IoT4CPS – Trustworthy IoT for CPS

FFG - ICT of the Future
Project No. 863129

Deliverable D5.5.3
Lifecycle Data Management Prototype

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: D5.5.3

Amendment History

Version	Date	Author	Description/Comments
V0.1	16.07.2020	Felix Strohmeier	Initial version prepared
V0.2	19.08.2020	Christoph Schranz	Update Use Case, introduce Streaming Applications, update and extended UI, update and extended appendix
V0.3	26.08.2020	Felix Strohmeier	Added Industry 4.0 Use Case
V0.4	27.08.2020	Christoph Schranz	Reformat the document
V1.0	31.08.2020	Felix Strohmeier	Finalisation for Review
V1.1	05.10.2020	Felix Strohmeier	Integrated comments from Review
V1.2	12.10.2020	Felix Strohmeier	Integrated comments from 2nd Review

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 4
Executive Summary 5
1. Introduction 6
2. Example Scenario Descriptions 6
2.1 Connected Car Scenario 6
2.2 Industry 4.0 Scenario 7
3. Architecture of Lifecycle Data Management Prototype III 8
3.1 Digital Twin Messaging and Data layer 9
3.1.1 Streaming Platform with Data Stream Apps (Apache Kafka) 9
3.1.2 Device Metadata (SensorThings) 9
3.1.3 Digital Twin Platform Identity Data Model 10
3.2 Digital Twin Service layer 11
3.3 Application and User Interface layer 11
3.4 Stream Hub Service 11
3.4.1 Streaming Application Types 11
3.4.2 Streaming Application Semantic 12
3.4.3 Stream Application Implementation 13
3.4.4 Deterministic Time-Series Join of Streaming Data 14
4. Example Use Case 16
4.1 Welcome screen and user registration 17
4.2 Company and Systems Management 20
4.2.1 Companies 20
4.2.2 Systems 21
4.3 Client Applications 23
4.4 Streaming Applications 25
4.4.1 Single-Source Streaming Application 27
4.4.2 Multi-Source Streaming Application 29
4.5 Monitoring and analysing data streams 31
5. Source Code and Current Status 33
6. Conclusion 33
7. Appendix 35
7.1 Appendix A. Installation Guide 35
7.1.1 Requirements 35
7.1.2 Setup Apache Kafka and its library 35
7.1.3 Setup SensorThings Server (GOST) to add semantics 36

7.1.4 Start Demo Applications 36
7.1.5 Streaming Applications 38
7.1.6 Track what happens behind the scenes: 38
7.1.7 Deployment on a Cluster 39
7.1.8 Starting the platform 39
7.2 Appendix B: Client Applications 40
7.3 Appendix C: Custom Functions for a Multi-source StreamApp 43

Abbreviations

API	Application Programming Interface
CPS	Cyber-Physical System
CRUD	Create, Read, Update, Delete
DNS	Domain Name System
ERP	Enterprise Resource Planning
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
SBI	Security By Isolation
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 final version of the Lifecycle Data Management Prototype developed in IoT4CPS. The core component of the prototype is the third 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. The novelty of the approach is that live-data sharing can happen “on thy fly”, i.e. if supported by the underlying network and hardware, also (soft) real-time requirements can be fulfilled. Customisation can be achieved by connecting data streams through customisable filtering mechanisms from a single data source or by injecting more complex streaming applications when joining data from multiple data sources. This concept was implemented as configurable platform based on the scalable open-source data-streaming framework “Apache Kafka”. Each pair of “data producer” and “data consumer” is identified 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. In the last part of the iteration, the focus is on Streaming Filter Applications as well as the secure prototype deployment using the “Security By Isolation” Concept implemented in the SBI-Box developed by X-Net.
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.

1. Introduction

This report contains the documentation of the 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, and D5.5.2 published in December 2019, it contains the advances achieved during the final implementation period of the project from December 2019 to August 2020.

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 deterministic filtering mechanisms and join operations for shared high-throughput data-streams between CPSs across boundaries of companies and administrative domains. This is especially important in use cases of a multi-tenant Digital Twin Platform, where streaming data is shared under constraints of the data source (e.g. for privacy or security reasons), or two or multiple streams have to be merged to a new one.
In each scenario, multiple systems from multiple companies are involved along the value chain, where each of them needs a different view and different combination of original raw data streams to realise their own application specific digital twin. Beneath raw data collections, it also allows real-time data analytics applications to prepare the input streams accordingly. In the final prototype application, we want to demonstrate the life cycle data management of both, industrial control units as well as automotive components in use, with separated stakeholder control over subsets of the data and – through customisation – keeping compliance with different regulations regarding privacy and third-party usage of data.
In order to facilitate the structured sharing of streaming data, we identified and implemented two different types of streaming applications between multiple tenants. The first type of streaming applications consumes data from a single source, applies a customisable filtering mechanism on it that is specified in an SQL-like expression language and forwards the data to a target tenant. The second type consumes streaming data from two sources, joins the respective time-series and applies custom functions on the resulting stream. This allows the implementation of different stream processing mechanisms over multiple data streams integrated in the already existing management UI.

2. Example Scenario Descriptions

In this section, we describe example scenarios within two application domains, one for connected cars and another for an Industry 4.0 context with connected production machines.

2.1 Connected Car Scenario

The goal of this scenario is to warn connected cars when approaching dangerous road conditions, such as ice or wetness, due to bad weather conditions. Each connected car is equipped with sensors to measure the temperature and brake events and to share information with others. As illustrated in Figure 1, information should only be exchanged between two vehicles when they are in proximity, to follow the data minimization principle. The size of the proximity area depends on a given radius. Since the relative distance between two vehicles is required for the filter condition, but the coordinates are not measured synchronously on the vehicles, it is necessary to join the time-series of the data streams and then filter them according to their resulting relative distance. For the rarely occurring brake events, it must be ensured that this quantity is always received by nearby cars. Furthermore, the presence of *n* cars in one geographical area leads to *n*(n-1)* possible streams between cars, resulting in a high throughput of data. Therefore, a demand for deterministic and efficient time-series joins emerged, which was solved within this project and explained in Section 3.4. The example use case provided in Section 4 is based on this connected car scenario.

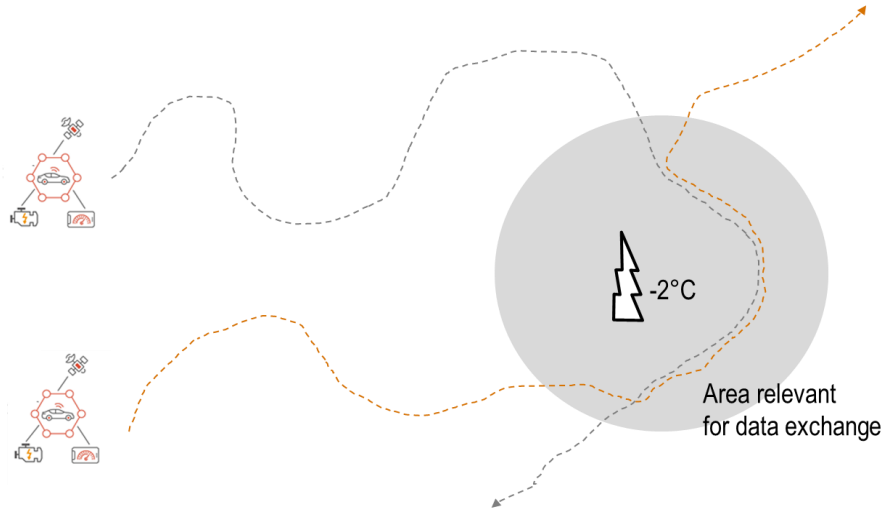

Figure 1: Example Scenario for Connected Cars

2.2 Industry 4.0 Scenario

The Industry 4.0 scenario targeted in this deliverable connects multiple sites from production companies to enable order distribution among free resources that are available in any production site. This is especially important in situations where orders have a targeted delivery date, but due to unplanned machine breakdowns they cannot be produced in the originally planned production site.

To support demonstration, a prototypical implementation of two smart production machines (= 3D printing machines, including network connection for remote interaction such as starting, stopping or pausing the production process) in remote locations have been connected to a central printing service that monitors the production processes and supports automatic failover in case of printing errors in one of the printers. The quality of the prints is permanently monitored and depending on the preconfigured quality level, prints are cancelled and automatically restarted on the selected failover printer. Figure 2 shows the generic setup of such a scenario. ERP and production machines (e.g. 3D printers) are located in the site of customer 1, while another set of machines are available at customer 2. In order to allow scenarios such as automatic failover in case of unplanned downtimes, selected data streams have to be exchanged between the two sites. By the application of customisable filtering mechanisms, data streamed from one customer to the other can be configured to ensure the required privacy, for example. The Industry 4.0 scenario is used in one of the IoT4CPS project demonstration setups (Deliverable D7.2).

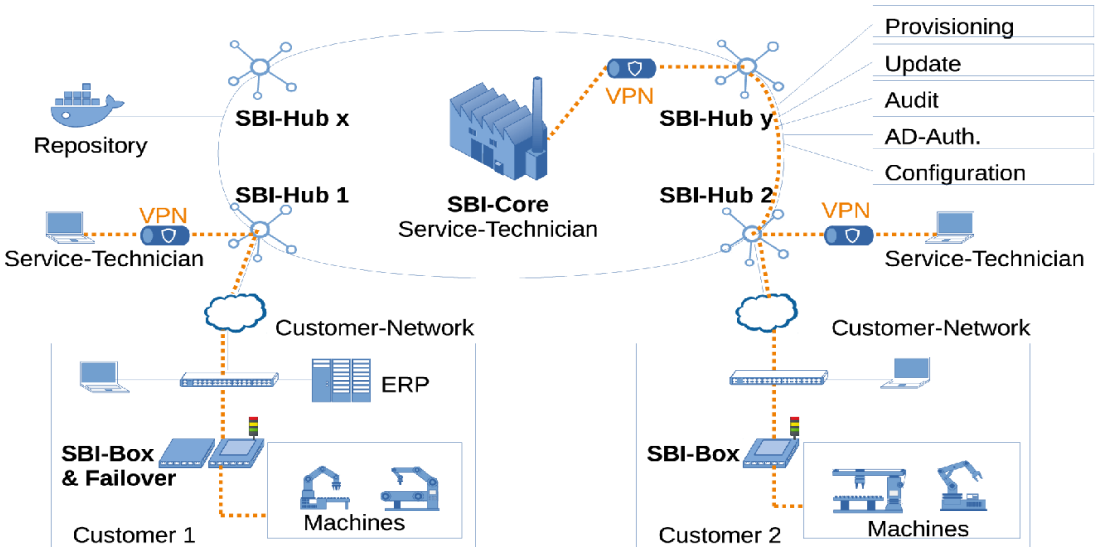

Figure 2: Example Scenario for Industry 4.0

3. Architecture of Lifecycle Data Management Prototype III

In this section, we describe the final architecture of our Digital Twin Platform prototype for Lifecycle Data Management between multiple tenants, including its surrounding components. An architectural overview is shown in Figure 3. On the upper layer, it distinguishes between two types of platform clients, the application running 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 or a connected machine), the latter one enables the interaction with a human user through a user interface, such as a web browser or a mobile application.

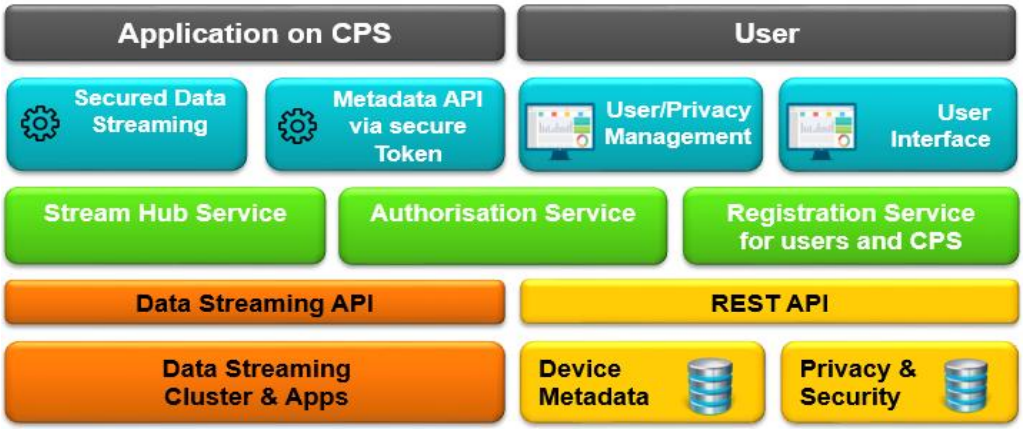

Figure 3: High-level Component Architecture of the Prototype

In order to achieve a privacy-aware and secure environment with the digital twin platform prototype, the architecture has been integrated with the latest security technology developed in IoT4CPS, the “SBI-Box”, as shown in Figure 4. This additional security layer connects selected distributed CPSs via dedicated, secure VPNs, easily configured and managed via the SBI-Cloud platform. Furthermore, it allows the implementation of firewall rules between the machines as well as external users, such as service technicians that can get access to the machines via SBI-Hubs that can be distributed among the internet. In case no fixed internet connection is provided, the SBI-Box is also equipped with a GSM/LTE mobile data module.

