Situation-Aware Linked heTerogeneous Enriched Data

D2.2: Report on data modelling and linking

Work package	WP 2
Task	Task 2.2
Due date	31/03/2023
Submission date	14/03/2023
Deliverable lead	UC
Version	1
Authors	Víctor González (UC), Laura Martín (UC), Jorge Lanza (UC), Juan Ramón Santana (UC), Pablo Sotres (UC), Maren Dietzel (Kybeidos), Amir Reza Jafari Tehrani (IMT), Benjamin Hebgen (NEC), Gürkan Solmaz (NEC)
Reviewers	Anja Summa (Kybeidos) Luis Sánchez (UC)

Abstract	This document, developed by the SALTED project, represents the D2.2 deliverable of the data modelling and linking. The focus of this document is the thorough definition of the architecture implementation and deployment, including details about how security is handled. Furthermore, D2.2 includes an update on the data modelling introduced in D1 and D2.1, and a review of the linking

This project is co-financed by the Connecting Europe Facility of the European Union under the Action Number 2020-EU-IA-0274.

This project is co-financed by the Connecting Europe Facility of the European Union under the Action Number 2020-EU-IA-0274.

	modules currently implemented.
Keywords	Data Modelling, Data Linking, Architecture, Security, Implementation

This project is co-financed by the Connecting Europe Facility of the European Union under the Action Number 2020-EU-IA-0274.

Table of Contents

This project is co-financed by the Connecting Europe Facility of the European Union under the Action Number 2020-EU-IA-0274.

Table of Figures

1 INTRODUCTION

1.1 SCOPE OF DOCUMENT

The main goal of the SALTED project is to add value to existing data by enriching them, and subsequently publish the enriched data in the European Data Portal (EDP) using the NGSI-LD information model. This document D2.2 focuses on the implementation and current deployment status of the SALTED architecture defined in D2.1 [1], adding key details about the federated setup and the security layer.

The document also describes the implementation of the Data Enrichment Toolchain (DET) components related to the data modelling (i.e. Injection Chains) and data linking (i.e. Linkers) functionalities, which are the main focus of Task 2.2.

1.2 TARGET AUDIENCE

This document is mainly intended for internal use, although it is publicly available in order to raise awareness of the SALTED project and its relation with the FIWARE ecosystem. The target audience is the SALTED technical team including all partners involved in the delivery of Work Packages 1, 2 and 3. It also serves as reference for the developers of the situation-aware applications in Work Package 4. The document provides a thorough review of the key functionalities that the SALTED architecture and DET currently support, how data is processed, and how security is handled in order to fulfil the project goal of publishing semantically enriched data.

1.3 STRUCTURE OF THE DOCUMENT

We first describe the SALTED Federation setup and implementation in Section 2. This includes an overview of the deployed architecture, a thorough explanation of the data plane, the control plane and the data flows, and a detailed review of the security scheme that protects the Federation setup. In Sections 3 and 4, we describe the implementation and deployment of the DET Injection Chains and the DET Linkers, respectively. Lastly, Section 5 is a short chapter where we provide some general conclusions derived from the development process.

2 SALTED FEDERATION IMPLEMENTATION

2.1 ARCHITECTURE OVERVIEW

The SALTED architecture is based on a Scorpio Broker federated setup and DETs. DETs are the composition of microservices, or modules, that progressively process data in a pipeline-like approach. Every module is independent of the rest of the DET, as long as it is able to communicate with the other modules based on a common interface. This means that modules have virtually no restrictions in their implementation: choice of programming language, parametrisation options, functionality and reusability are all permitted but not enforced.

In general, DETs support data discovery, formatting, curation, linkage and enrichment as thoroughly explained in [1].

The output of a DET must be one or more NGSI-LD entities, which are injected into a Scorpio Broker. The architecture implementation highly encourages that a DET developed by a partner injects data into a local Scorpio Broker hosted by that partner. Since partners develop multiple DETs, the injection of data occurs concurrently. All the local Scorpio Brokers are connected with the so-called Federator Scorpio Broker (or simply Federator). The federated setup allows partners to work on their DETs locally while users make their requests to a single central broker. The Federator is able to forward the requests to the corresponding local brokers that have relevant data for that request. The Federation setup is shown in Figure 1.

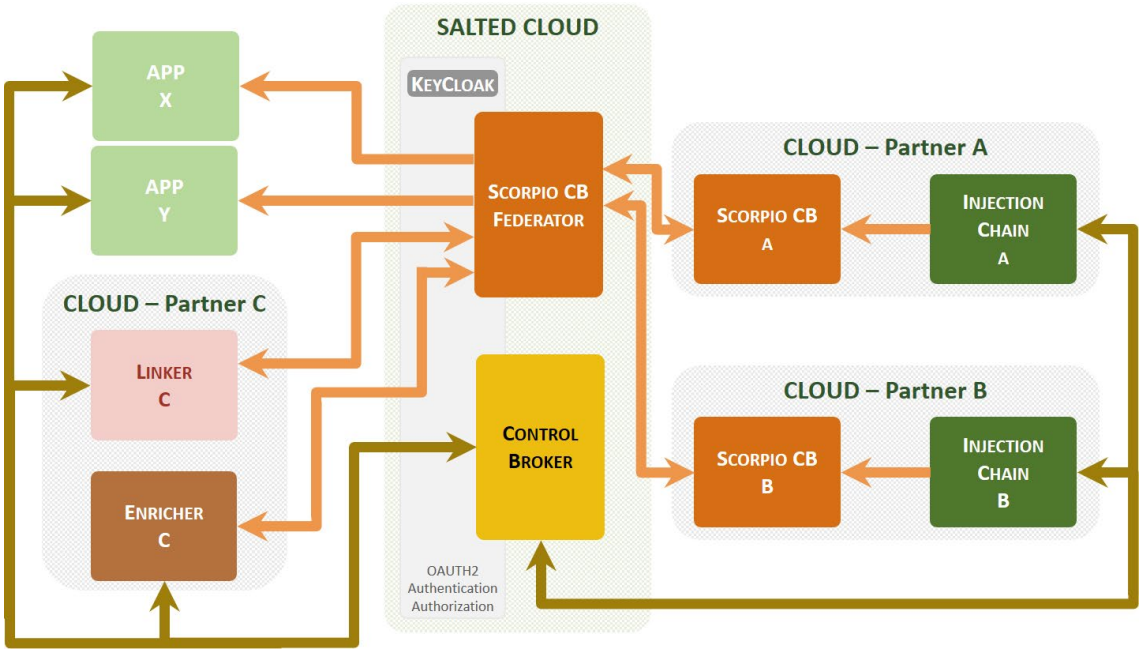


Figure 1. SALTED Federation setup

For the sake of clarity, we have separated the deployment of the DETs in “Clouds”. On the right hand, two sample partners are hosting an Injection Chain (part of the DET) and a local Scorpio Broker on their clouds, which is connected to the Federator through the federation mechanism. The Federator is located in the so-called SALTED cloud, which also includes the Control Broker (which will be expanded upon in the Control Plane section) and a Keycloak

instance (detailed in the Security section). The SALTED Cloud is currently deployed in the premises of the University of Cantabria.

On the left hand, another sample partner is hosting a Linker and an Enricher on their cloud, which are acting as separate modules within a DET. In this case, they are acting as users of the Federator since they do not have their own Scorpio Broker. The same can be said of the two sample Apps in the top left corner, which have read-only access to the Federator since they are external. We will give more details on roles and permissions in the Security section.

2.2 DATA PLANE

The Data Plane includes all functionalities and communication involving the SALTED data. In the Data Plane we collect, transform, curate, link, and enrich data. The architecture of the Data Plane is shown in Figure 2.

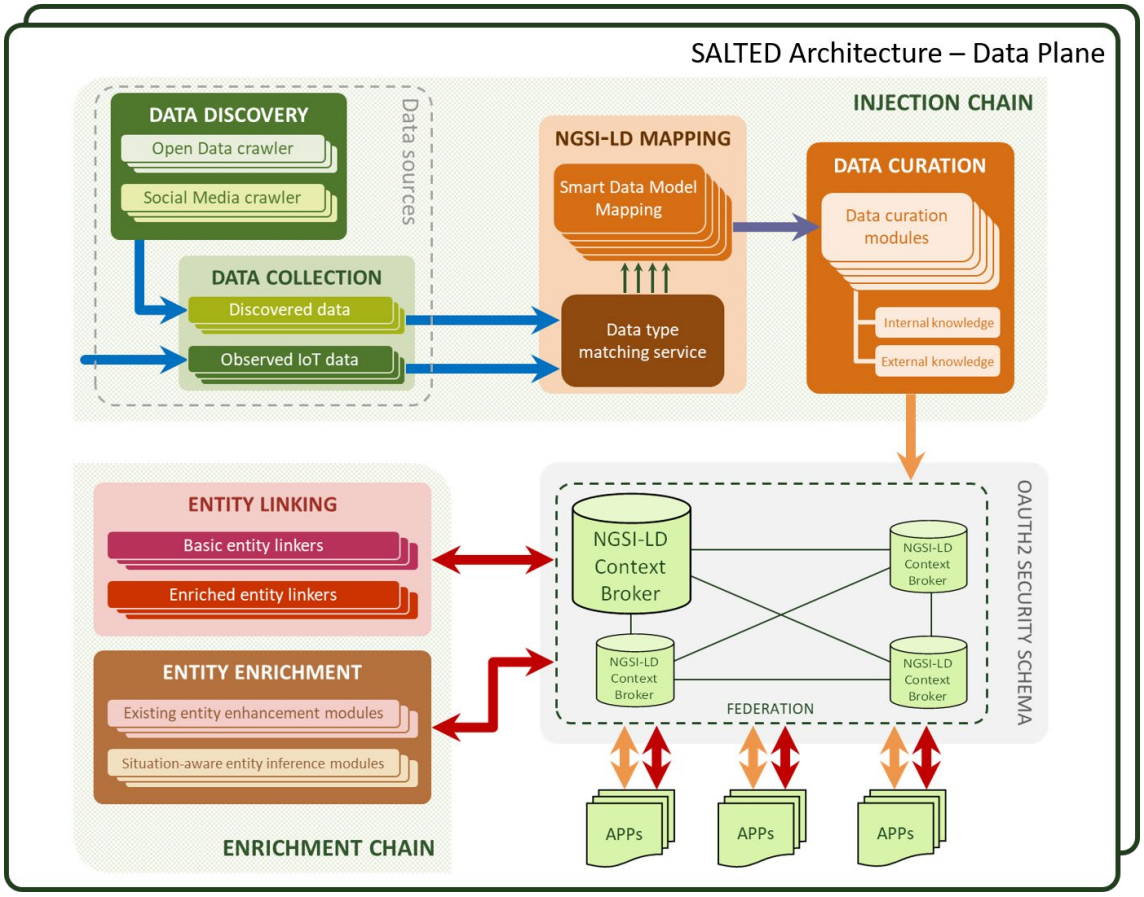


Figure 2. SALTED Data Plane architecture

A complete explanation of the DET components (i.e. data discovery, data mapping…) is available at [1]. In short, *data discovery and collection* crawl and gather data from heterogeneous data sources*, data mapping* transforms these data into the NGSI-LD information model (using Smart Data Models[1]), and *data curation* provides the means to assess the quality

[1] https://smartdatamodels.org/

of the data, filtering and tagging them before they are injected into the Scorpio Broker. This comprises the basic architecture of an Injection Chain.

