository: https://git-service.ait.ac.at/im-IoT4CPS/WP5-lifecycle-mgmt
- Install python modules:

```
pip3 install -r setup/requirements.txt
```

This is an instruction on how to set up a demo scenario on your own hardware using Ubuntu 18.04. It contains only the most essential steps without any special procedures for different computer environments. In case of any installation problems with individual basic software components, we ask you to visit the corresponding web pages.

### 2) Setup Apache Kafka and its library

The Datastack uses Kafka **version 2.1.0** as the communication layer, the installations is done in `/kafka`.

```
sudo apt-get update
sh setup/kafka/install-kafka-2v1.sh
sh setup/kakfa/install-kafka-libs-2v1.sh
# optional:
export PATH=/kafka/bin:$PATH
```

Then, start Zookeeper and Kafka and test the installation:

```
# Start Zookeeper and Kafka Server
/kafka/bin/zookeeper-server-start.sh -daemon
kafka/config/zookeeper.properties
/kafka/bin/kafka-server-start.sh -daemon kafka/config/server.properties

# Test the installation
/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --list
/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic
test-topic --replication-factor 1 --partitions 1
/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
test-topic
```

[15] Gitlab WP5: https://git-service.ait.ac.at/im-IoT4CPS/WP5-lifecycle-mgmt

[16] https://www.docker.com/community-edition#/download

[17] https://docs.docker.com/compose/install/

```
>Hello Kafka
> [Ctrl]+C
/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --
topic test-topic --from-beginning
Hello Kafka
```

If that works as described, you can create the default topics:

```
sh setup/kafka/create_defaults.sh
/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --list
```

If multiple topics were generated, everything worked well.

### 3) Setup SensorThings Server (GOST) to add semantics

```
docker-compose -f setup/gost/docker-compose.yml up -d
```

The flag -d stands for daemon mode. To check if everything worked well, open http://localhost:8084/ or view the logs:

```
docker-compose -f setup/gost/docker-compose.yml logs -f
```

## Start Demo Applications

Now, open new terminals to run the demo applications:

### CarFleet - Prosumer

```
python3 demo_applications/CarFleet/Car1/car_1.py
> INFO:PR Client Logger:init: Initialising Digital Twin Client with name
'demo_car_1' on 'at.srfg.iot-iot4cps-wp5.CarFleet'
....
> The air temperature at the demo car 1 is 2.9816131778905497 °C at 2019-
03-18T13:54:59.482215+00:00


python3 demo_applications/CarFleet/Car2/car_2.py
> INFO:PR Client Logger:init: Initialising Digital Twin Client with name
'demo_car_2' on 'at.srfg.iot-iot4cps-wp5.CarFleet'
...
> The air temperature at the demo car 2 is 2.623506013964546 °C at 2019-03-
18T12:21:27.177267+00:00
>   -> Received new external data-point of 2019-03-
18T13:54:59.482215+00:00: 'at.srfg.iot-iot4cps-wp5.CarFleet.car_1.Air
Temperature' = 2.9816131778905497 degC.
```

### WeatherService - Producer

```
python3 demo_applications/WeatherService/demo_station_1/demo_station_1.py
python3 demo_applications/WeatherService/demo_station_2/demo_station_2.py
python3 demo_applications/WeatherService/central_service/weather-service.py
```

Here, you should see that temperature data is produced by the demo stations and consumed only by the central service.

### Analytics - Consumer and DataStack

The Analytics Provider consumes all data from the stack and pipes it into an Elastic Grafana and Jupyter Datastack.

First, the following configurations have to be set in order to make the datastore work properly:

```
ulimit -n 65536  # Increase the max file descriptor
sudo sysctl -w vm.max_map_count=262144  # Increase the virtual memory
sudo service docker restart  # Restart docker to make the changes work
```

Further information is available on the Elastic Search Website[18].

Now it can be started:

```
sh demo_applications/InfraProvider/start-full-datastack.sh
# Wait until Kibana is reachable on localhost:5601
python3 demo_applications/InfraProvider/datastack_adapter.py
```

## Available Services:

- localhost:9200 Elasticsearch status
- localhost:9600 Logstash status
- localhost:5000 Logstash TCP data input
- **localhost:5601 Kibana Data Visualisation UI**
- **localhost:3000 Grafana Data Visualisation UI**
- **localhost:8888 Jupyterlab DataScience Notebooks**

As no StreamHub application runs for now, no data is consumed by the `datastack-adapter` that ingests it into the DataStack. Therefore, it is important to start the StreamHub applications as noted in the next section.

### Stream Hub - Connect the systems

The StreamHub application can be regarded as hub for the streamed data. Run the built jar file to share data from the specific tenant to others:

```
java -jar
demo_applications/streamhub_apps/out/artifacts/streamhub_apps_jar/streamhub_apps.jar --stream-name mystream --source-system cz.icecars.iot-iot4cps-wp5.CarFleet target-system at.datahouse.iot-iot4cps-wp5.RoadAnalytics --filter-logic {} --bootstrap-server 127.0.0.1:9092
```

If you want to change the streamhub application itself, modify and rebuild the java project in `demo_applications/streamhub_apps`.
It is recommended, to start and stop the stream-applications via the Platform UI, that provides the same functionality as the command line interface.

## Track what happens behind the scenes:

Check the created kafka topics:

```
/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --list
at.srfg.iot-iot4cps-wp5.CarFleet.data
at.srfg.iot-iot4cps-wp5.CarFleet.external
at.srfg.iot-iot4cps-wp5.CarFleet.logging
…
```

Note that kafka-topics must be created manually as explained in the Setup.