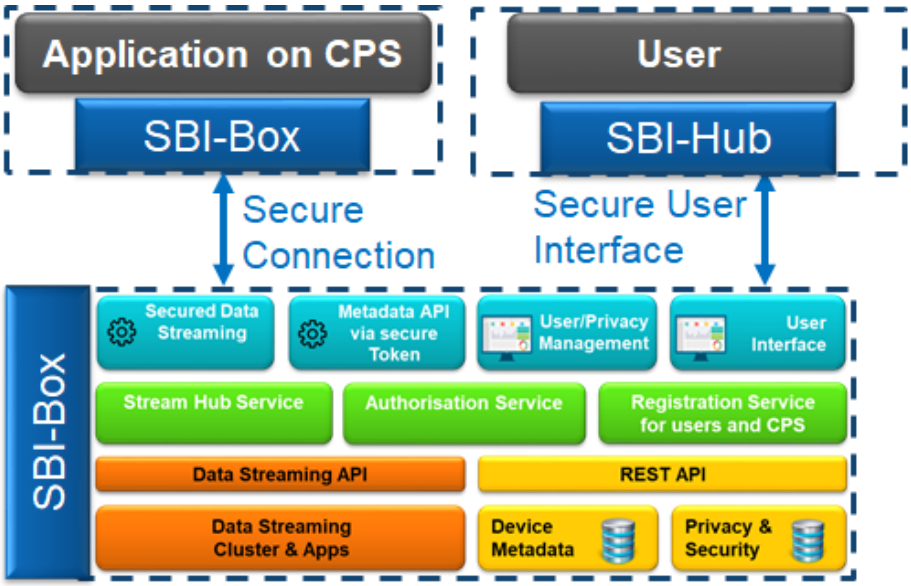

Figure 4: Digital Twin Platform secured by X-Net's SBI-Box

A common use case for such integration is to provide secure connection for either a smart production machine provider or service technician to remote network islands, e.g. if a production machines itself is not connected to the local internet or just connected behind a company firewall. As already shown in Figure 2, two separate production sites and a remote service technician need to get shared access to data streams from different production locations, e.g. to implement a smart, adaptive production schedule based on machine availability in the remote sites. The combination of the Digital Twin Streaming Platform with the SBI-Box enables a secure communication between the remote sites, and also enables integration of Streaming Applications (StreamApps) into the overall workflow. The concept of the StreamApps will be described in more details in Section 3.4 below.

The following sections describe the different layers of the architecture, from the bottom to the top.

3.1 Digital Twin Messaging and Data layer

3.1.1 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 various source data streams, filter, process, or analyse them and return altered data streams back to the Kafka cluster. These Data Streaming Applications are later utilized for single-source data sharing between two tenants.

3.1.2 Device Metadata (SensorThings)

As shown in the figures above, one of the architecture components is dedicated for storing metadata for devices, such as sensors. 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. To avoid that this metadata needs to be delivered in every data packet for a measurement, this information is managed using an external service accessible for all parties. A useful option for such service is to use a SensorThings

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

server, which provides a SensorThings API[2] as defined by the Open Geospatial Consortium (OGC). For the prototype implementation we use a GOST SensorThings server[3], composed of three separate Docker[4] containers (one for the database, one for the service API and one for the dashboard).

3.1.3 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 known 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 stream app implementation as described in more detail in section 3.4, flexible many-to-one 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, that enables the user to log in via an arbitrary OAuth2.0-registered account, as from e.g., Google, Facebook or Github.

The physical data model is depicted in Figure 5.

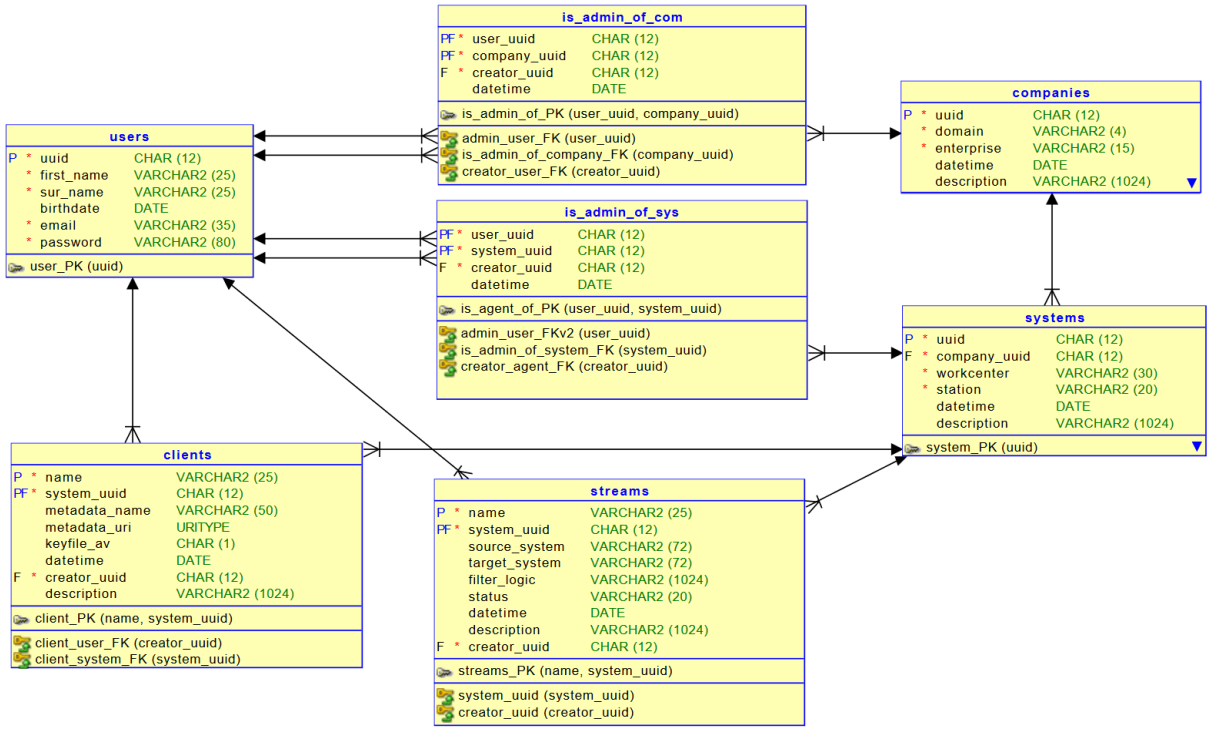

Figure 5: Data Model for the Digital Twin Platform

[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

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.

3.2 Digital Twin Service layer

The service layer provides controlled access to the database and implements the standard CRUD[7] operations on companies, users, systems, clients and streams. In the prototype, the service layer is implemented in the programming language Python using a light-weight web application framework, called Flask[8].

3.3 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 consumer and producer client applications implemented in Python are provided together with the source code of the platform.

3.4 Stream Hub Service

The components presented so far enable the communication between client applications and their dedicated system and therefore also the communication between various applications and devices within the same system. However, there are countless examples of cases that require a structured way of sharing a subset of streaming data from one or multiple sources to a single target system. To reduce the overall complexity, we distinguish between two basic types of streaming applications. On the one hand, there are streaming applications that consume data only from a single source stream and apply some filter function to produce the stream for the target system. On the other hand, more complex applications can have two source streams as input and apply a joining function on specific pairs of data points to produce the output stream for the target system. For both cases, separate prototypes have been implemented and are integrated by a single user interface within the Digital Twin Platform's Stream Hub Service. Both types of streaming applications and their implementations are described in more detail in the following subsections.

3.4.1 Streaming Application Types

As already mentioned, we present two different types of streaming applications for data exchange, one consuming data from a single source (i.e. client application) another one from multiple sources.
The more trivial case is the ***Single-source Stream App***, where only a subset of streaming data is consumed, a custom filtering is applied, and the resulting data is forwarded to the target system.
The Single-source Stream App seems flexible, but does not suffice for complex data streams that are based on joining two streams based on close timestamps. This requires a ***Multi-source Stream App***. For example, in the connected car scenario we illustrate a case in which data is transferred from one car to another only if the relative distance between the cars is below a certain threshold. As the positions of the cars change dynamically, both of their coordinates have to be consumed frequently in order to compute their relative distance.

[7] Create, Read, Update, Delete
[8] https://palletsprojects.com/p/flask/, https://github.com/pallets/flask

Moreover, the coordinates of the cars are not sent synchronously, meaning that in general all data point differ in their timestamps. Although we can assume that the clocks in the connected cars are sufficiently synchronised, delays in the transmission of data can occur due to network outages. To solve this problem, a time-series join of the data of both cars is required, in which each data point - a spatio-temporal sensor value - is joined with the precedent and subsequent data point from the other car. Then, the relative distance is computed between each pair of data points and filtered on the given threshold. Finally, the result value is produced and is sent together with the original timestamp and coordinates to the targeting system.
From an abstract perspective, there are similar patterns in an Industry 4.0 scenario: Imagine streaming data originating on a machine and on an ERP system. If the producer wants to forward the machine data for quality certifications to another tenant (e.g. the customer) only if a specific *product-id* that is issued by the ERP system is processed on the machine and where one of the systems (e.g. the ERP system) experiences much higher latencies than the other, i.e. the ERP-system delivers records containing the *product-id* significantly after the machine has started to work on the specific product. To enable the customer to still receive data close to real time when the production was started, the data streaming should start as soon as all data is available from the data streams of all connected systems. In the worst case, unsynchronised data streams in such scenario would lead to transmission of machine data from other customers, only because the product-id comes with latency! Therefore, the determinism of Time-Series joins is very important.
To summarize, there are cases both in the automotive industry and Industry 4.0 that require deterministic time-series joins and a more complex stream sharing semantic. To achieve that, an efficient algorithm for time-series joins was developed that is deterministic, has minimal latency and enables a high throughput. In our test setup, up to 100.000 joins per second in Python only, and up to 15.000 joins per second with exactly-once processing using Apache Kafka as streaming platform running on a medium-sized desktop computer was possible.

3.4.2 Streaming Application Semantic

Since the Digital Twin Platform is a prototype, the streaming application expression language is not yet unified for the two types. However, this decision ensures the flexibility of the second type while it preserves the simplicity of the first one.

Expression language for the Single-source StreamApp:

The main goal for the expression language for Single-source stream applications is the simplicity. Therefore, we chose an SQL-like expression, because SQL is a standard language, easy to learn and very compact.
For instance, if a specific temperature value measured at a weather station exceeds 30°C or is below 4°C and it should be transferred to a target system, one could apply this expression to define the behaviour of the StreamApp:

```
SELECT * FROM * WHERE name = ‘is.iceland.iot4cps-wp5-WeatherService.Stations.Station_1.Air Temperature'
AND (result < 4 OR result > 30);
```

This expression will forward each data point (marked by the asterisk symbol ‘*’) that suffices the filter mechanism defined after the “WHERE”-keyword.
As the transmitted data point should comply with the SensorThings standard, it is not recommended to select only a subset of attributes from a single data point. Therefore, this point is not implemented yet. The expression after the “FROM”-keyword is not yet required and is therefore also marked with the asterisk symbol ‘*’, because the input stream from the source system is already defined by the “TARGET_SYSTEM” within the UI of the Stream Hub Service. This is described in more detail below in the description of the UI implementation.

Expression language for the Multi-source StreamApp:

To handle the complexity of a Multi-source StreamApp and the demand for high flexibility, the customizable parts of the application are defined in a file called *'custom_fct.py'*, following a predefined, simple structure. It just consists of some constants and two functions, called *'ingest_fct'* and *'on_join'*. The required constants (uppercase) are the following:

```
KAFKA_BOOTSTRAP_SERVERS: kafka nodes of the form 'mybroker1,mybroker2'
SYSTEM_IN: list of systems to consume data from, multiple comma-separated sources
SYSTEM_OUT: target system which should receive the produced resulting data

# time-series join specific configuration:
TIME_DELTA: Maximal time difference between two Records being joined
ADDITIONAL_ATTRIBUTES: optional attributes in the observation records, "att1,att2,..."
USE_ISO_TIMESTAMPS: boolean: timestamp format of the resulting records, ISO 8601 or unix timestamp if False
MAX_BATCH_SIZE: consume up to this number of messages at once
TRANSACTION_TIME: time interval for committing transactions, in seconds
VERBOSE: boolean, prints out more messages for debugging
```

In addition to these constants, two functions have to be defined:

- ingest_fct: This method gets the received Record and the Stream Buffer instance as arguments and specifies under which constraints the Record is ingested into the left or right buffer (or not at all) of the Stream Buffer instance. Note that every binary join requires two single data points from two different inputs, whereby one of them is denoted as left-, the other as right join partner.
- on_join: This function receives two Records one left and one right join candidate. Within the function, custom filtering and merging mechanisms can be applied. Then the resulting record, or a NULL value if no join is desired, is returned.

A full example of a custom function file ('custom_fct.py', written in Python) can be found in Appendix C: Custom Functions for a Multi-source StreamApp of this document.

3.4.3 Stream Application Implementation

For both presented cases, an example implementation is presented that is integrated into the Digital Twin Platform. The goal for each implementation is to have a stand-alone streaming application that is externally controllable, i.e. it can be deployed, halted and logging output can be fetched. A practical way to do so is to deploy each Stream App in a separate container, such as provided by Docker[9]. Docker is an orchestration software meaning that it runs applications in isolated containers. This has the advantage that each Stream App can run in a dedicated Docker container using different programming languages (Java, Python) while they can be controlled via the user interface implementation of the Python-based Digital Twin Platform.

1. Implementation of the Single-source Stream App:

The source code for the Single-source Stream Apps was implemented in Java using the Kafka Streams[10] framework. Kafka Streams is used as a library to consume and produce data from a Kafka Cluster, where it enables a flexible way of applying filter mechanisms. The main java application receives given

[9] Docker: https://www.docker.com/
[10] Kafka Streams: https://kafka.apache.org/documentation/streams/

variables like the stream name, source and target system as environment variables and parses the expression language into a specialized StreamQuery instance. Once this recursive StreamQuery instance is initialized based on a filter expression, newly received data can be filtered very quickly. The Single-source Stream App processes the data exactly-once, as the delivery guarantees are handled by the Kafka Streams library. The Java application is deployed in Docker to feature a full control from the UI.