On the other hand, the entity linking and entity enrichment components are in charge of, respectively, establishing NGSI-LD relationships between entities and either adding new high-value properties to existing entities or generating entirely new ones.

After the data mapping phase, the data flowing through the Data Plane must be NGSI-LD compliant in order to be able to interact with the Scorpio Broker. The communication between DET components is left to the choice of the developer; due to the great differences in the nature of the collected data, enforcing a single communication mechanism would be counterproductive. However, the communication flows within the Data Plane are well-defined and we will describe them in Section 2.2.1.

2.2.1 Communication Flows

Injection Flow

The first internal flow to consider is the Injection Flow, in which data located in a heterogeneous Data Source are collected (and transformed into NGSI-LD if necessary) by an Injection Chain and pushed into the Scorpio Broker. We consider as internal flow any flow where the data do not leave the internal SALTED architecture. A schematic Injection Flow is shown in Figure 3.

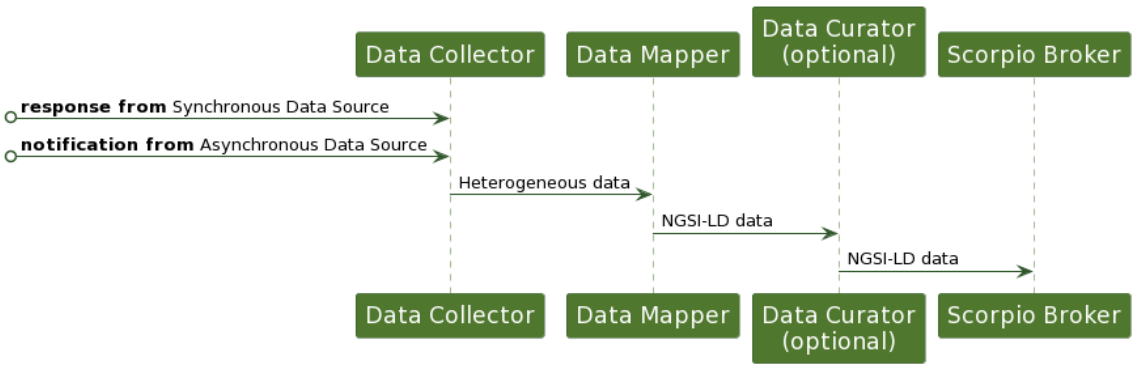


Figure 3. SALTED Injection Flow

The figure is focused on the flow of data, rather than the functionality of the DET components traversed by the data. Firstly, data is harvested from a Data Source, either synchronous (which would involve a request-response mechanism) or asynchronous (involving a subscription-notification mechanism). The Data Mapper receives the heterogeneous data and, after the transformation to NGSI-LD, forwards them to the Data Curator as an optional step. If there is no Data Curator, the data are injected directly into the Scorpio Broker. Otherwise, the Data Curator performs this injection.

Enrichment Flow

The Enrichment Flow is the other internal flow that is part of the Data Plane. It is triggered whenever an enrichment module gets data from the Scorpio Broker, carries out its functionality, and re-injects the enriched data back into the Scorpio Broker. This flow can be seen in Figure 4.

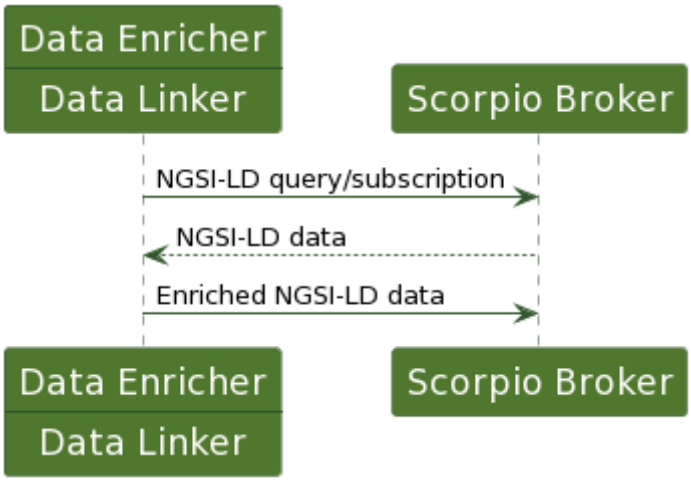


Figure 4. SALTED Enrichment Flow

The Enrichment Flow can be activated by both Enrichers and Linkers. These components get data from the Scorpio Broker by using any of the standard NGSI-LD operations (e.g. a query or a subscription) and perform a functionality that varies greatly from one component to another. In some cases, the NGSI-LD entities will be modified; in others, new entities may be generated. Regardless of the functionality, the newly enriched (or linked) NGSI-LD entities are re-injected into the Scorpio Broker to complete the data flow.

Query Flow

The first of the external flows, which involve external users or applications in some way, is the Query Flow. It is shown in Figure 5.

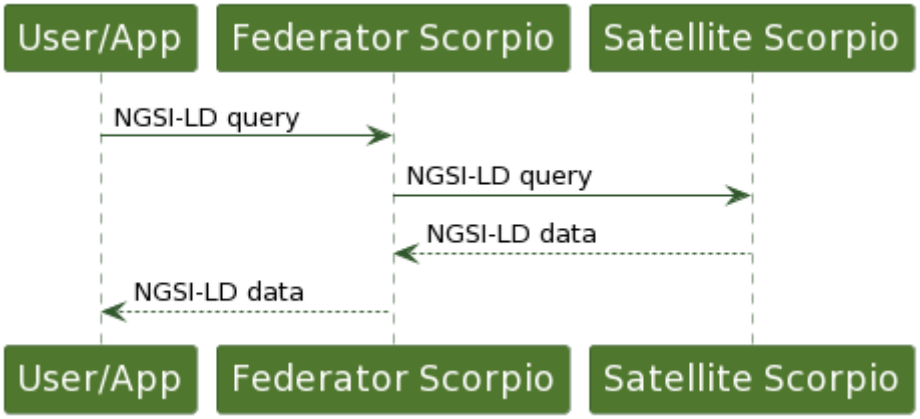


Figure 5. SALTED Query Flow

The Query Flow is triggered when a user or application requests data from the Scorpio Broker by means of a query. Since external users or applications only have access to the Federator Scorpio, and are unaware of the underlying architecture that involves Satellite Scorpios, this query is always made to the former. The Federator forwards the query to all of the Satellites that may have relevant data to respond. This is decided by leveraging the entity types and ID patterns requested against the ones stored in the context source registration. More detailed information can be found in the NGSI-LD specification [2]. Then, the Federator combines the responses into one and replies to the user's query with the requested data obtained from every Scorpio Broker in the federated setup.

This project is co-financed by the Connecting Europe Facility of the European Union under the Action Number 2020-EU-IA-0274.

Subscription Flow

Users or applications may also want to collect data asynchronously from SALTED. The Subscription Flow is meant to satisfy this need. It can be seen in Figure 6.

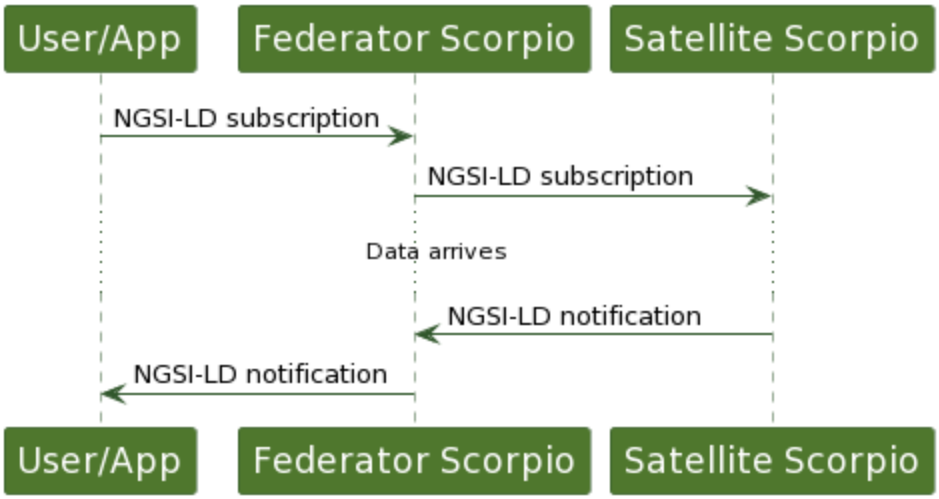


Figure 6. SALTED Subscription Flow

This flow is, to a certain extent, similar to the Query Flow. It is initiated by a user or application, but the mechanism used to obtain the data from the Scorpio Broker is a subscription instead of a query. This subscription is made to the Federator for the same reason as stated in the Query Flow. The Federator, in turn, makes a forward subscription to the Satellites that may have relevant data. Whenever a new NGSI-LD entity arrives in one of the Satellites, if it matches the subscription, a notification is generated from that Satellite to the Federator. Then, the Federator forwards it to the user or application.