To track the traffic in real time, use the `kafka-consumer-console`:

```
/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic at.srfg.iot-iot4cps-wp5.CarFleet1.data
```

[18] https://www.elastic.co/guide/en/elasticsearch/reference/7.2/docker.html#docker-cli-run-prod-mode

```
> {"phenomenonTime": "2018-12-04T14:18:11.376306+00:00", "resultTime":
"2018-12-04T14:18:11.376503+00:00", "result": 50.05934369894213,
"Datastream": {"@iot.id": 2}}
```

You can use the flag `--from-beginning` to see the whole recordings of the persistence time which are two weeks by default. After the tests, stop the services with:

```
/kafka/bin/kafka-server-stop.sh
/kafka/bin/zookeeper-server-stop.sh
docker-compose -f setup/gost/docker-compose.yml down
```

If you want to remove the SensorThings instances from the GOST server, run `docker-compose down -v`.

### Deployment on a Cluster

For a production deployment of the messaging system, we recommend to setup the platform in a cluster environment such as “Docker Swarm” or “Kubernetes”. A setup guide for a Docker Swarm deployment is available in the Software repository.

### Platform UI

The user interface is the recommended way to create system topics, which happens when registering a new client, and to deploy and stop stream-applications that forwards selected data from one system to another.

#### Starting the platform

Before starting the platform, make sure **postgreSQL** is installed and the configuration selected in `server/.env` points to an appropriate config in `server/config`. For instructions on how to install postgres, various tutorials can be found in the Internet[19].

```
cd server
sudo pip3 install virtualenv
virtualenv venv
source venv/bin/activate

pip3 install -r requirements.txt
sh start-server.sh
```

The Platform will be available on port 1908.

## Appendix B: Client Applications

In the following code snippet both, a subscriber (`client.subscribe(…)`) as well as a producer client (`client.produce(…)`) is implemented. Note that they will usually be separated into two separate threads.

```
#!/usr/bin/env python3
"""
Demo Scenario: Connected Cars
    CarFleet:
        The connected car wants to enhance it's safety by retrieving temperature data, to warn the driver on
        approaching slippery road sections. As each car has also temperature data that is of interest for other
        cars, it sends this data to the the platform.
```

[19] e.g. https://www.digitalocean.com/community/tutorials/how-to-install-and-use-postgresql-on-ubuntu-18-04

```
    InfraProv:
        The provider of the road infrastructure wants to enhance it's road quality and therefore consumes and analyses
        data.
    WeatherStation:
        stations: The weather stations are conducted by a local weather service provider which provides the data as a
                  service.
        service_provider: The weather information provider offers temperature data for it's customers.
"""

import os
import sys
import inspect
import time
import pytz
from datetime import datetime

# Append path of client to pythonpath in order to import the client from cli
sys.path.append(os.getcwd())
from client.digital_twin_client import DigitalTwinClient
from demo_applications.simulator.SimulateTemperatures import SimulateTemperatures

# Get dirname from inspect module
filename = inspect.getframeinfo(inspect.currentframe()).filename
dirname = os.path.dirname(os.path.abspath(filename))
INSTANCES = os.path.join(dirname, "instances.json")
SUBSCRIPTIONS = os.path.join(dirname, "subscriptions.json")

# Set the configs, create a new Digital Twin Instance and register file structure
# This config is generated when registering a client application on the platform
# Make sure that Kafka and GOST are up and running before starting the platform
config = {"client_name": "car_1",
          "system": "cz.icecars.iot-iot4cps-wp5.CarFleet",
          "gost_servers": "localhost:8084",
          "kafka_bootstrap_servers": "localhost:9092"}
client = DigitalTwinClient(**config)
# client.register_existing(mappings_file=MAPPINGS)
client.register_new(instance_file=INSTANCES)  # Registering of new instances should be outsourced to the platform
client.subscribe(subscription_file=SUBSCRIPTIONS)
randomised_temp = SimulateTemperatures(t_factor=100, day_amplitude=4, year_amplitude=-4, average=3)

try:
    while True:
        # unix epoch and ISO 8601 UTC are both valid
        timestamp = datetime.utcnow().replace(tzinfo=pytz.UTC).isoformat()

        # Measure the demo temperature
        temperature = randomised_temp.get_temp()

        # Send the demo temperature
        client.produce(quantity="temperature", result=temperature, timestamp=timestamp)

        # Print the temperature with the corresponding timestamp in ISO format
        print("The air temperature at the demo car 1 is {} °C at {}".format(temperature, timestamp))

        # Receive all queued messages of the weather-service and other connected cars and calculate the minimum
        minimal_temps = list()
        if temperature <= 0:
            minimal_temps.append({"origin": config["system"], "temperature": temperature})

        received_quantities = client.consume(timeout=0.5)
        for received_quantity in received_quantities:
```

```
            # The resolves the all meta-data for an received data-point
            print("  -> Received new external data-point at {}: '{}' = {} {}."
                  .format(received_quantity["phenomenonTime"],
                          received_quantity["Datastream"]["name"],
                          received_quantity["result"],
                          received_quantity["Datastream"]["unitOfMeasurement"]["symbol"]))
            # To view the whole data-point in a pretty format, uncomment:
            # print("Received new data: {}".format(json.dumps(received_quantity, indent=2)))
            if received_quantity["result"] <= 0:
                minimal_temps.append(
                    {"origin": received_quantity["Datastream"]["name"], "temperature": received_quantity["result"]})

        if minimal_temps != list():
            print("    WARNING, the road could be slippery, see: {}".format(minimal_temps))

        time.sleep(10)
except KeyboardInterrupt:
    client.disconnect()
```