2. Implementation of the Multi-Source Stream App:

In contrast to the Single-source Stream App, the Multi-source variant is based on an efficient time-series join of data from two data streams. The proposed in-memory implementation enables the exactly-once processing of a join for Apache Kafka as a streaming platform. It is deterministic even for arbitrary high latencies of received data.

The Multi-Source StreamApp is written in Python with the *confluent_kafka_python*[11] library and plain Apache Kafka[12] and is containerized with Docker. It receives the required constants and functions, and initializes a new Stream Buffer instance. Based on the specified method 'ingest_fct', received data is ingested into the left or right buffer (or not at all) of the Stream Buffer instance. As soon as a new join candidate is available, both join partners are passed into the 'on_join' function that defines the arbitrary merge and filter behaviour of the join partners. The resulting record is then produced to the messaging system using the provided topic for the target system. As this kind of join is very important for stream processing and a novel approach originally developed in IoT4CPS, the algorithm is presented in the next section 3.4.4.

3.4.4 Deterministic Time-Series Join of Streaming Data

The *LocalStreamBuffer* is an algorithm for the deterministic time-series join of two data streams. A time-series join merges each record within one time-series with its previous and subsequent complement from the other time-series, independently of the record's latency. The used algorithm is optimised for high throughput and ensures minimal latency, while still joining each candidate. The pre-assumption is that records within each stream are received in order, as it is guaranteed by streaming platforms such as Apache Kafka for example, as long as each record shares the same dedicated message key. A global order across streams is not required. More detailed information is published as a work-in-progress paper[13]. The subsequent description of the algorithm is related to the mentioned work that was originally developed within this project.

In a nutshell, a time-series join matches each record within one time-series with its previous and subsequent complement from the other time-series. This property holds for the record's event times and is **independent of the latency and therefore ingestion timestamp** (the records should be ingested in order within a stream but not across them, as it is guaranteed by e.g. Apache Kafka). In Figure 6, for two different ingestion orders (top: ingestion time = event time, i.e. $r_{0.5}$, s_1, $r_{1.5}$, s_2… and bottom: first all s, then all r: s_1, s_2 … s_7, $r_{0.5}$, $r_{1.5}$, $r_{5.5}$), the three join cases (JR1, JR2, JS2) are distinguished.

[11] Confluent Kafka Python library: https://docs.confluent.io/current/clients/python.html
[12] Apache Kafka: https://kafka.apache.org/
[13] Christoph Schranz and Peter-Michael Jeremias: "Deterministic Time-Series Joins for Asynchronous High-Throughput Data Streams". IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2020).

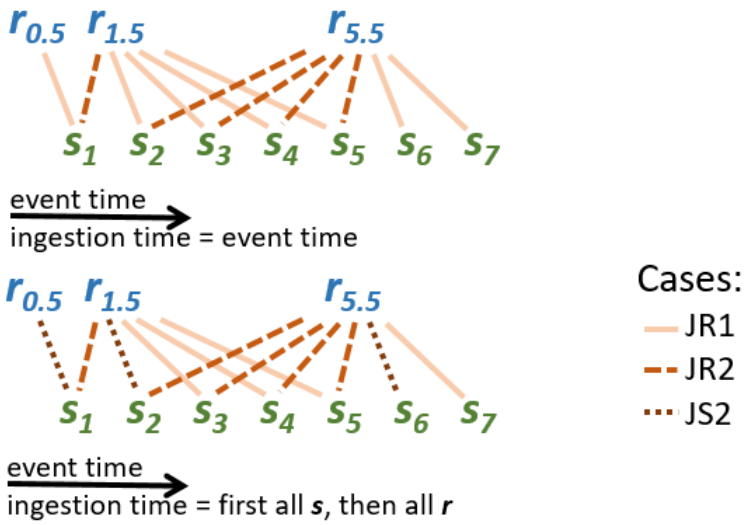

Figure 6: Deterministic Time-Series Join for two different ingestion orders

The main goal of Time-Series joins is to join the correct pairs of records deterministically with minimal latency, even if records of one data stream are delayed up to a certain threshold.

The invariance of the proposed Time-Series join algorithm is, that for any record s_j, there are two join partners r_i and r_{i+1} with the property $t(r_i) \leq t(s_j) < t(r_{i+1})$. While the indices *i* and *j* denote record counts, the function t(r) returns the records' event time, i.e., the timestamp for the event of sensing. For practical reasons, a join is discarded, if its partner's event times exceed a given threshold Δt. The invariance implies, that the number of resulting join pairs *u* is upper bounded by $|u| \leq 2\,(|r| + |s|)$ for two continuous input streams *r* and *s*.

To summarize, our Time-Series join algorithm has following constraints and measures to optimize:

- Guarantee **correct joins of records with subsequent event times** independent of their latency and ingestion times. Therefore, neither the false omission of a join pair, nor the false join of a pair is tolerated.
- **Exactly-once processing**, i.e., on a crash and restart of the join application, records in the buffer should be re-consumed and already joined pairs should not joined again.
- **Low-latency**, resulting tuples should be joined as soon as possible.
- The computational costs should be low, which allows **high sample rates** and therefore **high throughput** of data.

To meet these requirements, the algorithm makes the following assumptions: The records of each input data stream differ in their event times and should be received in the correct chronological order. Apache Kafka guarantees this behaviour for records produced with a common key. Moreover, Apache Kafka enables an exactly-once delivery and playback capability.[14] The LocalStreamBuffer algorithm is implemented in a way that utilizes this functionality in order to enable an exactly-once processing for joins.

The main data structure of the *LocalStreamBuffer is* a combination of two FIFO (first in, first out) queues, implemented as Double Linked Lists. These queues buffer all records of the two data streams that could find a

[14] M. Kleppmann, J. Kreps, "Kafka, Samza and the Unix Philosophy of Distributed Data," IBulletin of the Technical Committee on Data Engineering, p. 10, 2015.

join partner in future records. A newly received record is “enqueued” into the respective buffer, then possible join pairs are identified, and finally tuples that cannot find any further join partners are “dequeued”.

For the identification of potential join partners, we distinguish three cases on which a join occurs. For this consideration it is important that two partners are only handled as a join, if there can’t be received a new record which makes this specific join obsolete. Those join cases are illustrated in Figure 6 and named JR1, JR2 and JS2 which stand for:

- Join-case **JR1**: the pivotal buffer has the leading record and the pivotal record's predecessor finds partners,
- Join-case **JR2**: the pivotal buffer has the leading Record and the pivotal Record finds partners,
- Join-case **JS2**: the external buffer has the leading Record and the pivotal Record finds one partner,

where the pivotal record is the most recently received record and checked for new join partners. A buffer containing the pivotal record is called a pivotal buffer, therefore, the other buffer is called external.

The search and iteration for records within the buffers is optimized using Double-Linked Lists. Additionally, each new record triggers a trimming of the Double-Linked List which removes records that are obsolete, independently how new records will arrive. Additionally, those records are committed as received to the streaming platform Apache Kafka. Committing only those records will re-consume records with possible join partners stored in one of the two buffers of the *LocalStreamBuffer* in the case of a failure, which leads to an at-most-once processing. The additional usage of transactions on the producer’s side of the algorithm leads to an exactly-once stream processing for the desired Time-Series joins.

In order to filter some of the join pairs and merge them in an arbitrary way, the Stream Hub Service requires a customizable function “*on_join*” defined in “*custom_fct.py*” that is passed to a generic “*stream_join_engine*”. More detailed information about the actual implementation of the algorithm can be found in the Digital Twin Platform’s repository.

4. Example Use Case

In this section, we describe the main flow of events using screenshots of the prototype according to an example 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 temperature and breaking information (i.e. deceleration) with each other and with a central weather service. Before we discuss the use of the platform in detail, we describe the data flow in the platform shown in Figure 7.

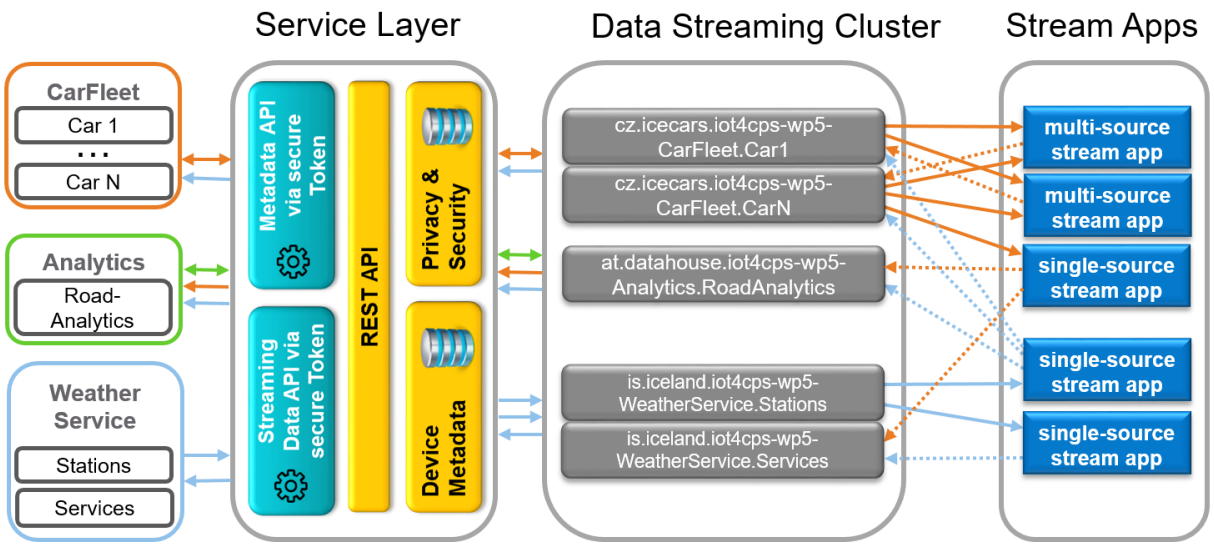

Figure 7: Data Streaming between CPSs

On the left side, the systems within separate administrative entities (e.g. companies) are shown. Client applications dedicated to a system can communicate with the Digital Twin Platform. Those applications are implemented directly in the cars, in analytics back-ends or in weather service provider stations. The 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 metadata 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 to an internal topic, or 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 subscribes to that data stream.

4.1 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 8, 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.

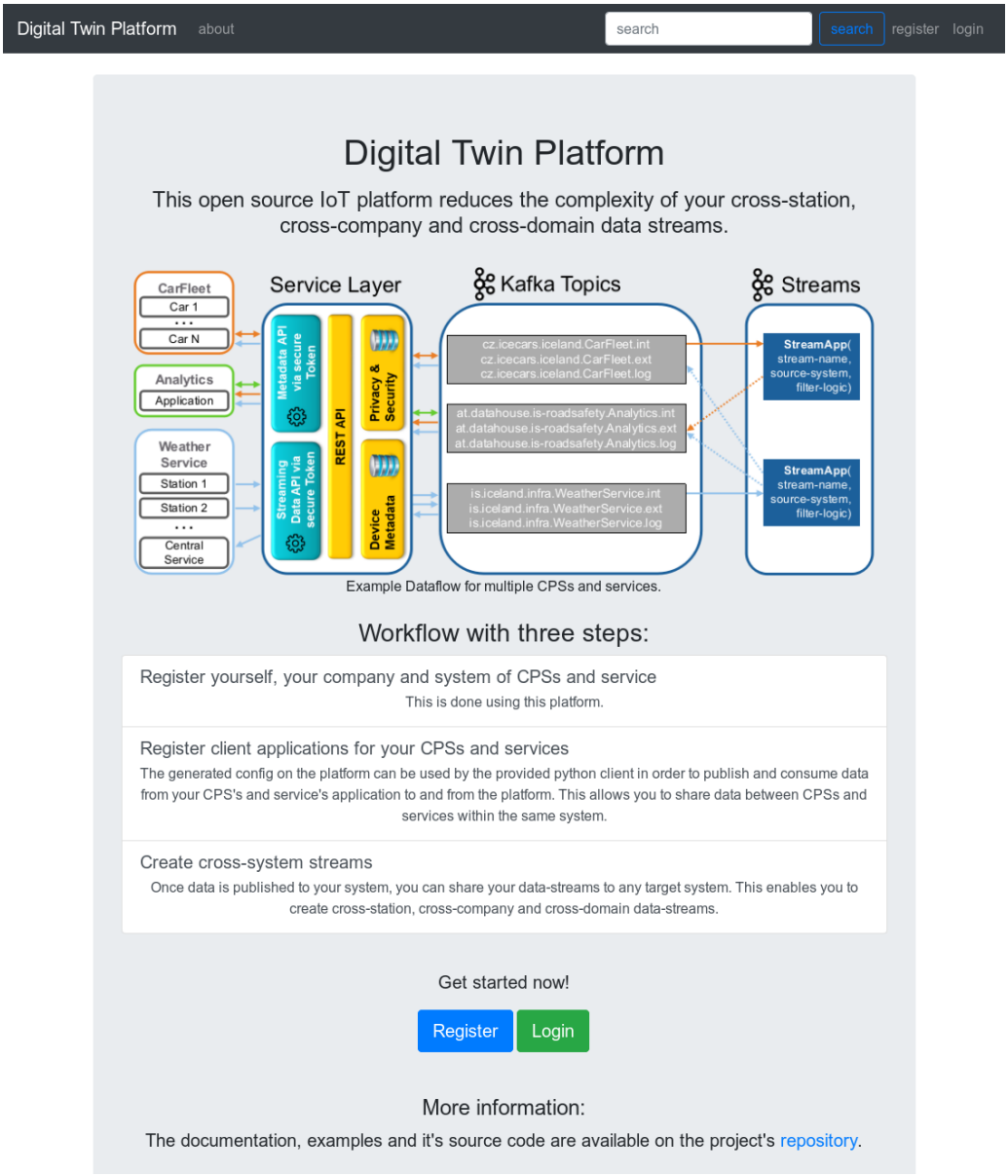

Figure 8: Welcome Screen and Introduction

From the welcome screen, the user can use the “register” button for self-registration to the system as shown in Figure 9. If the user is already registered, he or she can directly navigate to the login.