2.3 CONTROL PLANE

Some of the DET components running on the Data Plane provide a configuration interface, allowing feedback or even requests coming from users and applications. The way this higher-level communication occurs is through the Control Plane, which works in parallel to the Data Plane. They are completely decoupled from each other, pushing the control functionalities into separate components. The Control Plane interfaces are enforced by SALTED, and they must follow some basic rules so that the communication between users/applications and DET components are well-defined and standardised.

The architecture of the Control Plane is represented in Figure 7. The core component is the Control Broker connecting all modules. We have implemented the Control Broker using EMQX[2], an Open Source MQTT broker focused on scalability for IoT. By using MQTT, we allow the configurable components to have an asynchronous connection to the broker that does not have to interfere with their Data Plane functionalities. The deployment of the EMQX Broker is located in the SALTED Cloud as shown in Figure 1, and it is publicly available[3].

[2] https://www.emqx.io/
[3] mqtts://control-broker.salted-project.eu

We will describe the communication flow between users and DET components, with the Control Broker in the middle, in Section 2.3.1.

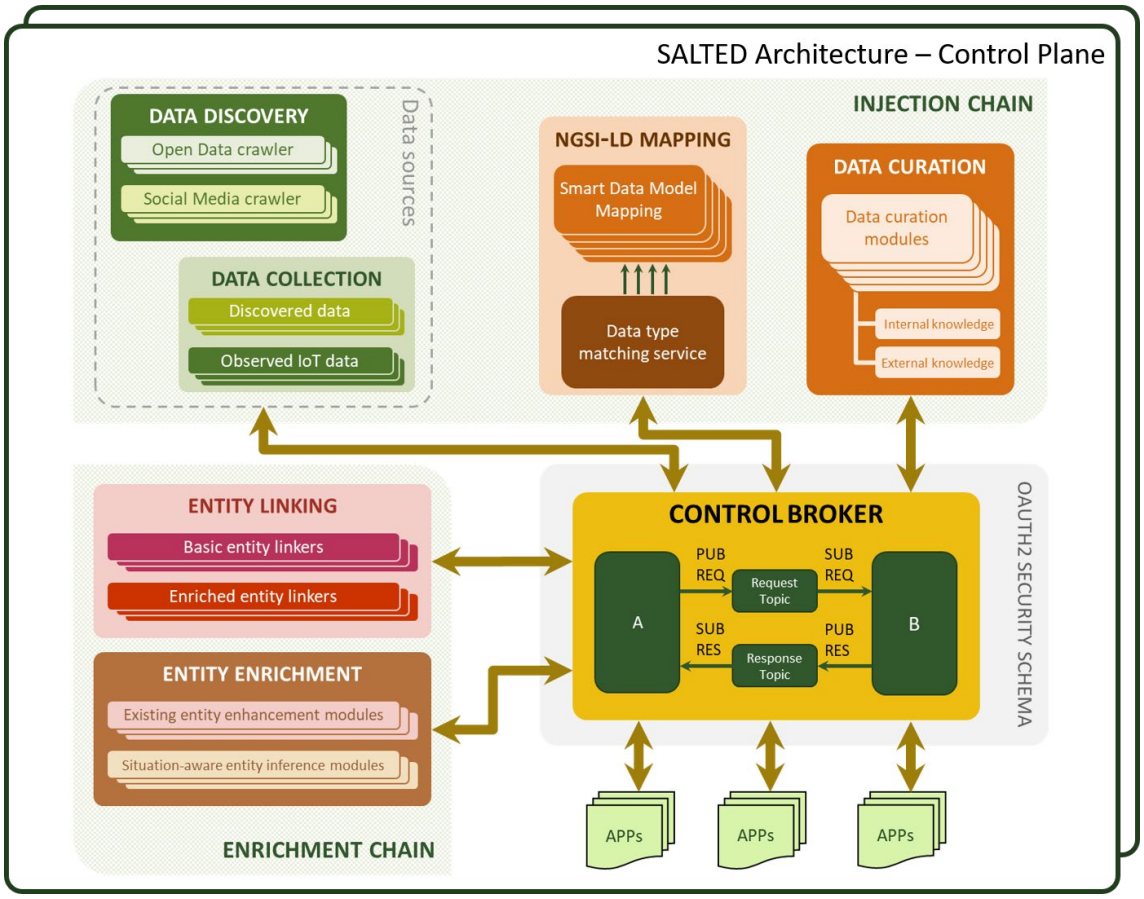


Figure 7. SALTED Control Plane architecture

2.3.1 Communication Flow

Parametrisation Flow

The Control Plane works with a single flow, in which applications can make a request to a parametrisable DET component through the Control Broker. The so-called Parametrisation Flow is shown in Figure 8.

This external flow is triggered by an application seeking a parameter change in one of the DET components. As a previous step, all parametrisable DET components are subscribed to the MQTT topics corresponding to their pre-configured ID, followed by a slash and any string (*det_id/#*). The application performing the request must first subscribe to the topic corresponding to their pre-configured ID (*app_id*) if they want to receive a confirmation. Then, the application posts a configuration update to the *det_id/app_id* topic, which the DET component is subscribed to. This allows the latter to obtain the *app_id* in addition to the configuration update itself. Once the reconfiguration is done, the DET component posts a confirmation message to the *app_id* topic, which the application is subscribed to. The EMQX broker is in charge of distributing the messages according to topics and subscriptions as defined by the MQTT protocol.

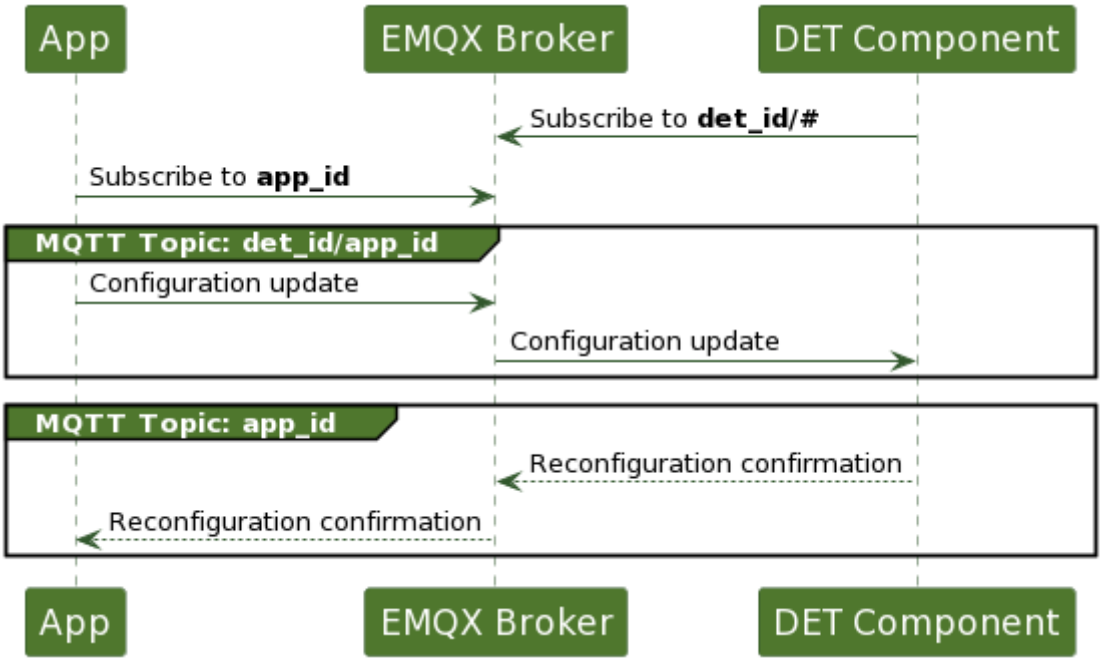


Figure 8. SALTED Parametrisation Flow

2.4 SECURITY

Access to the Federator Scorpio Broker and the EMQX Control Broker is restricted using OAuth 2.0[4]. We have used Keycloak[5] as our Identity and Access Management (IdM). The deployment, located in the SALTED Cloud as previously shown in Figure 1, is publicly available[6]. In our Keycloak setup, we have defined a set of roles, resources, scopes, policies and permissions that allow us to enable fine-grained authorisation to access the SALTED resources. We have also designed, implemented and deployed a proxy using *NodeJS* and *Express* that acts as the Policy Enforcement Point (PEP) to allow or deny access based on the policies previously defined in Keycloak.

2.4.1 Keycloak

The Keycloak setup starts with a realm called SALTED. We perform all the necessary configurations within this realm. The key element is the so-called *ScorpioBroker* client, that acts both as a Client and a Resource Owner, in OAuth 2.0 terms. It has all of the authorisation metrics defined in its configuration. We will detail these metrics before moving on to the rest of the setup.

Roles

Roles can be assigned to individual users, and they can be used later to identify whether those users have access to certain resources or not. We have defined three main roles: Admin, App, and Partner, although we can expand the number of roles in the future if we need to be more precise with the authorisation. The list of roles defined in Keycloak can be seen in Figure 9.

[4] https://oauth.net/2/
[5] https://www.keycloak.org/
[6] https://auth.salted-project.eu/

This project is co-financed by the Connecting Europe Facility of the European Union under the Action Number 2020-EU-IA-0274.

Role name	Composite
Admin	False
App	False
Partner	False

Figure 9. Roles defined in the SALTED realm

As the name implies, partners will be assigned the Partner role whereas external applications using the SALTED infrastructure will have the App role. The Admin role is meant to have access to everything with no limitations. We will define the permissions given to every role in a later paragraph.

Resources and scopes

The resources defined in the *ScorpioBroker* client mimic the routes available in the Scorpio Broker itself. This lets us allow or deny specific Roles access to specific functionalities (e.g. external users may read an entity, but they will not be able to delete it). In a similar manner, the scopes defined mirror the HTTP methods used to interact with the REST interface of the broker. The resources we have defined are shown in Figure 10.

Name	Type	Owner	URIs
csourceRegistrations (Resource)	ScorpioBroker:restricted	ScorpioBroker	/ngsi-ld/v1/csourceRegistrations*
Entities (Resource)	ScorpioBroker:restricted	ScorpioBroker	/ngsi-ld/v1/entities*
EntityOps delete (Resource)	ScorpioBroker:restricted	ScorpioBroker	/ngsi-ld/v1/entityOperations/delete
EntityOps post (Resource)	ScorpioBroker:restricted	ScorpioBroker	/ngsi-ld/v1/entityOperations/update 2 more
Subscriptions (Resource)	ScorpioBroker:restricted	ScorpioBroker	/ngsi-ld/v1/subscriptions*
Temporal (Resource)	ScorpioBroker:restricted	ScorpioBroker	/ngsi-ld/v1/temporal/entities*
Types (Resource)	ScorpioBroker:open	ScorpioBroker	/ngsi-ld/v1/types

Figure 10. Resources defined in the ScorpioBroker client

The *URIs* column shows the actual route of the Federator, not including the base URL. The scopes can be seen in Figure 11.

Name	Resources	Permissions
DELETE	Temporal (Resource) 3 more	entities:DELETE 3 more
GET	Temporal (Resource) 4 more	types:GET 4 more
PATCH	Temporal (Resource) 3 more	temporal:POST 3 more
POST	Temporal (Resource) 5 more	temporal:POST 5 more
PUT	Temporal (Resource) 3 more	temporal:POST 3 more

Figure 11. Scopes defined in the ScorpioBroker client

The combination of resources and scopes will allow us to make fine-grained authorisation decisions based on the role of the user that is making the request, and the route and HTTP method that are being requested.

Policies and permissions

Policies define the conditions that needs to be met in order to pass a check. We have used the possession of the aforementioned roles as our conditions. These policies are shown in Figure 12.

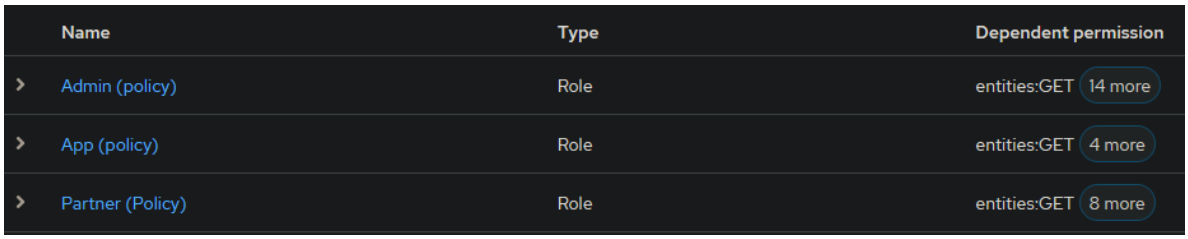


Figure 12. Policies defined in the ScorpioBroker client

For instance, the App policy will check if the user has the App role assigned. Lastly, permissions are a combination of resources and scopes that are associated with a policy. In a way, permissions are where all the previous configurations are linked. Figure 13 shows a list of all the permissions defined in the *ScorpioBroker* client.

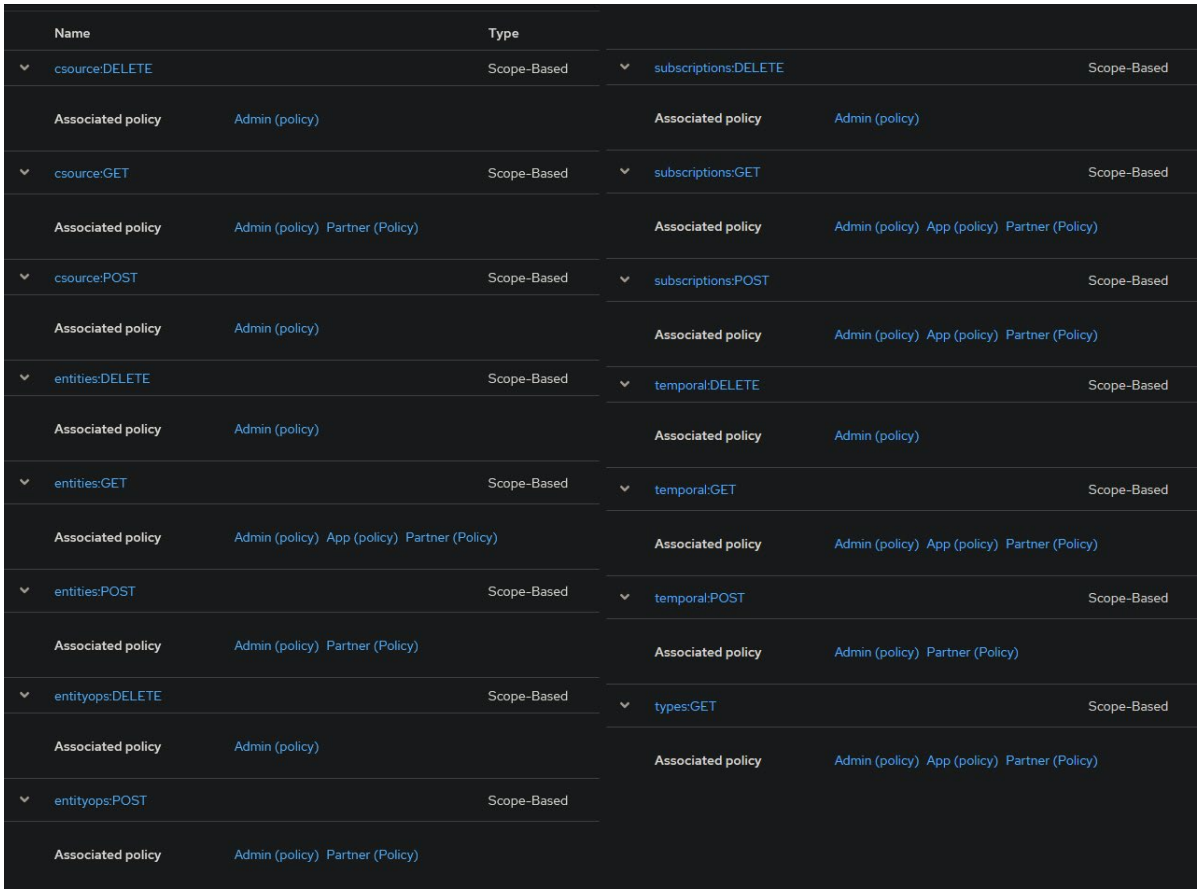


Figure 13. Permissions defined in the ScorpioBroker client

By looking at the associated policies, we can tell that all of the roles (Admin, Partner and App) can make a GET request to the */entities* endpoint, but only an Admin or a Partner can make a

POST request in the same route. Meanwhile, only an Admin can make a DELETE request. We have defined a set of similar permissions for every resource, scope and policy.

As a side note, the PATCH and PUT HTTP methods have been merged into the POST-based permissions, and that is the reason why they don't appear directly as permissions.

2.4.2 PEP proxy

The policies and permissions defined in Keycloak need to be enforced externally, since Keycloak does not act as an enforcer by itself. We have deployed a proxy acting as the PEP, which intercepts all the traffic, checks the claims and allows access or denies it based on the aforementioned configuration.

In order to do this, all incoming requests need to include a JSON Web Token (JWT) granted by Keycloak. Users authenticate to Keycloak with their own client ID and client secret, and they are granted a JWT that includes their role. This role is used by the PEP proxy to determine whether the users have access to the requested resource and scope (URL and HTTP method, respectively).

The PEP proxy is implemented using *NodeJS* and *Express*, and deployed in the same Ubuntu Virtual Machine as the Federator Broker. It listens to HTTPS requests only. All available routes are protected with the *keycloak.enforcer* function available in the *keycloak-connect* package. This function checks if the claims contained in the JWT match those required to access the protected resource. If the claim is valid, the request is forwarded to the Broker and the response is forwarded to the user seamlessly. Asking for a protected resource without the necessary claims will trigger an "Access Denied" message. Asking for an invalid endpoint will trigger a "Requested endpoint not found or not available" message.

2.4.3 EMQX

The EMQX Control Broker allows for authentication by using native plugins. For the sake of consistency, we have implemented security by using the *emqx_auth_jwt* plugin, that uses JWT for authentication. One of the options offered by the plugin is to define a JSON Web Key Sets (JWKS) server, which is a server containing the public keys used to verify a JWT. By configuring the */certs* endpoint of the SALTED realm of our Keycloak instance[7] as the JWKS server, we make sure that the JWTs used for authentication must have been issued by the SALTED Authentication Server.

Users looking to connect to the EMQX Control Broker must include a JWT as their Username. A sample connection can be seen in Figure 14. If the JWT is not included, not valid, or not issued by SALTED, the connection is refused.

[7] https://auth.salted-project.eu/realms/SALTED/protocol/openid-connect/certs

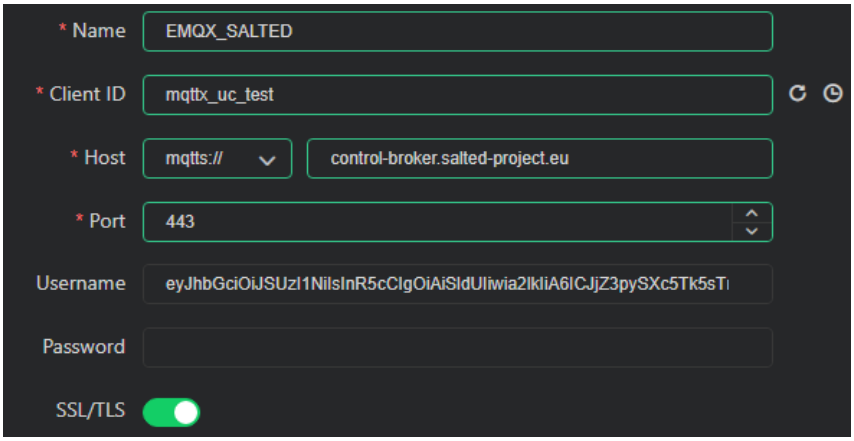


Figure 14. EMQX sample connection

2.4.4 Secured Communication Flows

The aim of this section is to provide an update of the communication flows explained above. The update adds a security layer to the communication flows, underpinned by the mechanisms seen in the Security section.

Injection Flow

The updated Injection Flow is shown in Figure 15.