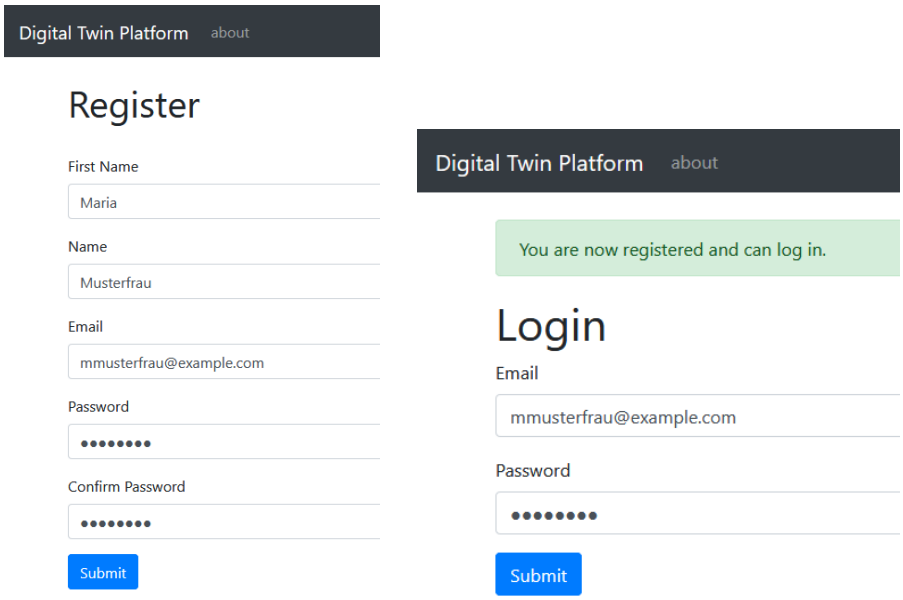

Figure 9: Self-Registration and Platform Login

Once the user is registered and logged in, the user will see an empty dashboard as shown in Figure 10, listing companies, systems, client applications and data streams accessible by the current user. Then the user has the option to add new instances, which is described in more detail in the next section.

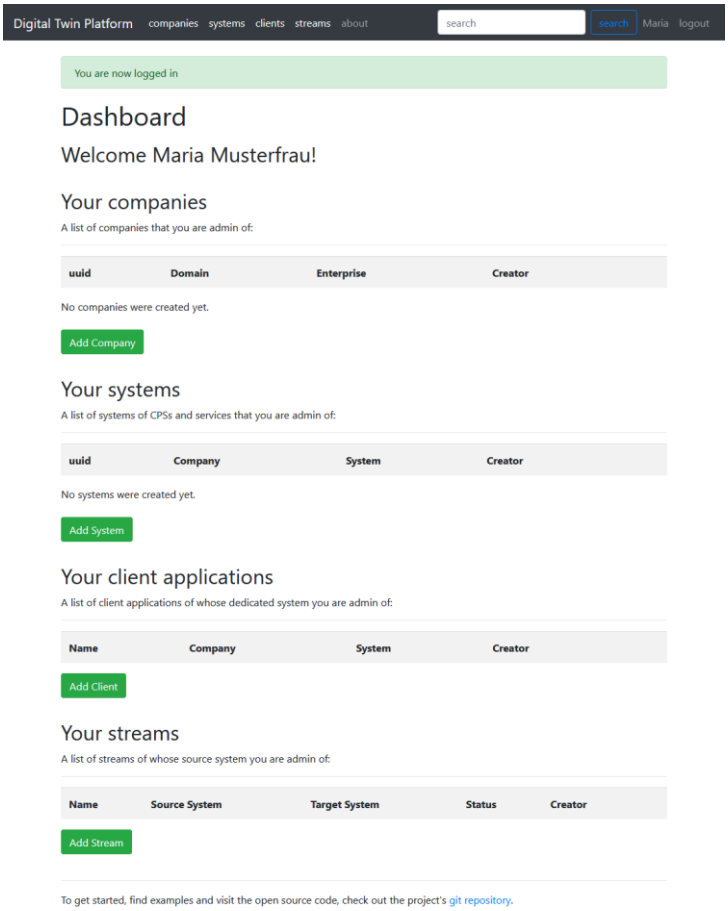

Figure 10: Initial, empty Dashboard

4.2 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 data streams between multiple systems.

4.2.1 Companies

The first step is to register a new company, which will be the owner by the cyber-physical systems created later. As shown in the screenshot in Figure 11, for demonstration purposes, a company is simply identified by a top-level domain, a short name and an optional description.

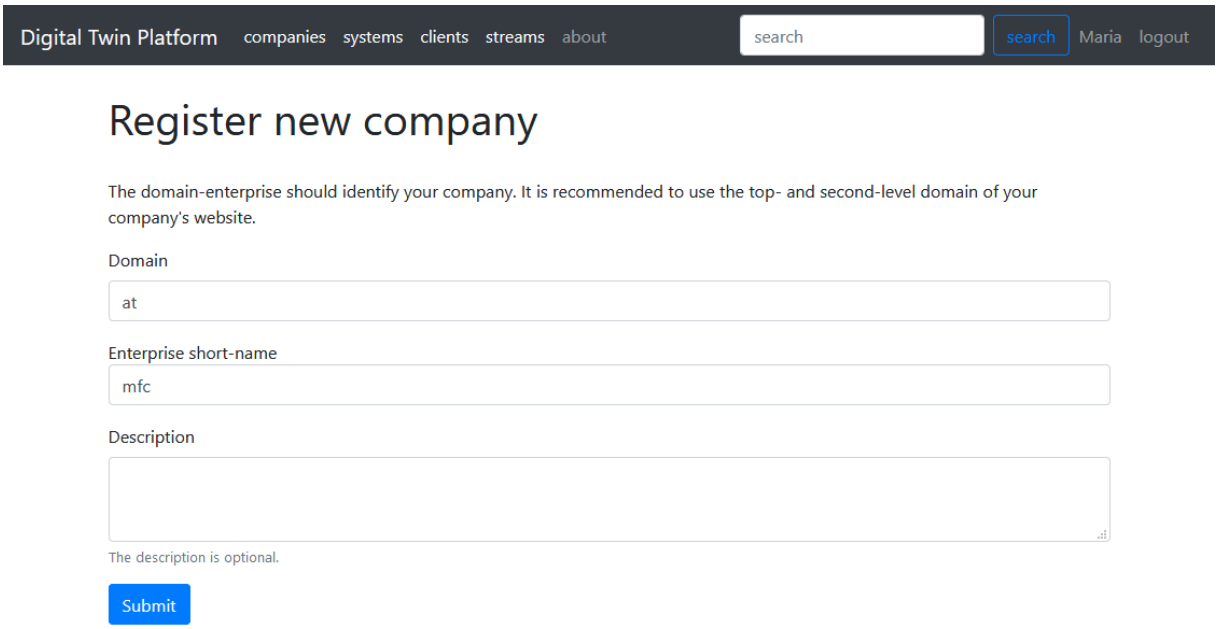

Figure 11: Register / create companies

A list of all companies that can be managed by the current user is shown in Figure 12, which currently contains exactly the just registered company. The blue button “manage company” leads to the next screen as illustrated in Figure 13, in which other company admins and systems can be added.

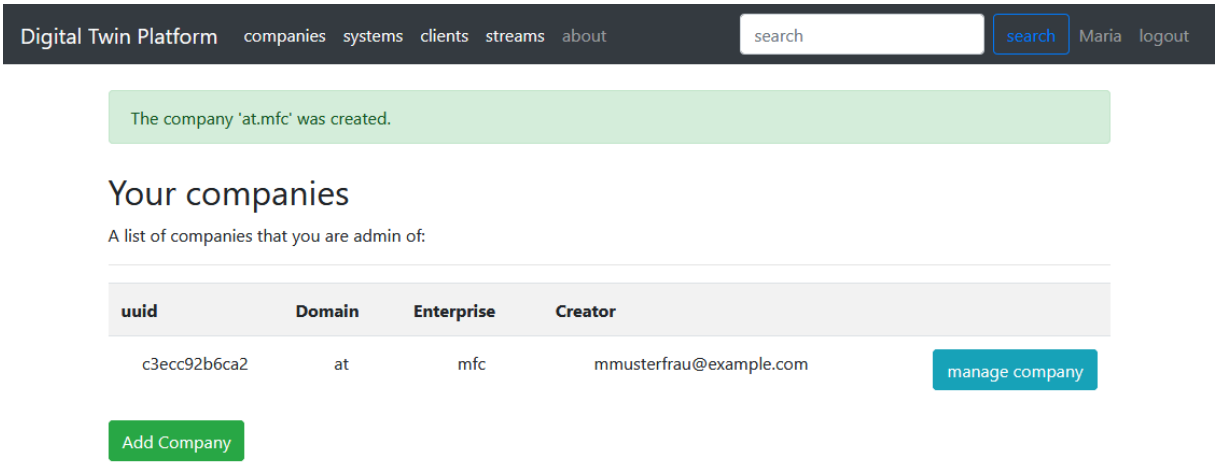

Figure 12: List companies

A company admin has the permission to create and delete systems as well as other company admins, if there are no registered systems in the company. Note that any admin of the company can delete the dedicated company.

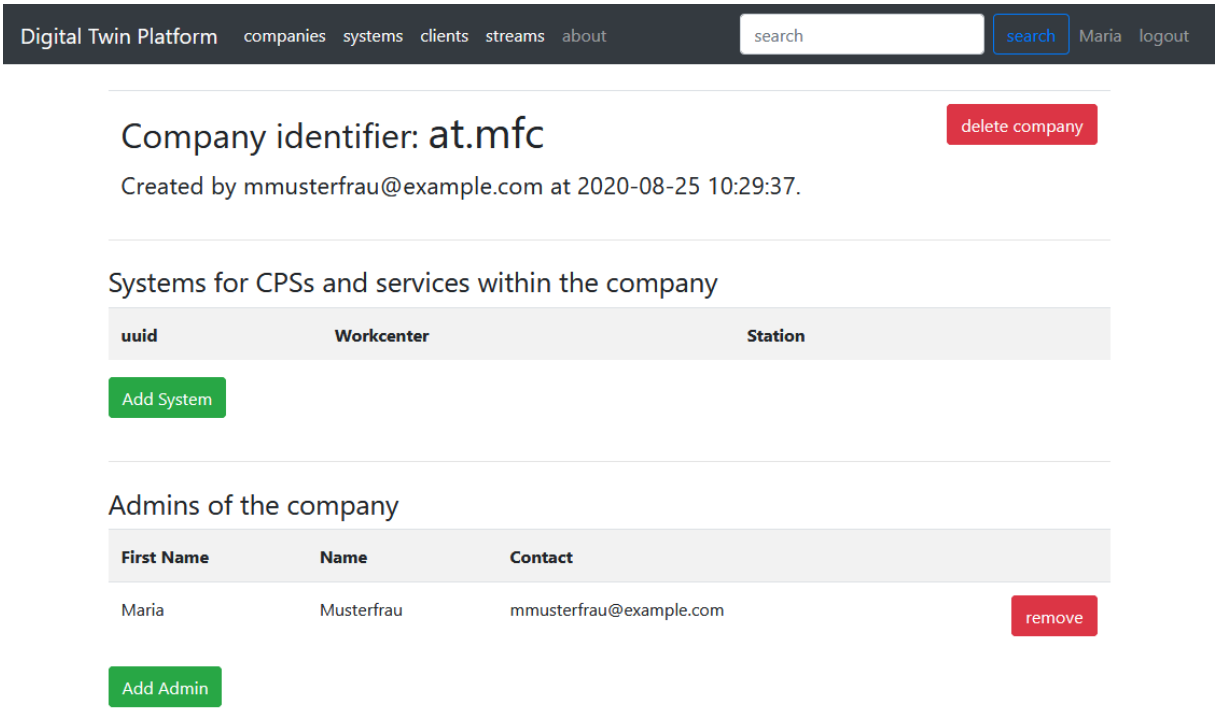

Figure 13: Show a company

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.

4.2.2 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 a 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 14 with the example of a station named “Station“ in the weather service.