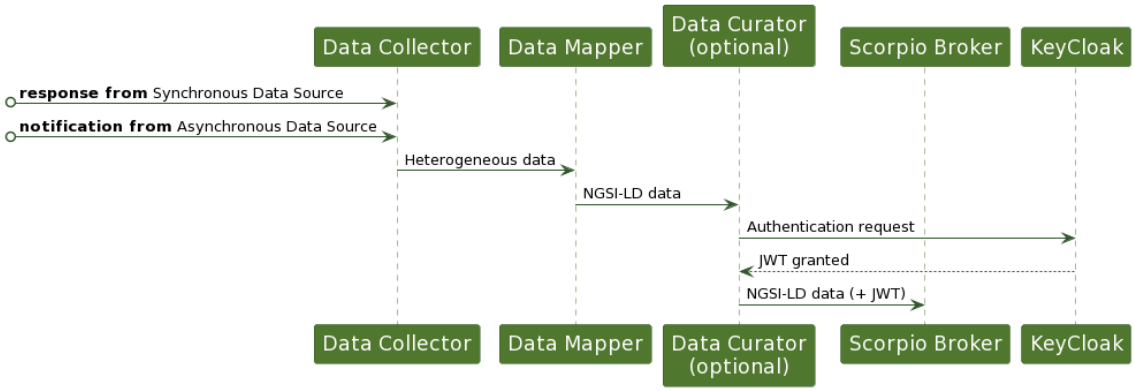


Figure 15. SALTED Injection Flow with security

Since the security check is located in front of the Scorpio Broker, the DET component that injects the data (whether the Data Mapper or the Data Curator, the latter in the figure) is in charge of doing so in possession of a valid JWT. The DET component obtains the JWT beforehand by making a token request to the SALTED Keycloak instance. The DET components have the necessary client credentials in their configuration, so they can get fresh tokens anytime. The tokens can be included in any HTTP request by means of the Authorization header.

Enrichment Flow

The updated Enrichment Flow is shown in Figure 16.

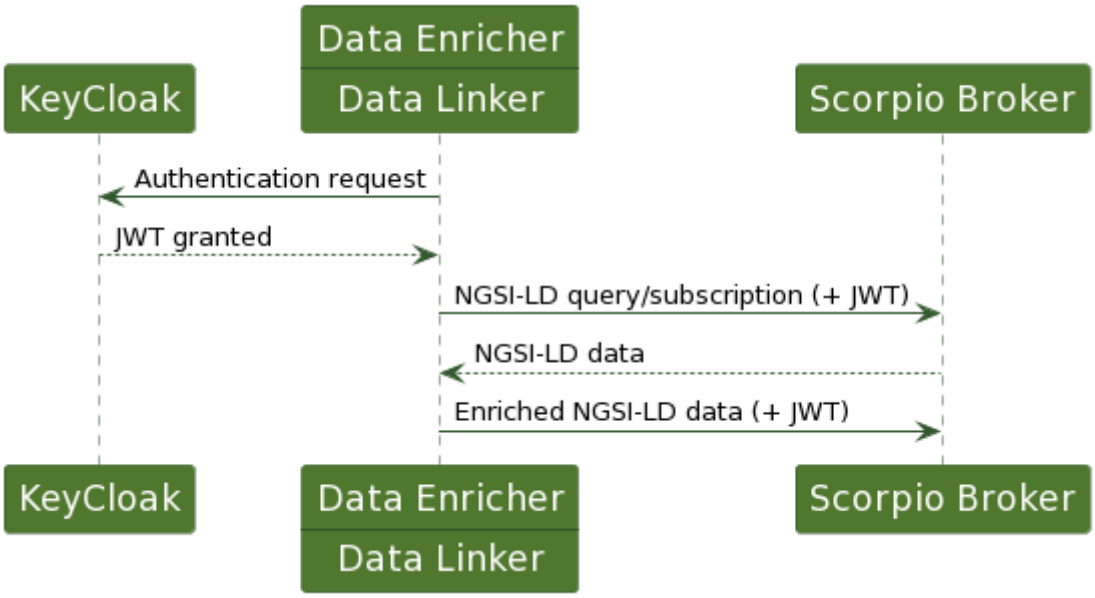


Figure 16. SALTED Enrichment Flow with security

The update is similar to the previous case. The DET component obtains a JWT before making the request (or subscription) to the Scorpio Broker. After processing the NGSI-LD entities, the DET component has to re-inject them into the Scorpio Broker, which also requires authorization. If the token has expired since it was last used, then the DET component asks the SALTED KeyCloak instance for a new one. In SALTED, tokens have an expiry time of 15 minutes.

Query Flow

The updated Query Flow is shown in Figure 17.

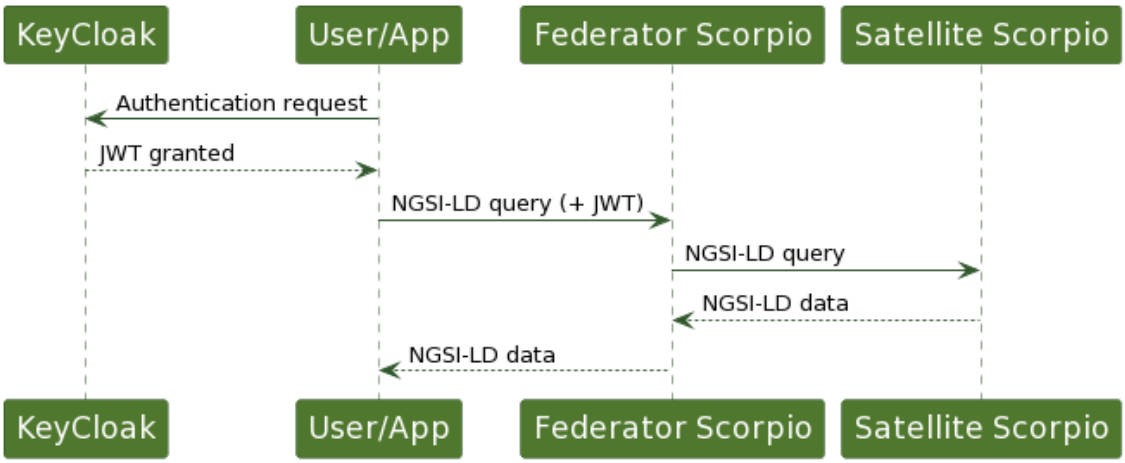


Figure 17. SALTED Query Flow with security

The users, or external applications, must be registered in the KeyCloak instance as clients before they can access the SALTED data. Once they have valid credentials, they can initiate the Query Flow by obtaining a JWT and performing a query operation to the Federator with the JWT included in the Authorization header.

Subscription Flow

The updated Subscription Flow is shown in Figure 18.

This project is co-financed by the Connecting Europe Facility of the European Union under the Action Number 2020-EU-IA-0274.

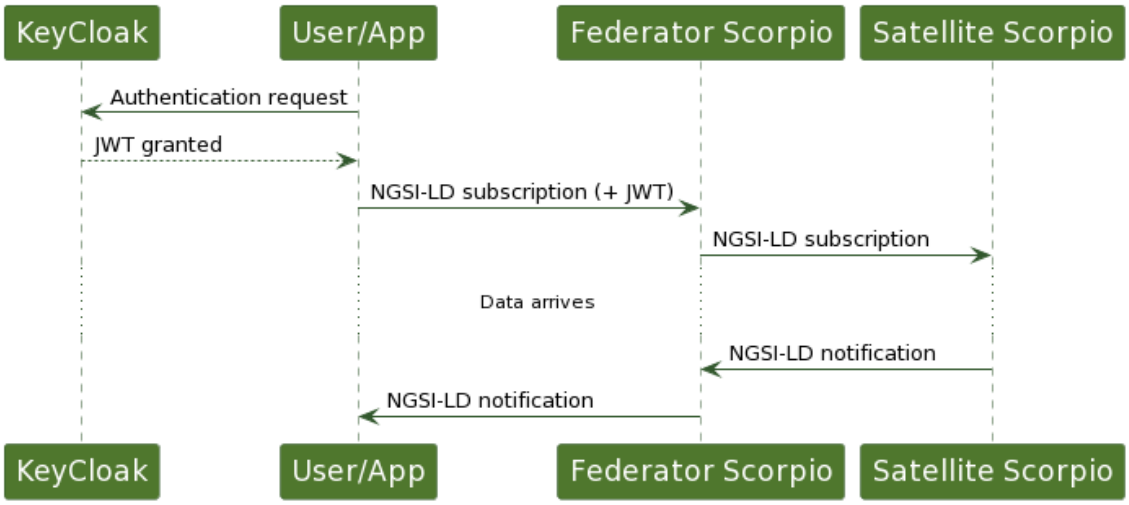


Figure 18. SALTED Subscription Flow with security

The security layer added in the Subscription Flow is analogous to the Query Flow case. The user or application, registered beforehand as a client in the KeyCloak instance, authenticates and obtains a JWT. This token is included in the Authorization header when making the subscription request to the Federator. Since notifications flow from the Satellite Scorpio to the user, no further security steps are required in the asynchronous phase of the Subscription Flow.

Parametrisation Flow

The updated Parametrisation Flow is shown in Figure 19.

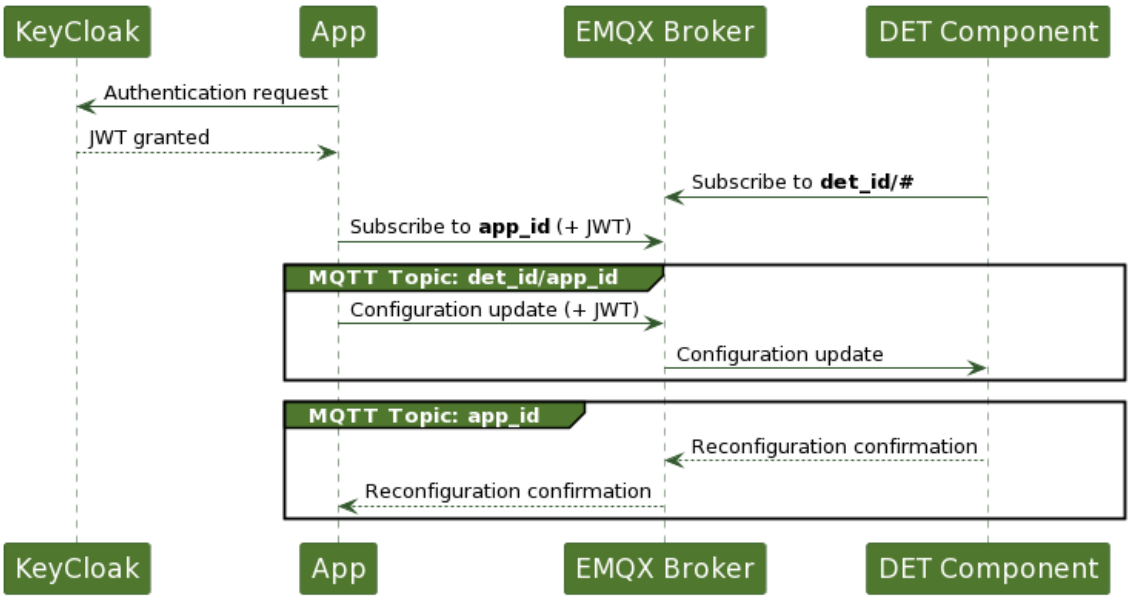


Figure 19. SALTED Parametrisation Flow with security

Even though the update to the Parametrisation Flow is used in the Control Plane, its structure hardly differs from the previous ones, used in the Data Plane. The application requesting a parametrisation update has to obtain a JWT from the KeyCloak instance in order to access the EMQX Broker. However, in this scenario, the token is not included in an Authorization header (this concept only applies to HTTP communication). Instead, the token must be used to fill the Username field in the MQTT connection to the EMQX Broker, as previously shown in Figure 14.

3 DET INJECTION CHAINS

3.1 OVERVIEW

Injection Chains, as the starting element of DETs of the SALTED project, are data pipeline processes that consist of the discovery and collection of heterogeneous data, NGSI-LD mapping, data curation and finally injection of this data into a Linked-Data Context Broker.

As defined in detail in previous deliverables [1], the following steps need to be considered for any SALTED Injection Chain:

1. Identifying the source of the data: Determine where the data is coming from and what format it is in.
2. Clean the data: Ensure that the data is in a suitable format for the system and remove any irrelevant or duplicated information.
3. Transform the data: Convert the data into the format required by the target system.
4. Load the data: Insert the transformed data into the target system.
5. Verify the data: Ensure that the data has been successfully loaded and that it is accurate and complete.
6. Monitor the data: Continuously monitor the data to ensure that it remains accurate and up-to-date.

The first step in the process is to gather relevant data from various sources, including IoT data sources, social media data, web data etc. using the appropriate tools. Once we have collected this data, it must be standardised and organised into a unified format that allows for the clear definition of semantic concepts and the linking of existing information. This is achieved by identifying the various data types, such as entities, properties, and relationships, from the raw data and mapping it to the NGSI-LD Information Model. This mapping process is followed by a thorough evaluation of the quality of the data, including the addition of valuable metadata, if available. This ensures that the enriched resulting data is meaningful and of high quality before made accessible to external applications.

3.2 NGSI-LD DATA MODELS

Due to the continuous integration of new data sources into the SALTED project, data models are also continuously evaluated, since they are used to map data into the NGSI-LD information model. Through the Smart Data Models project[8], there is a large pool of already created models, which is also continuously growing. This benefits the SALTED project, since the adaptation with already existing data models points to further brokers, which represent data in the same way, enables future use cases across brokers. Furthermore, the Smart Data Models may be integrated directly into the development in the future through a newly available Python package[9].

[8] https://smartdatamodels.org/
[9] https://smartdatamodels.org/index.php/draft-of-a-python-package-available/

The following part will give a short overview about additional NGSI-LD data models leveraged by SALTED since the D2.1 and D1 reports. Additionally, considerations on questions such as "How to model an entity? What is important?" that were identified through the continuing work with NGSI-LD and building up use cases will be listed.

How to – Important questions

- What makes an entity easily understandable for users and other interested parties?
    - How much explanation do abstract/conceptual entities need, if it's not as "self-explainable" as a physical entity like BikeHireDockingStation?
- How should information sources be modelled?
    - There is a need for transparency/explainability and accountability.
- Should there be project guidelines on using the @context?
    - The broker would also accept entities without @context specified for all properties, which could lead to no understanding or unused entities in the future.

EVChargingStation

This data model can be found within the Smart Data Models[10].

Within SALTED, this model is used when integrating further data from the MRN Data Broker, which holds valuable information regarding the Metropol-Rhine-Neckar region[11]. The mapping handles the differences between the data model used within the MRN Broker and the NGSI-LD format. Essentially the adding of the context information is needed in this specific case. Right now, the property "name" is used to identify if an entity representing this real-world object is already present in the Broker. Figure 20 shows an example entity.

[10] https://github.com/smart-data-models/dataModel.Transportation/tree/master/EVChargingStation
[11] https://digitale-mrn.de/projekte/kooperative-dateninfrastrukturen/datenportal

```
{
    "id": "urn:ngsi-ld:EVChargingStation:0cd0c31e-43d8-4abf-9212-a3c3f4c2b3bf",
    "type": "EVChargingStation",
    "address": {
        "type": "Property",
        "value": {
            "addressCountry": "DE",
            "addressLocality": "Speyer",
            "postalCode": "67346",
            "streetAddress": "Birkenweg"
        }
    },
    "amperage": {
        "type": "Property",
        "value": [
            32,
            32
        ]
    },
    "availableCapacity": {
        "type": "Property",
        "value": 0,
        "observedAt": "2022-12-13T10:20:03.000Z"
    },
    "capacity": {
        "type": "Property",
        "value": 2
    },
    "contactPoint": {
        "type": "Property",
        "value": {
            "openingHours": []
        }
    },
    "socketType": {
        "type": "Property",
        "value": [
            "TYPE_2_OUTLET",
            "TYPE_2_OUTLET"
        ]
    },
    "chargingUnitId": {
        "type": "Property",
        "value": [
            "BE-8200-0",
            "BE-8204-6"
        ]
    },
    "location": {
        "type": "Property",
        "value": {
            "type": "Point",
            "coordinates": [
                8.429105,
                49.351394
            ]
        }
    },
    "name": {
        "type": "Property",
        "value": "P000275"
    },
    "operator": {
        "type": "Property",
        "value": "SWS Stadtwerke Speyer GmbH"
    },
    "@context": [
        "https://raw.githubusercontent.com/smart-data-models/dataModel.Transportation/master/context.jsonld",
        "https://smartdatamodels.org/context.jsonld",
        "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context-v1.3.jsonld"
    ]
}
```


Figure 20. Example of an EVChargingStation entity

KeyPerformanceIndicator

This data model can be found within the Smart Data Models[12].

Within SALTED, this model is used to represent the results of the compliance score calculations within the Agenda Analytics use case of SALTED. Entities of this type can be visualized for the end user by a dedicated service (e. g. in a coloured map). Within this state of the project the concrete usage of the properties given by the NGSI-LD data model are still being discussed. The storage concept of the calculation base, the transparency for the end user and the persistence are concepts that have big influence on these decisions on how to model an entity. A first attempt is shown in the following graphic included in Figure 21:

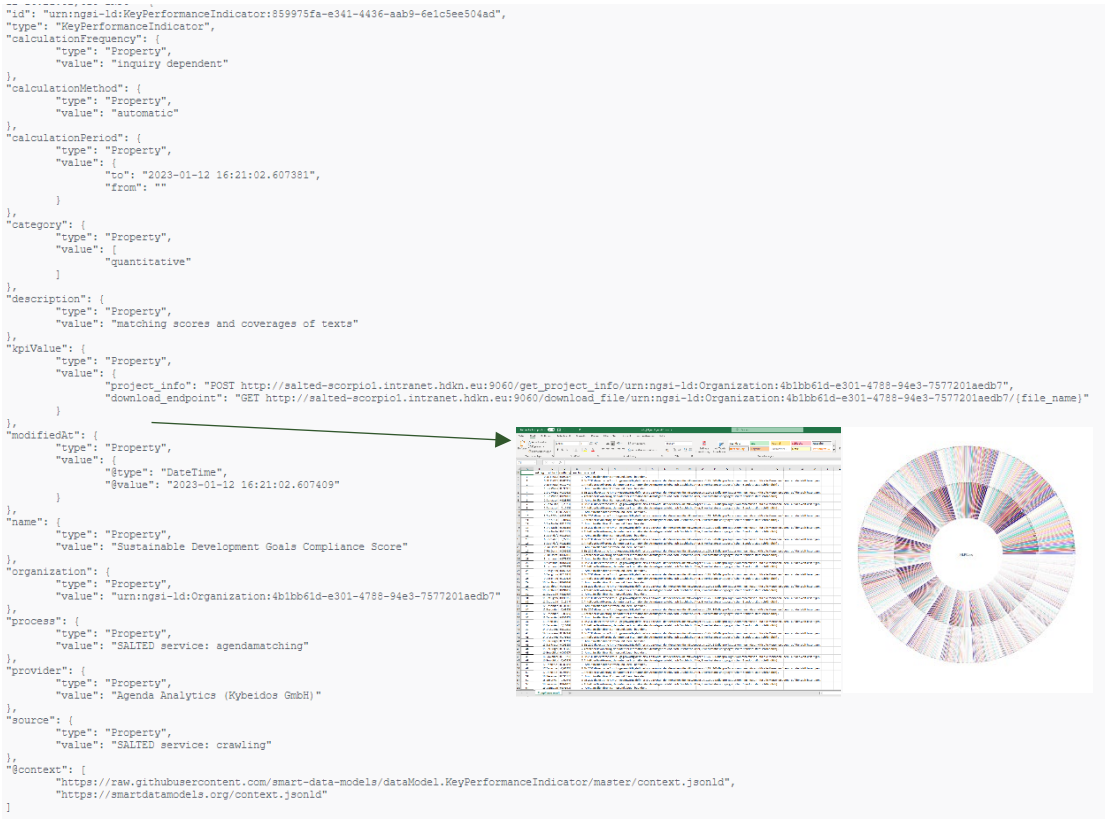


Figure 21. Example of use of the KeyPerformanceIndicator entity

SDMX - Statistical Data and Metadata eXchange

This data model can be found within the Smart Data Models[13].