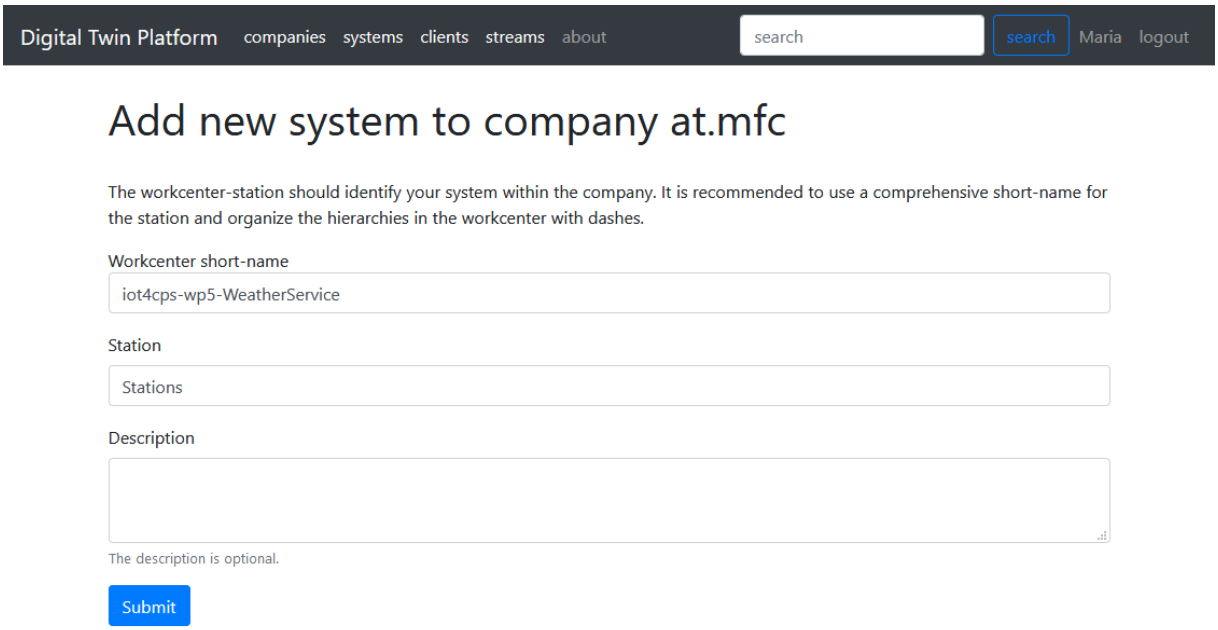

Figure 14: Add systems to a company

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 15, 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 single source system or one of multiple source systems, also a target system has to be specified.
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.

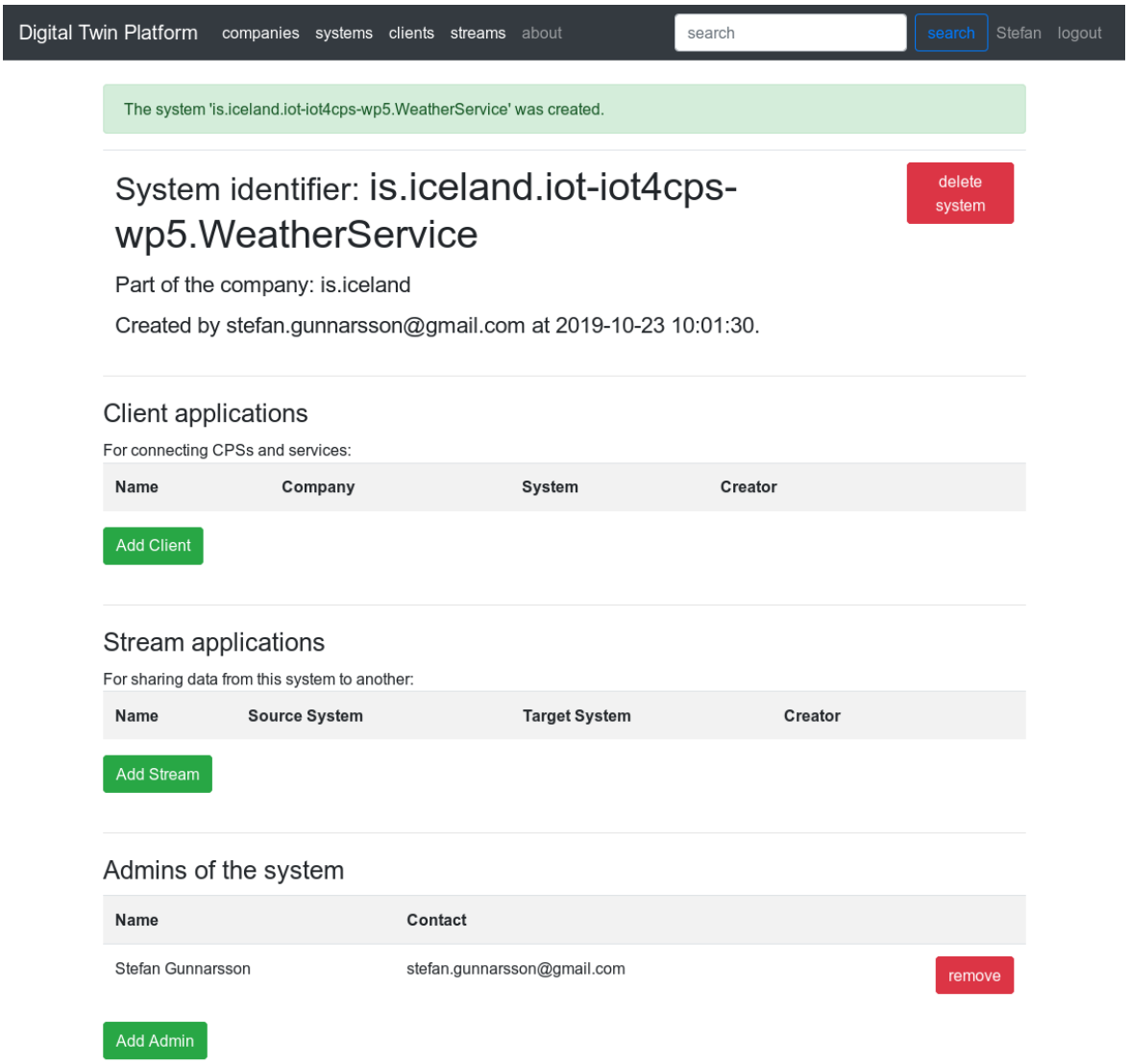

Figure 15: Show details for a newly registered system

4.3 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, as well as brake events including spatiotemporal information to nearby other cars. At the same time, it may receive external temperature and brake event data from other cars or weather stations in the proximity 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 16 for the system “at.mfc.iot4cps-wp5-WeatherService.Stations”.

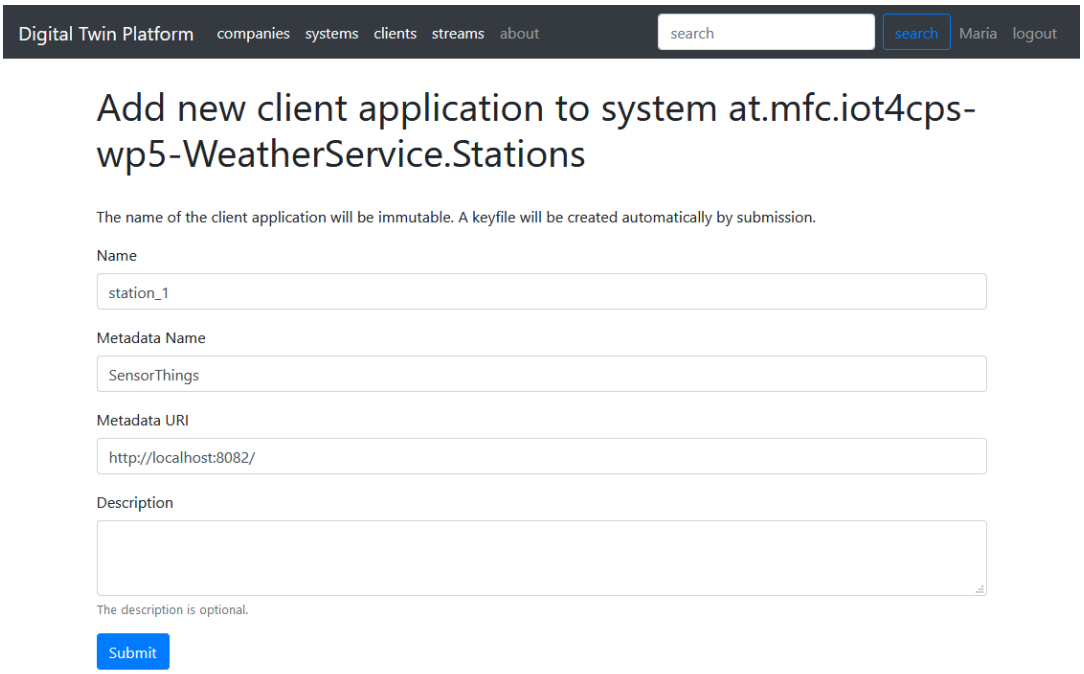

Figure 16: Register client applications within systems

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 17 below. In our example, we create a client application for a single weather station (“station_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 17 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 full usage of this key by the client application is not implemented yet. 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.

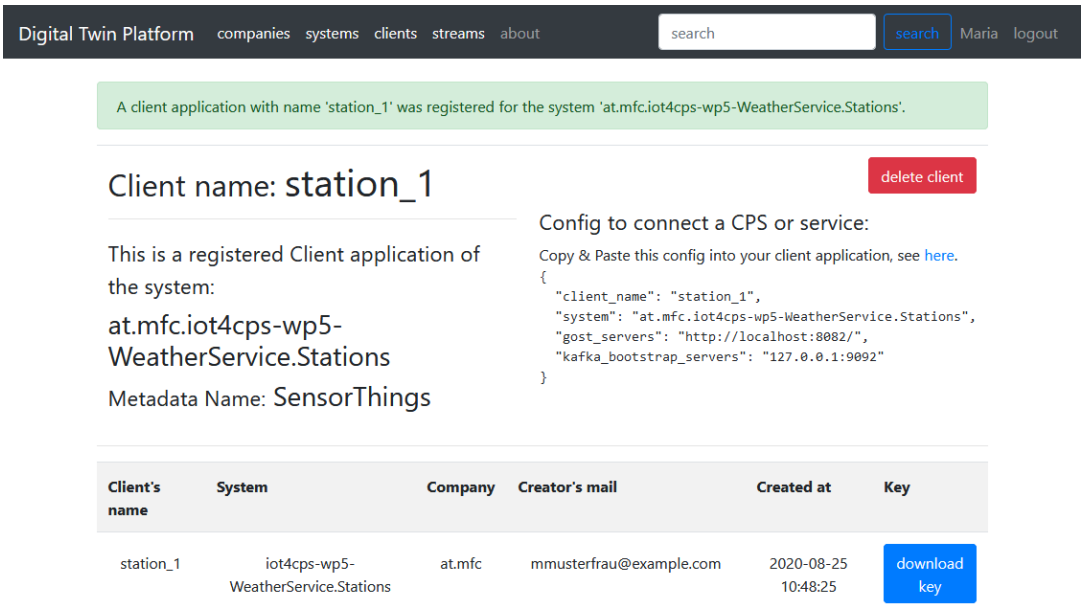

Figure 17: Manage a client application within a system

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 configuration object shown below needs to be passed as the only constructor parameter.

```
config = {"client_name": "station_1",
          "system": "at.mfc.iot4cps-wp5-WeatherService.Stations",
          "gost_servers": "localhost:8082",
          "kafka_bootstrap_servers": "localhost:9092"}
```

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 metadata information for the sensor data flow (as described in section 3.1.2). 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.

4.4 Streaming Applications

A streaming application enables the communication between a source and a target system. Once deployed, it consumes streaming data from one or multiple internal topics of the specified source system(s), processes the data, e.g. to only get warnings if some predefined thresholds are passed, and finally the resulting data (if not omitted) is produced into the external topic of the target system. Only streaming applications are permitted to publish data into external topics, and only client applications of the dedicated system can consume data from them.

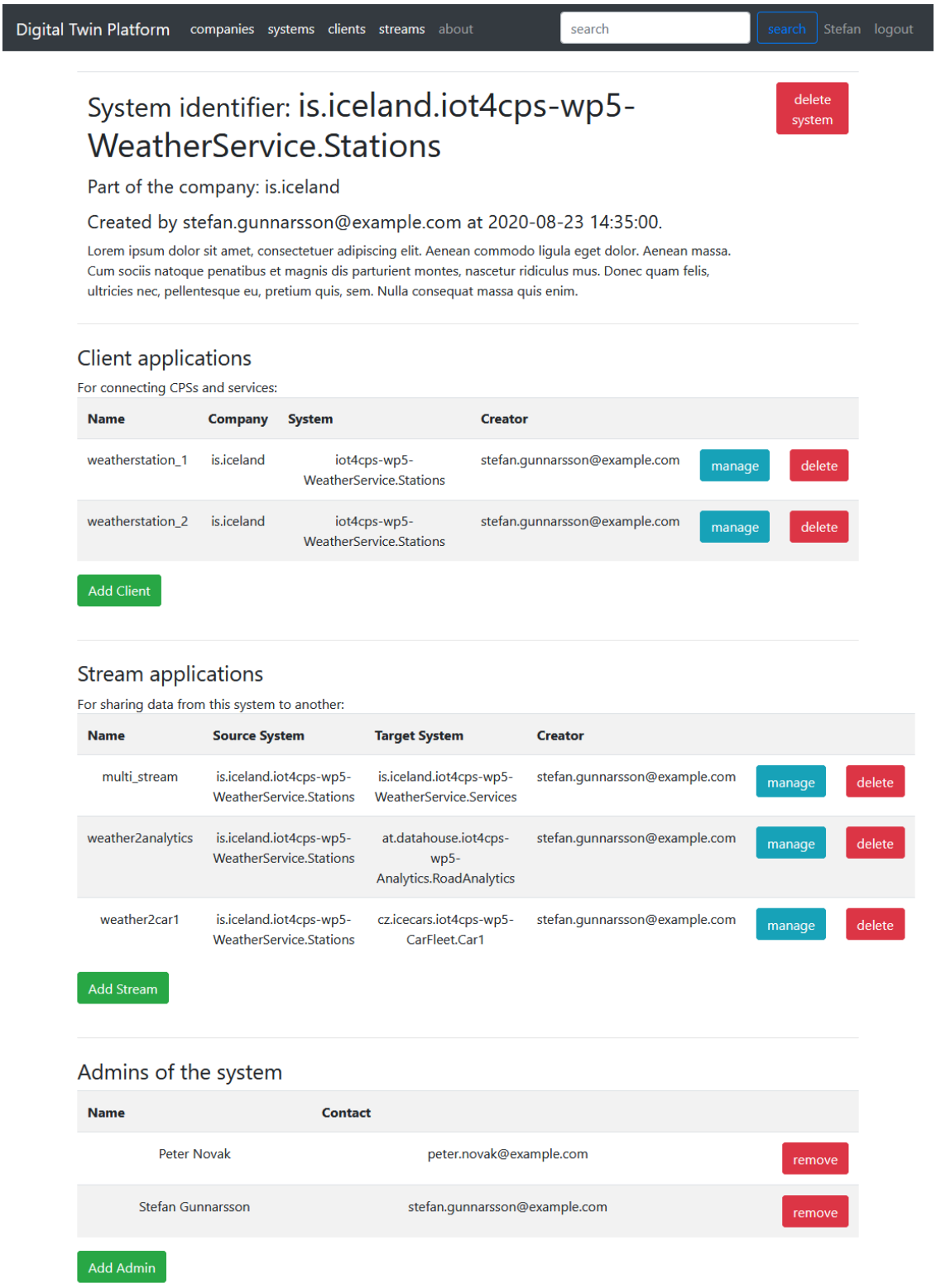

Figure 18: View on a system including two client applications and three streaming applications

Figure 18 shows an overview of a system of our demo scenario including two client applications for connecting CPSs, here the weather stations, three streaming applications and two system administrators. One streaming application is to distribute the weather information from the stations to the weather service, respectively to the analytics service, while the third one will send weather information to the system “Car1” if it is in the proximity of the station.