This data model extends the data model KeyPerformanceIndicator with multidimensional statistical information and is based on the work of the community of the same name 'SDMX -

[12] https://github.com/smart-data-models/dataModel.KeyPerformanceIndicator
[13] https://github.com/smart-data-models/dataModel.SDMX

Statistical Data and Metadata eXchange'[14]. Eurostat, the statistical office of the European Union, also relies on SDMX as a future standard for the publication of open data[15].

The integration of SDMX as a standard for modelling data from official statistics in NGSI-LD is, in our view, actually the first point of contact between the worlds of open statistical data and open data from the urban and industrial context.

At first glance, the emerging interoperability of these standards may seem to be mere technicality. However, from our point of view, it is interesting to see which new possibilities of data access and application result from this for statistical data:

- The focus of the NGSI-LD standard on the description of context (place, time, surrounding) and on the semantic connection of data models means:
    - Statistics data could be accessed in the future based on context, i.e. with reference to time, place and occasion.
    - Statistical data could be linked with adjacent data models - e.g. accident statistics with traffic volume.
- Conversely, the knowledge and tools from official statistics could be applied to the context of smart cities, smart industry and IoT.

  The statistically qualified processing of data from NGSI-LD platforms may enable a new level of quality to be achieved in the further development of data-based steering processes.

Within the SALTED project this possibility will be evaluated more deeply. The following explanations set the scope defined and intended by now: Socio-economic data provided by the German Federal Statistical Office will be used as an example within a proof of concept. The idea is to use a social media platform, such as a chatroom, to connect an end user to the statistical information available around him. A bot implementation could use the geo tag of the location sharing functionality to answer a user's questions regarding data recorded in the specific area.

Data is available on annual basis on the statistics portal GENESIS of the Federal Statistical Office - at the level of territorial authorities of the defined regional levels: the cities and municipalities, the association municipalities, the districts, the regional districts, and the federal states.

Through enrichment processes on the SALTED platform, the data will be furthermore enhanced in the following way, to enable more complex applications:

- Regional unit keys are linked to geographical information so that information can be mapped and linked to other geolocated data.

The enrichment process will also show how time series aspects and the linking with hierarchically higher or lower territorial units as well as the calculation of derived key figures can be done.

[14] https://www.sdmx.org
[15] https://ec.europa.eu/eurostat/web/sdmx-infospace

DataQualityAssessment

This data model can be found within the Smart Data Models[16].

Within the SALTED project, we have contributed to the Smart Data Models initiative by releasing a new data model for Data Quality Assessment. Our main goal is to characterise a set of quality properties of several entities or measurements, such as temperature, and provide high-quality information to our consumers. These properties are obtained as a result of an assessment module (the Curator module in the SALTED architecture) and embedded into the actual data stream through the power of the NGSI-LD relationships.

This data model includes the description of four Data Quality (DQ) Dimensions and the outcome of two DQ enhancement techniques. The four DQ Dimensions are:

- **accuracy:** Accuracy measures the maximum systematic numerical error produced in a sensor measurement. It may take values between 0 and 1.
- **precision:** Precision measures the standard deviation of a dataset. That is, it measures how close the values in the dataset are to each other. It may take values between 0 and 1.
- **timeliness:** Timeliness measures the update time of sensor measurements. Default unit: minutes.
- **completeness:** Completeness quantifies the number of missed measurements or observations in a given time window. It may take values between 0 and 1.

On the other hand, the two DQ enhancement techniques are outlier/novelty detection and interpolation. The first one is represented in the data model as:

- **outlier:** Includes information about the outlier characteristics of the measurement.
    - **isOutlier:** Determine whether the measurement has been considered an outlier or not. It may take a Boolean value.
    - **methodology:** Reference (relationship) to another entity including AI methodology information.

Whereas the second enhancement technique is described as:

- **synthetic:** Includes information about the origin of the measurement.
    - **isSynthetic:** Determine whether the measurement has been created synthetically or not. It may take a Boolean value.
    - **methodology:** Reference (relationship) to another entity including AI methodology information.

An example of use of the DataQualityAssessment entity is shown below, in Figure 22.

[16] https://github.com/smart-data-models/dataModel.DataQuality/tree/master/DataQualityAssessment

```
{
    "id": "urn:ngsi-ld:DataQualityAssessment:Temperature:smartsantander:u7jcfa:f3058",
    "type": "DataQualityAssessment",
    "dateCalculated": {
        "type": "Property",
        "value": {
            "@type": "DateTime",
            "@value": "2022-09-10T10:01:20Z"
        }
    },
    "source": {
        "type": "Property",
        "value": "https://salted-project.eu"
    },
    "outlier": {
        "type": "Property",
        "value": {
            "isOutlier": {
                "type": "Property",
                "value": true
            },
            "methodology": {
                "type": "Relationship",
                "object": "urn:ngsi-ld:AI-Methodology:Outlier:Temperature:smartsantander:u7jcfa:f3058"
            }
        },
        "observedAt": "2022-09-10T10:01:20Z"
    },
    "synthetic": {
        "type": "Property",
        "value": {
            "isSynthetic": {
                "type": "Property",
                "value": false
            },
            "methodology": {
                "type": "Relationship",
                "object": "urn:ngsi-ld:AI-Methodology:Synthetic:Temperature:smartsantander:u7jcfa:f3058"
            }
        },
        "observedAt": "2022-09-10T10:01:20Z"
    },
    "accuracy": {
        "type": "Property",
        "value": 0.25,
        "observedAt": "2022-09-10T10:01:20Z",
        "unitCode": "CEL"
    },
    "timeliness": {
        "type": "Property",
        "value": 3,
        "observedAt": "2022-09-10T10:01:20Z",
        "unitCode": "minutes"
    },
    "precision": {
        "type": "Property",
        "value": 1.3,
        "observedAt": "2022-09-10T10:01:20Z",
        "unitCode": "CEL"
    },
    "completeness": {
        "type": "Property",
        "value": 0.5,
        "observedAt": "2022-09-10T10:01:20Z",
        "unitCode": "P1"
    },
    "@context": [
        "https://raw.githubusercontent.com/smart-data-models/dataModel.DataQuality/master/context.jsonld",
        "https://smartdatamodels.org/context.jsonld"
    ]
}
```


Figure 22. Example of a DataQualityAssessment entity

3.3 IoT Data Injection Chain

The IoT Data Injection Chain (IoTDIC) is composed of three main phases: Data Collection, Data Mapping, and Data Curation. The data processed by the IoTDIC include data obtained from Smart Cities (e.g. SmartSantander), Open Data portals (e.g. Barcelona) and initiatives like The Things Network (TTN). These data sources are heterogeneous in nature and require different treatment for their collection and processing. For instance, SmartSantander delivers data in an asynchronous way through a subscription interface, whereas Open Data portals based on Comprehensive Knowledge Archive Network (CKAN) offer synchronous endpoints based on a request/response philosophy. These aspects demand that the IoTDIC is prepared to deal with several heterogeneous data flows generating data concurrently.

Data Collection

The Data Collection phase is responsible for accessing the data hosted in the aforementioned heterogeneous external sources and introducing them into the DET. We have implemented several parallel modules, named Data Collectors, one for each of the data sources. This brings the current number of Data Collectors within this Injection Chain to 11. Details on the data sources associated can be found in Deliverable 2.1 [1].

Synchronous Data Collectors are implemented in Python, and they use the *json* library for formatting, the *requests* library for communication, and several others (e.g. *configparser* or *logging*) for miscellaneous tasks. In terms of functionality, they send an HTTP request to the corresponding endpoint and receive the data in the body of the HTTP response. These data are either in JSON format, CSV format or a proprietary format in the specific case of traffic data from Barcelona. The Data Collector normalises the format to JSON and the dates to UTC, and forwards the data to the next phase of the IoTDIC, the Data Mapping phase, via an HTTP POST request to a REST server. Data are processed as a batch whenever possible.

The two asynchronous Data Collectors implemented actually work with different protocols. The Data Collector that gathers data from the SmartSantander IoT sensors implements a REST server, due to the fact that SmartSantander offers a REST subscription API. We have implemented this with Python's *flask* library, along with *waitress* for achieving good performance with several threads working at the same time. On the other hand, the TTN enables a MQTT broker to access the data provided by the LoRaWAN devices, which requires the corresponding Data Collector to implement an MQTT client. We used the *paho-mqtt* library for the implementation of this client. In both cases, after the individual datum is received through a notification (whether via HTTP or MQTT), the Data Collector performs the same kind of normalisation as in the synchronous case and forwards it to the Data Mapping phase. In this case, however, data are treated as a stream.

In all cases, an Ubuntu 20.04.5 LTS Virtual Machine is hosting the Data Collectors and the *cron* tool is scheduling the synchronous ones so that they request the data periodically with the frequency they are updated (e.g. Bilbao Open Data portal updates its traffic data every five minutes, so we perform the request periodically every five minutes). The *cron* tool also performs a health check on the asynchronous Data Collectors to make sure they are up and running.

Data Mapping

In the Data Mapping phase, both datasets produced by the synchronous Data Collectors and data-streams coming from the asynchronous Data Collector are received at the REST server's

endpoint. This server is implemented with Python libraries *flask* and *waitress*. The Data Mapper, which is a unique instance handling several flows at a time, is able to discern whether the request contains an individual entity or a batch by using two different routes. Regardless of the nature of the data, the mapping process is handled entity by entity.

The mapping process begins by identifying the entity type. This refers to the *type* attribute of the resulting NGSI-LD entity. In some cases, the Data Collector informs the Data Mapper of this entity type beforehand so this step is straightforward. However, in other cases the type is not known in advance, so the Data Mapper performs an AI-aided type identification. This is done with Python's *tensorflow* library, and more specifically the *keras* module. The AI model used for type identification has been previously trained with JSON entities coming from some of the data sources we have been working with. This training dataset contains entities belonging to all the entity types we use, since omitting a type will stop the AI from recognising it. If the AI model detects an unrecognised data type, it saves the entity in a file in order to manually retrain the model later with the new data.