As already mentioned in section 3.4.3, two types of stream apps can be distinguished:

4.4.1 Single-Source Streaming Application

Single-Source Streaming Applications consume streaming data only from a single source system. This reduces the complexity of the required filter logic and therefore a short SQL-like filter logic suffices to customize most stream sharing 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, a new user with the name “Sue” is logged in. Here, the source system is the “Car 1” of the rental company and the target system will be the Road Analytics software of the “DataHouse” analytics company. Therefore, the stream is simply named “car1_to_analytics”.
This stream will forward data from the source system to the target system. Additionally, a filter logic is defined as shown in Figure 19, which can be considered as a description language for selecting and filtering streaming data. The default value is an empty clause which implies that any data from the source system is forwarded to the defined target system. In the depicted example each data is forwarded that is a temperature or an acceleration.

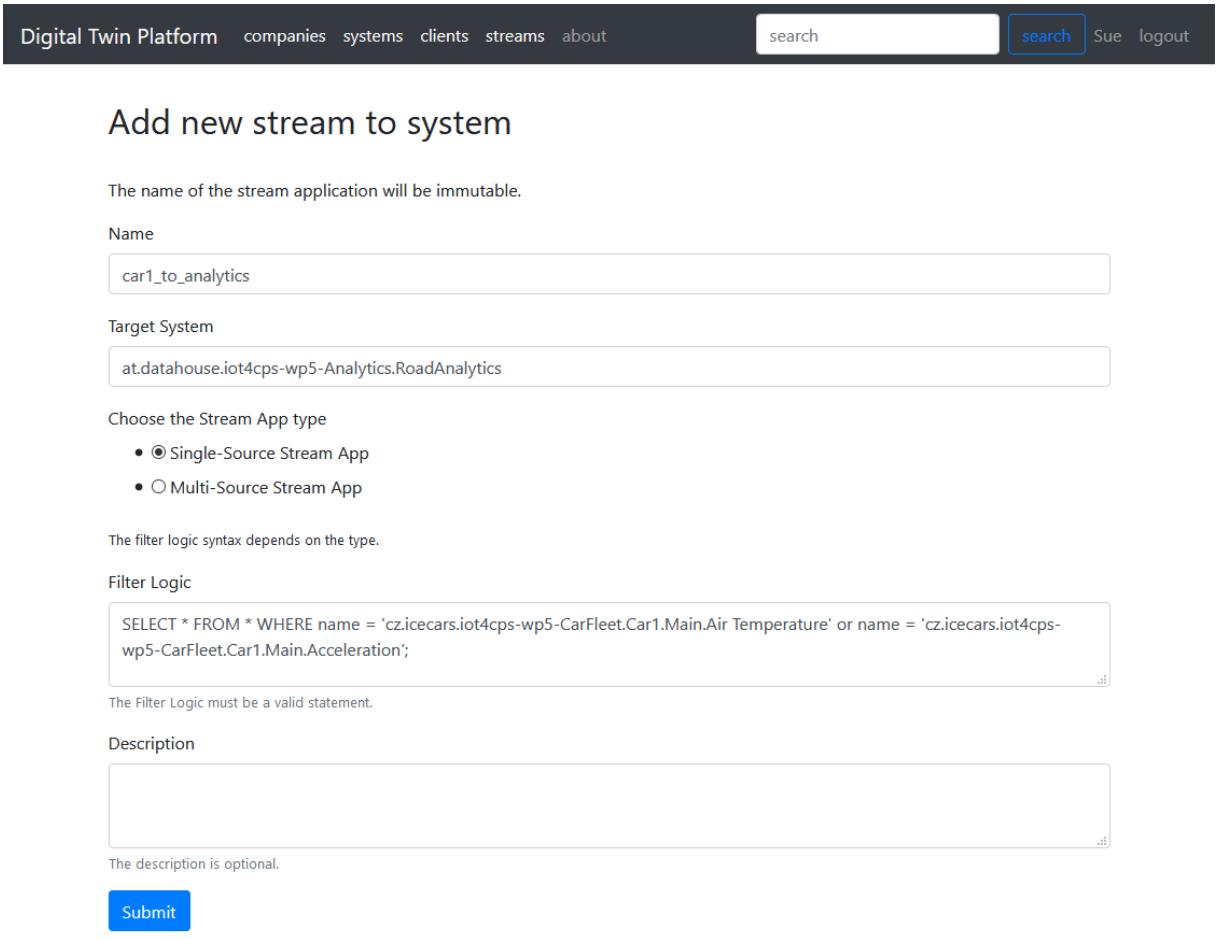

Figure 19: Creating a new Single-Source streaming application

In Figure 20, the view of the streaming application “car1_to_analytics” is shown. 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.

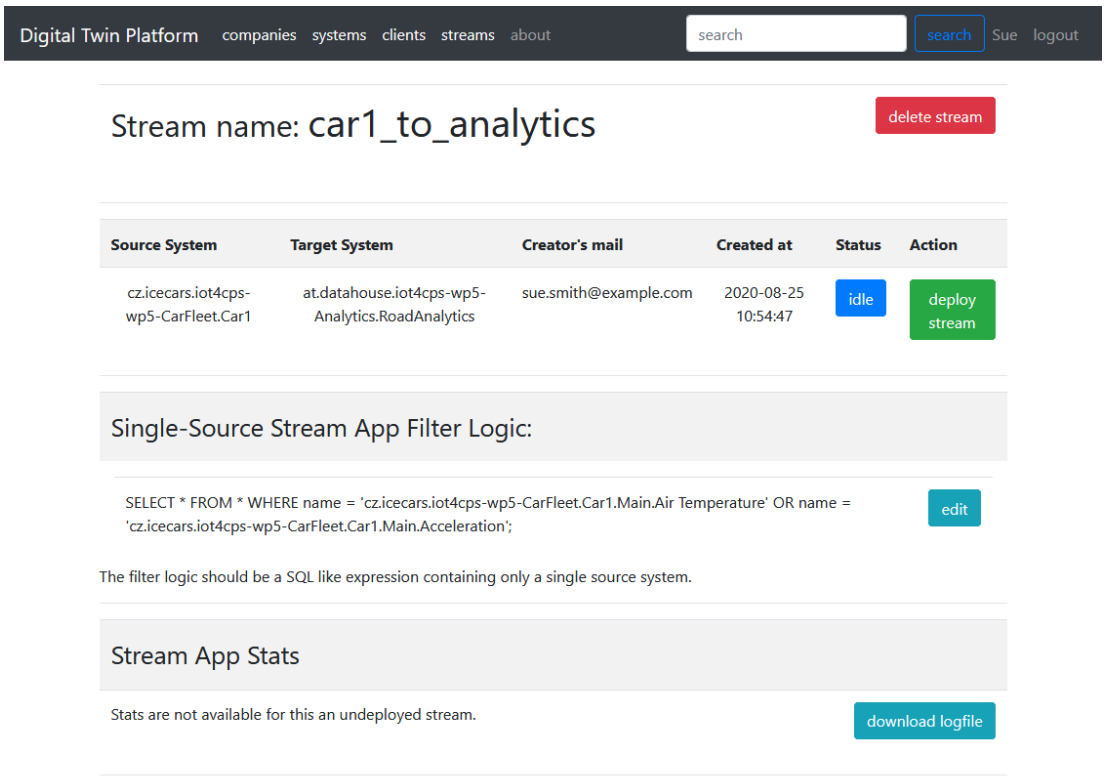

Figure 20: View of the streaming application "car1_to_analytics".

Finally, Figure 21 shows the deployed stream app that forwards selected quantities from the system “Car1” to the “Road Analytics”.

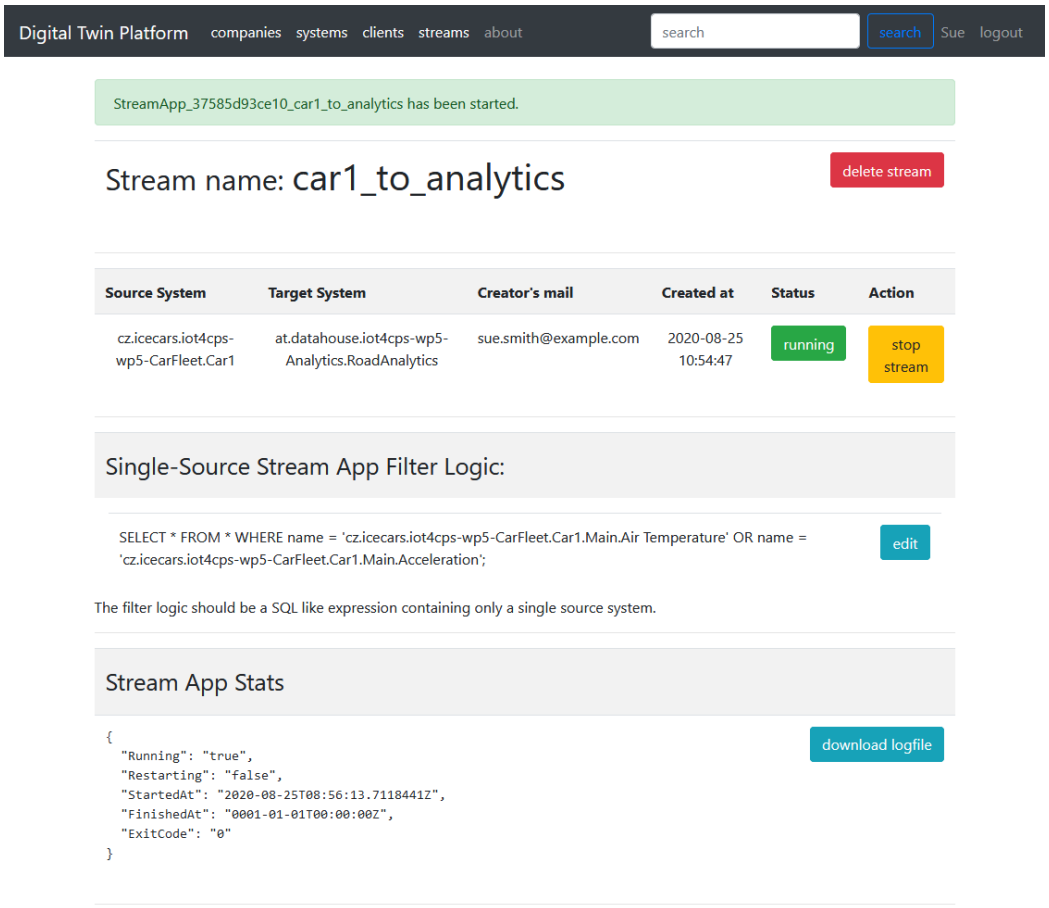

Figure 21: View of the deployed and successfully running streaming application "car1_to_analytics".

4.4.2 Multi-Source Streaming Application

From the user point of view, the Multi-Source variant is similar to the Single-Source Streaming Application, differing only in the fact that multiple quantities are consumed from one or two source systems, and more importantly two data streams are time-series joined to a new one. A lot of flexibility is achieved by customizing the stream join behaviour via defining arbitrary functions in the “filter_logic” field of the “add new stream” window. A created Multi-Source stream app could look like the example illustrated in Figure 22. Note that the filter logic is here a Python script that defines some constants and two functions, as described in section 3.4.2.

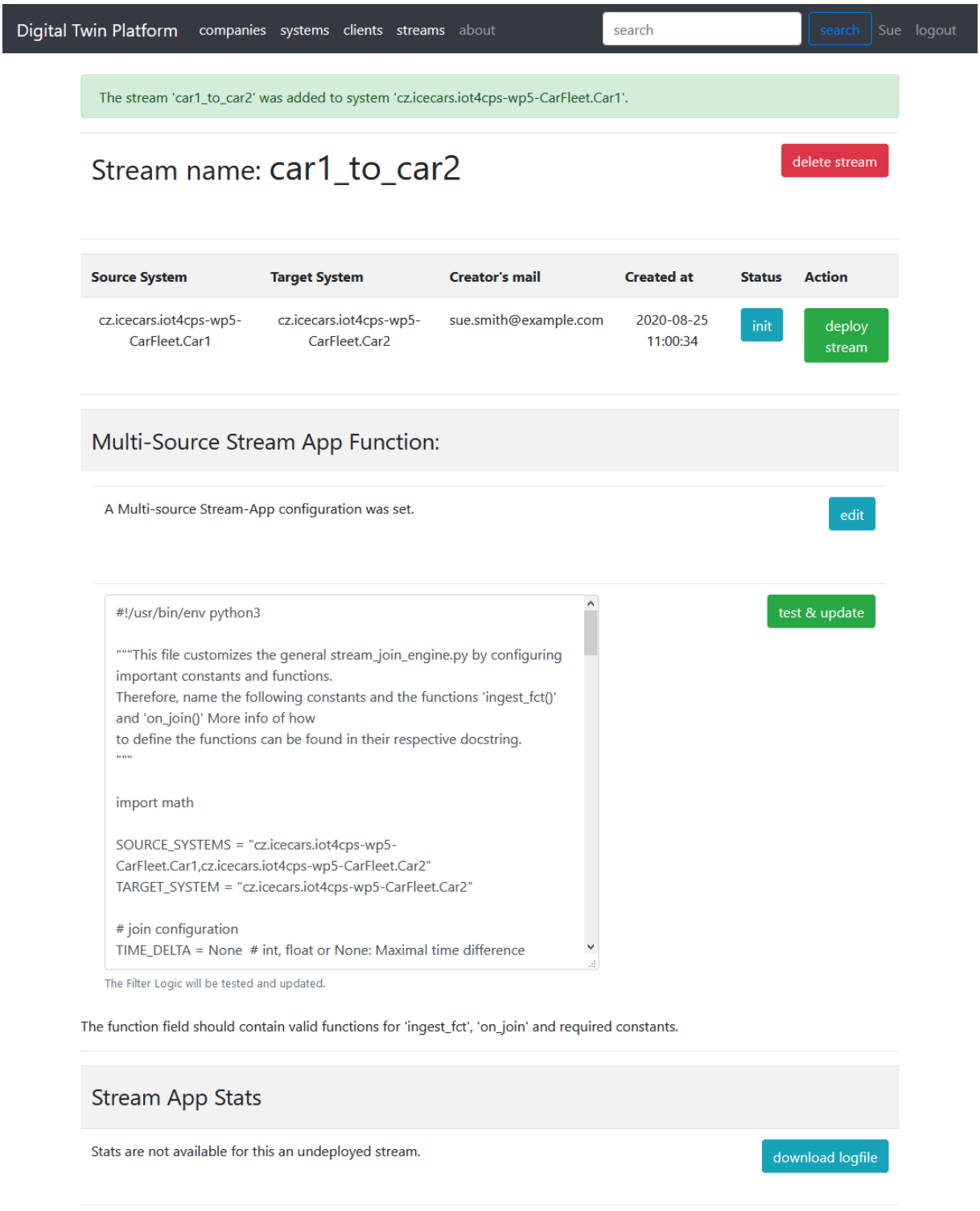

Figure 22: View of the Multi-Source streaming application "car1_to_car2".

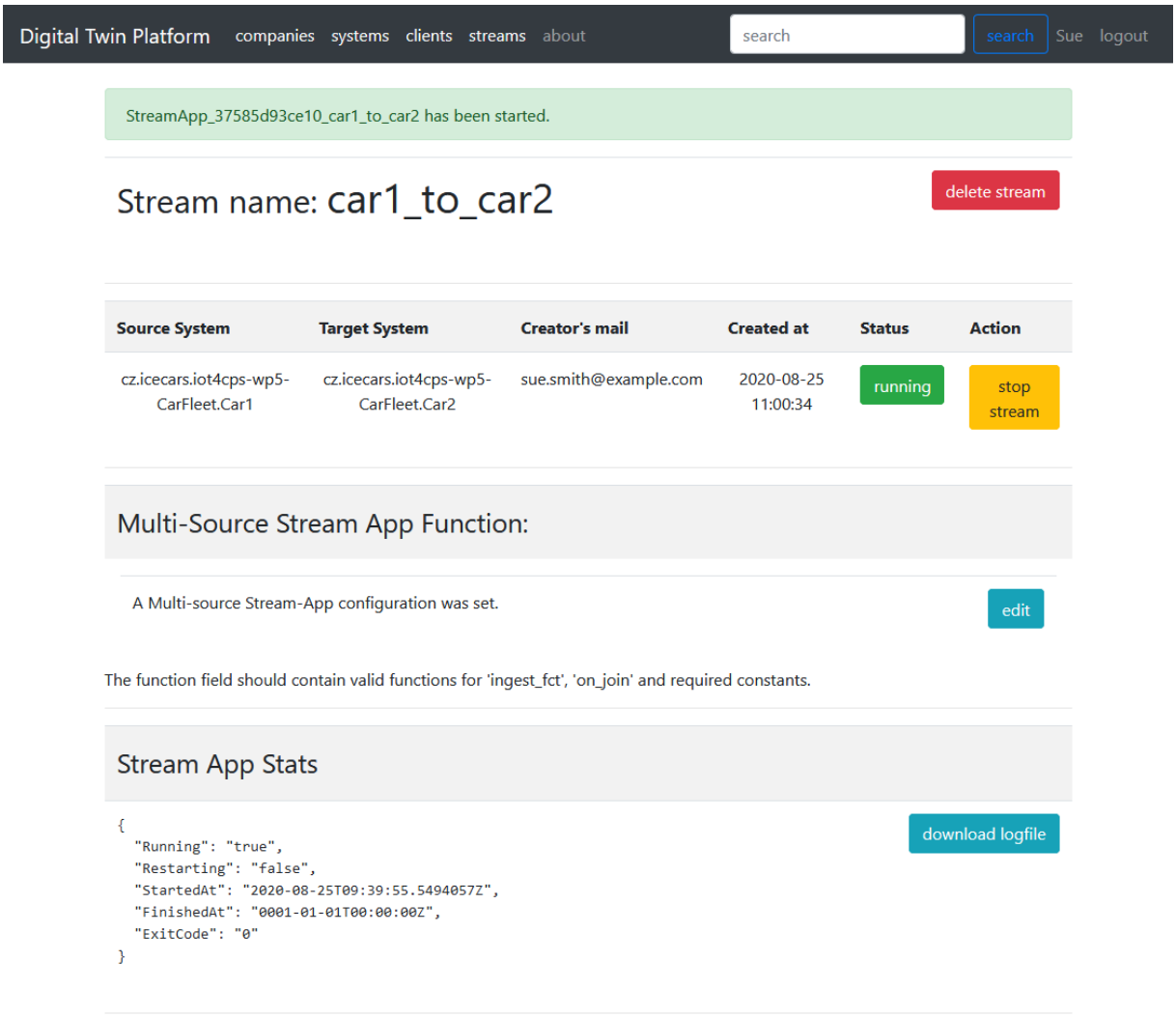

Figure 23: View of the deployed Multi-Source streaming application "car1_to_car2".

For both types of streaming applications, the filter logic can be edited and the log of the deployed stream app can be downloaded using the “download logfile” button. This helps to debug the application if necessary.

4.5 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 is used for storing the time-series data and Grafana 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 notebooks, which are browser applications that run code of various languages and use the rich data science packages provided by the Anaconda project. Figure 24 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 that can be deployed using a single command. More information about the setup can be found in Appendix A. Installation Guide and the referred repository.

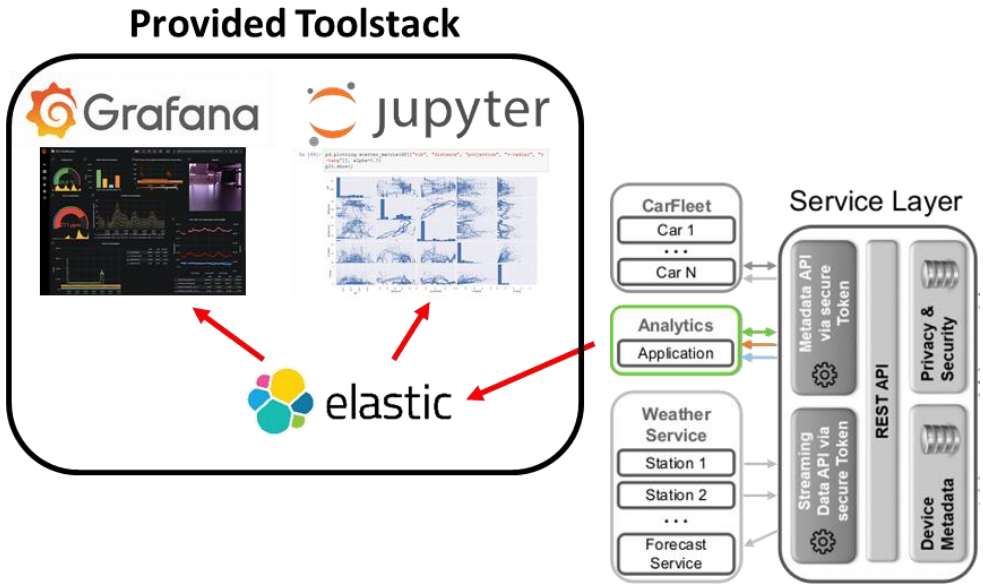

Figure 24: The Analytics sample code provides a Data Science toolstack.

5. Source Code and Current Status

The original source code of the platform has been hosted on a GitLab instance provided by AIT. However, to provide any reader of this public deliverable access to the open source code, a fork on GitHub has been created that is public and can be maintained by the community also after the runtime of the project.

Table 1: Current status per milestone.

Milestone	Status1
Use-Case	Finished
Architecture	Finished
Streaming Capability	Finished
Semantic	Finished. Potential improvements were found: 1. The primary key of instances within GOST's SensorThings implementation is named "iot.id" which is a serial number instead of an unique, human-readable identifier. This increases the complexity without a valid reason 2. The point in (1) implies that stream apps have to request the data stream name for "iot.id" on the GOST-server. This leads to a trade-off between frequently fetching the metadata that leads to higher latency and the rare updating that could lead to instability. Therefore, the usage of another Semantic server is suggested. In a follow-up project, the team of Salzburg Research develops an "Asset Registry", which is part of the "i-Asset Platform" and provides improved services integrating the concepts of the Asset Administration Shell from RAMI4.0 as well as other standards.[15]
Demo-Clients	Finished
Platform UI	Finished
Security	Finished to secure the platform and device communication via X-net's SBI-Box. The optional and additional usage of SSL/TLS encryption and authentication via a provided key is prepared but not implemented.
Stream apps	Finished the UI; The declarative language of single-source and multi-source Stream App Logics could be unified in order to improve the user experience, however, this goes beyond the scope of prototype implementation.
Dissemination	Finished

6. Conclusion

This deliverable documents the status of the Digital Twin Platform Prototype by August 2020. Based on a proof-of-concept use case it was demonstrated that the platform enables user and company registration, and that users can create client applications for exchanging data via data streaming applications. The platform itself and client applications were secured via X-Net's SBI Boxes. Additionally, two types of streaming applications feature the structured sharing of streaming data between clients and even complex data stream processing based on two data streams is possible. As the streaming applications are deterministic, process data exactly-once and have minimal latency, the safety issue illustrated in our demo use case could be solved.

[15] https://www.maintenance-competence-center.at/i-asset/plattform/

The final implementation of the prototype will be made available to a Github repository to make our Digital Twin Platform available to the open source community.

7. Appendix

7.1 Appendix A. Installation Guide

In this appendix, we provide the final version of the installation guide, which will also be published on GitHub.

7.1.1 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.

7.1.2 Setup Apache Kafka and its library

The easiest way to set up a cluster for Apache Kafka is via Docker and docker-compose. Deploy each three instances of Kafka and its underlying Zookeeper on the same node via:

```
cd setup/kafka
docker-compose up -d
```

The flag -d stands for daemon mode. The containers can be investigated (stopped) via `docker-compose logs (down)`. Three instances of Kafka are then available, each on the ports 9092, 9093 and 9094. Therefore, a replication factor of up to three is possible using this setup.

In case one doesn't want to install Kafka via Docker (as it is suggested for production), the installation can also be done directly. The Datastack uses Kafka **version 2.3.1** as the communication layer, the installation is done in `/kafka`.

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

Then, start Zookeeper and Kafka and test the installation: (If the setup was done using Docker one can skip the Start-step)

```
# 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
```

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

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

```
# 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
>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 for the platform using:

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

If multiple topics were generated, everything worked well.

7.1.3 Setup SensorThings Server (GOST) to add semantics

The SensorThings server (GOST) is set up using Docker, using:

```
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:8082/ or view the logs:

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

7.1.4 Start Demo Applications

Now, open new terminals to run the demo applications:

CarFleet – Prosumer

Start the Car Simulator via:

```
python3 demo_applications/CarFleet/Car1/car_1.py
> INFO:PR Client Logger:init: Initialising Digital Twin Client with
name 'client' on 'cz.icecars.iot4cps-wp5-CarFleet.Car1'
....
> The demo car 1 is at [47.822495, 13.04113], with the temp.: 3.059
°C  and had a maximal acceleration of 0.028 m/s² at 2020-08-
25T09:13:06.627023+00:00


python3 demo_applications/CarFleet/Car2/car_2.py
> INFO:PR Client Logger:init: Initialising Digital Twin Client with
name 'client' on 'cz.icecars.iot4cps-wp5-CarFleet.Car2'
...
> The demo car 2 is at [47.804533, 13.044287], with the temp.: 2.559
°C and had a maximal acceleration of 0.028 m/s² at 2020-08-
25T09:13:26.999313+00:00
> The demo car 2 is at [47.79771, 13.031169],  with the temp.: 3.53
°C  and had a maximal acceleration of 0.55 m/s² at 2020-08-
25T09:13:32.712785+00:00
  -> Received new external data-point from a nearby car 2020-08-
23T19:36:30.295946+00:00: 'temperature' = -8.226 degC, measured 0.34
km away.
  -> Received new external data-point from a nearby car 2020-08-
23T19:36:40.339215+00:00: 'temperature' = 1.006 degC, measured 0.20
km away.
```

Note that external messages can only be received, if a corresponding streaming application is running. For testing and debugging, one can start stream apps manually in the respective subprojects *"server/StreamHub/src/main/java/com/github/christophschranz/iot4cpshub/StreamAppEngine.java"* for Single-Source and *"server/TimeSeriesJoiner/stream_join_engine.py"* for Multi-Source stream apps.

WeatherService – Producer

The scripts for the demo weather service are:

```
cd demo_applications/WeatherService
python3 demo_station_1/demo_station_1.py
python3 demo_station_2/demo_station_2.py
python3 forecast_service/forecast-service.py
```

Here, you should see that temperature data is produced by the demo stations and consumed only by the central service, if and only if a corresponding stream application is started, that consumes data from the System "Stations" and produces them to the system "Services". A Single-Source stream app is suggested to do this.

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:

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

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.

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

7.1.5 Streaming Applications

As there are two different types of stream apps that are based on different technologies, we have to distinguish:

Single-Source streaming applications

Single-Source streaming applications are implemented in Java using the Kafka Streams library. A pre- built jar file to share data from a specific tenant to others can be started with:

```
java -jar server/StreamHub/target/streamApp-1.1-jar-with-
dependencies.jar --stream-name mystream --source-system
is.iceland.iot4cps-wp5-WeatherService.Stations is.iceland.iot4cps-
wp5-WeatherService.Services --filter-logic "SELECT * FROM * WHERE
result < 4;" --bootstrap-server 127.0.0.1:9092
```

If you want to change the streamhub application itself, modify and rebuild the java project in `server/StreamHub/src/main/java/com/github/christophschranz/iot4cpshub/StreamAppEngine.java`.
It is recommended, to start and stop the streaming applications via the Platform UI, that provides the same functionality as the command line interface.

Multi-Source streaming applications

Multi-Source streaming applications are implemented in Python, plain Apache Kafka and is based on the Time-Series join that is implemented using the *LocalStreamBuffer* algorithm. The stream app can be started using:

```
python3 server/TimeSeriesJoiner/stream_join_engine.py
```

This script uses the customization set in `server/TimeSeriesJoiner/customization/custom_fct.py` which contains all required constants and two functions "ingest_fct" and "on_join" that suffice to customize the stream app's behaviour. For more information read the README file in the `server/TimeSeriesJoiner/` sub-project. In the Appendix C: Custom Functions for a Multi-source StreamApp an exemplary `custom_fct.py` -file is presented.

7.1.6 Track what happens behind the scenes:

Check the created kafka topics:

```
/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --list
cz.icecars.iot4cps-wp5-CarFleet.Car1.ext
cz.icecars.iot4cps-wp5-CarFleet.Car1.int
cz.icecars.iot4cps-wp5-CarFleet.Car1.log
…
```

Note that kafka-topics must be created in advance as explained in the setup.

To track the traffic in near 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
```

```
> {"phenomenonTime":"2020-08-25T12:31:47.864597+00:00",
"resultTime":"2020-08-25T12:31:47.885536+00:00",
"Datastream":{"@iot.id":2}, "longitude":13.009834,
"latitude":47.799216, "attitude":425.870343, "result":0.695}
```

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, one can 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`.

7.1.7 Deployment on a Cluster

For a production deployment of the messaging system, we recommend to set up 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 under *“setup/README-Deployment.md”*.

7.1.8 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].

```
sudo apt install libpq-dev
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt
$(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc
| sudo apt-key add -
sudo apt-get update
sudo apt-get install postgresql

sudo -u postgres psql
postgres=# CREATE ROLE iot4cps LOGIN PASSWORD 'iot4cps';
postgres=# CREATE DATABASE iot4cps OWNER iot4cps;

# Start of the platform
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.

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

7.2 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:
        Connected cars want to enhance their safety by retrieving temperature, acceleration
        and position data from each, to warn the drivers on approaching dangerous road
        sections. As each car measures these quantities by
        themselves, they are shared to others via the platform.
   Analytics:
        A provider of applied data analytics with the goal to improve the road quality.
        Therefore, data of various sources are consumed.
   WeatherService:
        A Weather Service provider that conducts multiple Stations that measure weather
        conditions, as well as a central service to forecast the Weather. Additionally, the
        temperature data is of interest for the CarFleet and therefore shared with them.
"""

import os
import time
import pytz
import threading
from datetime import datetime

from client.digital_twin_client import DigitalTwinClient
from demo_applications.simulator.CarSimulator import CarSimulator

# load files relative to this file
dirname = os.path.dirname(os.path.abspath(__file__))
INSTANCES = os.path.join(dirname, "instances.json")
SUBSCRIPTIONS = os.path.join(dirname, "subscriptions.json")


def produce_metrics(interval=10):
   while not halt_event.is_set():
        # unix epoch and ISO 8601 UTC are both valid
        timestamp = datetime.utcnow().replace(tzinfo=pytz.UTC).isoformat()

        # Measure metrics
        temperature = car.temp.get_temp()
        acceleration = car.get_acceleration()
        latitude = car.get_latitude()
        longitude = car.get_longitude()
        attitude = car.get_attitude()

        # Print the temperature with the corresponding timestamp in ISO format
        print(f"The demo car 1 is at [{latitude}, {longitude}],   \twith the temp.:
{temperature} °C  \tand had a " +
              f"maximal acceleration of {acceleration} m/s²  \tat {timestamp}")

        # Send the metrics via the client, it is suggested to use the same timestamp for later
analytics
        client.produce(quantity="temperature", result=temperature, timestamp=timestamp,
                       longitude=longitude, latitude=latitude, attitude=attitude)
        client.produce(quantity="acceleration", result=acceleration, timestamp=timestamp,
                       longitude=longitude, latitude=latitude, attitude=attitude)

        time.sleep(interval)
```

```
# Receive all temperatures of the weather-service and other cars and check whether they are
subzero
def consume_metrics():
    while not halt_event.is_set():
        # In this list, each datapoint is stored that is below zero degC.
        subzero_temp = list()

        # Data of the same instance can be consumed directly via the class method
        temperature = car.temp.get_temp()
        if temperature < 0:
            subzero_temp.append({"origin": config["system"], "temperature": temperature})

        # Data of other instances (and also the same one) can be consumed via the client
        received_quantities = client.consume(timeout=0.1)
        for received_quantity in received_quantities:
            # The resolves the all meta-data for an received data-point
            print(f"  -> Received new external data-point from
{received_quantity['phenomenonTime']}: "
                  f"'{received_quantity['Datastream']['name']}' = {received_quantity['result']}
"
                  f"{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["Datastream"]["unitOfMeasurement"]["symbol"] == "degC" \
                    and received_quantity["result"] < 0:
                subzero_temp.append(
                    {"origin": received_quantity["Datastream"]["name"], "temperature":
received_quantity["result"]})

        # # Check whether there are temperatures are subzero
        # if subzero_temp != list():
        #     print("    WARNING, the road could be slippery, see: {}".format(subzero_temp))


if __name__ == "__main__":
    # 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": "client",
              "system": "cz.icecars.iot4cps-wp5-CarFleet.Car1",
              "gost_servers": "localhost:8082",
              "kafka_bootstrap_servers": "localhost:9092",
              "additional_attributes": "longitude,latitude,attitude"}
    client = DigitalTwinClient(**config)
    client.logger.info("Main: Starting client.")
    client.register(instance_file=INSTANCES)  # Register new instances could be outsourced to
the platform
    client.subscribe(subscription_file=SUBSCRIPTIONS)  # Subscribe to data streams

    # Create an instance of the CarSimulator that simulates a car driving on different tracks
through Salzburg
    car = CarSimulator(track_id=1, time_factor=100, speed=30, cautiousness=1,
                       temp_day_amplitude=4, temp_year_amplitude=-4, temp_average=3, seed=1)
    client.logger.info("Main: Created instance of CarSimulator.")

    client.logger.info("Main: Starting producer and consumer threads.")
    halt_event = threading.Event()

    # Create and start the receiver Thread that consumes data via the client
    consumer = threading.Thread(target=consume_metrics)
    consumer.start()
    # Create and start the receiver Thread that publishes data via the client
    producer = threading.Thread(target=produce_metrics, kwargs=({"interval": 10}))
    producer.start()
```

```
# set halt signal to stop the threads if a KeyboardInterrupt occurs
try:
    while True:
        time.sleep(1)
except (KeyboardInterrupt, SystemExit):
    client.logger.info("Main: Sent halt signal to producer and consumer.")
    halt_event.set()
    # wait for the threads to get finished (can take about the timeout duration)
    producer.join()
    consumer.join()
    client.logger.info("Main: Stopped the demo applications.")
    client.disconnect()
```

7.3 Appendix C: Custom Functions for a Multi-source StreamApp

In this section an exemplary file is depicted that holds all necessary information to define the behaviour of a Multi-source StreamApp:

```
#!/usr/bin/env python3
# custom_fct.py

"""This file customizes the general stream_join_engine.py by configuring important constants and functions.
Therefore, name the following constants and the functions 'ingest_fct()' and 'on_join()' More info of how
to define the functions can be found in their respective docstring.
"""

import math

# Kafka configuration
KAFKA_BOOTSTRAP_SERVERS = "localhost:9092"  # kafka nodes of the form 'mybroker1,mybroker2'
# declare one or two Kafka Topics to consume from
KAFKA_TOPICS_IN = ["cz.icecars.iot4cps-wp5-CarFleet.Car1.int", "cz.icecars.iot4cps-wp5-CarFleet.Car2.int"]
KAFKA_TOPIC_OUT = "cz.icecars.iot4cps-wp5-CarFleet.Car2.ext"

# join configuration
TIME_DELTA = None  # int, float or None: Maximal time difference between two Records being joined
ADDITIONAL_ATTRIBUTES = "longitude,latitude,attitude"  # optional attributes in observation records "att1,att2,..."
USE_ISO_TIMESTAMPS = True  # boolean: timestamp format of the resulting records, ISO 8601 or unix timestamp if
False
MAX_BATCH_SIZE = 100  # consume up to this number of messages at once
TRANSACTION_TIME = 1  # time interval  for committing transactions
VERBOSE = True


# ingest routine for record into the StreamBuffer instance
def ingest_fct(record, stream_buffer):
  """Ingestion function. Defines how a record is ingested into the stream_buffer, i.e., specifies under which
  constraints a record is ingested into the buffer's left or right buffer.

  Use the stream_buffer methods 'ingest_left(record)' and 'ingest_right(record)', as well as the record's getter
  functions:
    * 'get_quantity()' get the quantity name if set
    * 'get("topic")' get Kafka's topic name
    * 'get("thing")' get the thing name if set

  :param record: Record
    Record instance defined in local_stream_buffer.py
  :param stream_buffer: LocalStreamBuffer
    Instance of the LocalStreamBuffer, allows to ingest left and right join partners, joins them automatically
  :return: None
  """
  # ingest into left buffer, if the records was consumed from system Car 1
  if record.get("topic") == "cz.icecars.iot4cps-wp5-CarFleet.Car1.int":
      stream_buffer.ingest_left(record)  # with instant emit
  # ingest into left buffer, if the records was consumed from system Car 2
  elif record.get("topic") == "cz.icecars.iot4cps-wp5-CarFleet.Car2.int":
      stream_buffer.ingest_right(record)


def on_join(record_left, record_right):
  """Procedure on a join event.
  This function customizes the behaviour on a join event. It receives two Records, one that is a left join partner
  and one right.
  :param record_left:
  :param record_right:
```

```
    :return: dictionary, None
      If no join should be done return None. Else, return a dictionary containing the mandatory keys "quantity",
      "result" and "phenomenonTime". It is allowed to use more keys.
    """
    # calculate the relative distance between the cars from the given GPS coordinates based on a spherical approach.
    # This solution is even correct for large distances. The distance is measured in kilometers.
    k = math.pi/180
    distance = 6378.388 * math.acos(
        math.sin(k * record_left.get("latitude")) * math.sin(k * record_right.get("latitude")) +
        math.cos(k * record_left.get("latitude")) * math.cos(k * record_right.get("latitude")) *
        math.cos(k * (record_right.get("longitude") - record_left.get("longitude"))))
    # # the above solution is for small distances similar than the one below, but the later is better understandable
    # dx = 111.3 * math.cos(k * (record_left.get("latitude") + record_right.get("latitude")) / 2) * \
    #      (record_right.get("longitude") - record_left.get("longitude"))
    # dy = 111.3 * (record_left.get("latitude") - record_right.get("latitude"))
    # # distance = math.sqrt(dx * dx + dy * dy)
    # print(f"Distances: {distance} -- {math.sqrt(dx * dx + dy * dy)}")
    # # more information for calculating distances based on coordinates are here: www.kompf.de/gps/distcalc.html

    if distance < 1.23:  # distance is lesser than the set distance in kilometers
        record_dict = dict({"thing": record_left.get("thing"),
                            "quantity": record_left.get("quantity"),
                            "result": record_left.get_result(),
                            "phenomenonTime": record_left.get_time(),
                            # often the mean is used: (record_left.get_time() + record_right.get_time()) / 2
                            "longitude": record_left.get("longitude"),
                            "latitude": record_left.get("latitude"),
                            "attitude": record_left.get("attitude"),
                            "rel_distance": distance})
        return record_dict

    elif VERBOSE:
        print(f"The relative distance of {distance:.3f} km between the cars exceeds the maximal allowed distance.")
        return None
```