Once the entity type is known, the Data Mapper begins the process of transforming the JSON object to a proper NGSI-LD entity. All NGSI-LD entities generated in SALTED are compliant with the Smart Data Models initiative. The transformation uses a Nearest Neighbour approach to individually map every field of the input JSON object into the most fitting NSGI-LD property in the corresponding Smart Data Model. First, the JSON object goes through a flattening function and is vectorized using the *sklearn* library that can be found in Python. The vectors obtained, one for every field, are compared with the training vectors that contain a growing list of words related to every NGSI-LD property present in the Smart Data Model. This way, every field in the JSON object finds its nearest neighbours and is mapped into it. If a field does not find a proper neighbour, it is copied into an external file for a possible retraining and then it is discarded. After finding the nearest neighbours, a *JMESPath* template is used to obtain the final NGSI-LD entity. These templates allow us to fine tune some of the more complex parts of the entity, such as precise timestamps or sub-properties (properties of properties).

After we obtain the mapped NGSI-LD entity, compliant with the corresponding Smart Data Model, the Data Mapper can inject it into the Scorpio Broker through an HTTP POST request. However, the IoTDIC includes one more phase. The single entity, or the batch of entities, are forwarded to the Data Curation phase via MQTT. The client used in the Data Mapper is based on the *paho-mqtt* library for Python.

The Data Mapper is deployed in an Ubuntu 20.04.5 LTS Virtual Machine, and the *cron* tool is constantly performing a health check on it to make sure it is working properly.

Data Curation

The Data Curation module is, by nature, a particular case of a Data Enrichment module. As such, the definition of its implementation will be thoroughly covered in Deliverable 2.3. For the sake of completeness, we summarise some of the main aspects here. The Data Curator receives NGSI-LD entities via MQTT, even though it is prepared to also extract the data from the Scorpio Broker directly. It performs several functionalities for each of the to-be-curated entities, such as novelty detection, interpolation or calculation of data quality properties. The new information obtained is then appended to the now curated entity, or in most cases, stored in a new Data Quality entity linked with the original. Finally, these new entities are pushed into the Scorpio Broker via its standard REST interfaces.

3.4 WEB DATA INJECTION CHAIN

The web data Injection Chain is composed of four phases: Data Collection, Data Mapping, Data Enriching and Data Publishing (also Curation). The additional step of Data Enriching, compared to the IoTDIC, results from the compilation of the KPI entity, which is a completely new entity, but results as an enriching of the organization entity with additional information: a description of the performance regarding the compliance to the SDGs, presented as KPI. Another key difference that sets the Web Crawling Injection Chain apart from others is the need for additional data storage apart from the Context Broker itself. The web-crawled data that is used in the Agenda Analytics application will not be written directly into the Broker. Instead, a project external solution comprised of mongodb[17] and seaweedfs[18] (for large data files) is used to hold the raw web data and resulting converted/cleaned data for further processing by other services. The Broker will contain links to persistent data locations, if the access for the user is intended (reason could be the increased transparency of the calculation basis of a KPI or the need for further references to enable full understanding of an entity).

The following sections will give further insight into the implementations of the four phases. All have their origin in the top-down approach with Agenda Analytics as first use case in mind. Anyhow, they consider the following use cases in their setup and are therefore flexible to additions. The setup of those services is realised with Python and the *FastAPI* framework[19]. For the upcoming implementation of orchestration possibilities, the Python package *fastapi-mqtt*[20] will be used. At the moment, all services are triggered manually, whereas the output of one service can be used as input for the next one in the IC.

Data Collection

The collection phase identifies the data sources needed for the implementation of Agenda Analytics.

The centre of analysis is a real-world organization. This results in the need for identifying and collecting data which can initially be used to represent organizations of interest themselves. The Data Collector is therefore a service called DiscoverAndStore, since it leverages multiple ways within the web to obtain this data needed and stores it. To enable a decoupling of the broker at this point, a PostgreSQL database is chosen as point of agglomeration, which gets updated every time the service runs. The service offers multiple endpoints for searching new organizations and enriching the PostgreSQL database, each with a data source in mind. Those endpoints can be triggered with a GET-request. The response will be list of obtained/enriched database records in the format of JSON entities. For the automated use of organizations (database records) already identified, or the manual interaction with these, CRUD endpoints are supplied as well.

At this moment two search endpoints are implemented. The first leverages the Open Street Map API (OSM API) via GET requests to obtain relevant organizations, the second leverages a GET request to a human identified Wikipedia table of interest. Both endpoints perform further

[17] https://www.mongodb.com/
[18] https://github.com/seaweedfs/seaweedfs
[19] https://fastapi.tiangolo.com/
[20] https://pypi.org/project/fastapi-mqtt/

processing to obtain all necessary data (at least the name) to create a JSON representation for an organization before they check the PostgreSQL database if a new record can be created or if an update must take place.

Similarly, two enriching endpoints are implemented at the moment, which try to enrich the representations generated by the searching endpoint, the database records. The first leverages the crawling of Google results obtained by a GET request to identify the domain URL of an organization. The second leverages the OSM API again, which has a special impact if the original database record was created by the Wikipedia related search because the OSM enriching can then add location information.

This setup of the service enables a certain flexibility for adding further collection possibilities or enriching possibilities.

Data Mapping

The top-down approach for web data results in the fact that entity types are known, and we explicitly search for data needed to form a valuable representation of the entity of interest. This makes it possible to use *JMESPath* templates for all entity types/data models needed.

At the moment, there is one service called Mapping, that offers multiple endpoints - one for each differing data source. But generally, all endpoints accept a list of JSON and transform each list item to NGSI-LD, resulting in a list of NGSI-LD items as output. Data sources are at the moment the output of the DiscoverAndStore service and the MRN broker.

Data Enriching

For the Agenda Analytics use case, each entity of type Organization present in the broker has to be augmented with a related entity of type KPI. Thus, a first Enricher is needed that crawls the public web searching, identifying and collecting information on the sustainability efforts taken by one specific organization. This information will be later used in the compliance computations regarding the SDGs, as reference agenda of interest. This service is called Crawling. The input of this service is a list of NGSI-LD entities of the type organization, for which the data should be crawled, and a keywords parameter which specifies further what data is looked for (e.g. “sustainability report pdf”). This approach follows the result of an evaluation phase, which concluded that the best calculation basis can be obtained when first a GET request is made using Google to obtain the first 3 results for the query “<organization domain url> <keywords>”. The service then checks if the obtained results are of the PDF format and have a valid URL before these URLs are crawled by the Stormcrawler. The Stormcrawler is used to ensure a scalability of this approach when further applications require a crawling with more depth or a crawling of more sites. The crawled documents are stored in a *mongodb* and further in the *seaweedfs*, which belong to the Stormcrawler setup as file storage/access point for download. The file storage implements the concept of the originator, which in this case is the entity id of the input organization. This ensures a connection of an organization and crawled documents that can later be used by other services. Depending on crawling concept/intention of use and actuality requirements of crawled data a concept of entity and time can also be implemented as originator. The output of the service gives feedback in form of a status if the

crawling was successful or if an error occurred. This service can be triggered by a POST request, where the body holds the list of organization entities, and the keywords are given as parameter.

The second Enricher needed for Agenda Analytics is the one that processes the crawled information, performs the actual compliance matching and creates the resulting KPI entity. This Enricher is called AgendaMatching. The input of this service is the entity id of an organization for which the computation should take place. First the service checks if the file storage used by the Stormcrawler holds PDF documents for this specific organization. If yes, those documents are converted into text and pre-processed so they can be used as input to the ComplianceService that computes the compliance scores regarding the SDGs. The results are used to create a KPI entity which is also the output of this service. The KPI additionally contains, as specified in the last section, the links to the persistent storage location for the calculation base and the link to the originator organisation within the property "organization". This service can be triggered by a GET request, where the entity id is given as parameter.

Data Curation / Publishing

In addition to the above described services, a Publish service was implemented that handles the interface to the Scorpio Broker. Entities that result as output of the Mapping service (types: Organization, BikeHireDockingStation, EVChargingStation) and entities that result as output of the AgendaMatching service (type: KeyPerformanceIndicator) are supplied as list of NGSI-LD entities to this service's publish endpoint, when they should be added within the partner's broker.

The service handles the curation process, where it checks if each entity is already present in the broker (a rule-based system is used, emerging from known entity types in the Broker: e. g. name property for organization). Depending on the situation either the entity is added as a whole, or an already existing entity gets enriched with new properties, or already present properties get updated.

Besides this publish endpoint, the Publish service also offers endpoints that hold the default context for each entity type within this specific partner's broker, which can be used by further services that want to query the Broker.

Deployment Specification

All services explained above are running on an Ubuntu 22.04 Virtual Machine. The services are further deployed via *Jenkins*. *Jenkins* clones the code from the GitHub repository directly, to ensure transparency of code versions deployed.

3.5 IoT Injection Chain – Madrid and Dublin

3.5.1 Madrid city

In the context of IoT Injection Chains, we are working with the mapping of IoT sensed data of Madrid city to the available smart data models offered by FIWARE (essentially, Traffic Flow Observed and Air Quality Observed). There is the possibility to integrate other relevant models

that technically match the data, for instance, the Device data model. At the start, the collected data in form of CSV batches of a certain period, corresponding to the specific months before the Covid era and after, is under consideration for conversion into NGSI-LD format and sent to the broker agents.

This data may not exactly adhere to the naming convention as defined in existing models discussed above but a naming mapping (python dictionaries defined over the relevant attributes) is used to reflect the meaning of any data column, close to the defined components/attributes in the existing model. For the pre-processing of data, it is loaded in the Pandas library and tested for various test case scenarios, for example data validity. We can also say sensor information is valid if it is collecting the data over the specified periodic time slots. To normalize these CSV batch files or Pandas-loaded DataFrames in form of NGSI-LD, we have tried python scripts to introduce all the relevant and compulsory dependencies of the model (i.e., context, URN) and following a few examples as well shared by the FIWARE platform[21].

Similarly, pre-processing for the Pandas library loaded Madrid sensed data is also ongoing to find out the outliers in data. However, a straightforward approach to map the data from Pandas series/data frames to the model is not suitable as per our observations (for instance, the collected sensed data is loaded in single column of the CSV file, available on website), and need to be programmatically treated. A sample script is shown in Figure 23.

```
def Splitting_and_concatination_of_CSV_Columns_single_tuple_first():
    Traffic_data_directory = glob.glob(
        os.path.join("sample data for processing to NGSILD model/Madrid Traffic data", "*.csv"))
    Traffic_sensor_information = glob.glob(
        os.path.join("sample data for processing to NGSILD model\Madrid Traffic data\sesor information", "*.xlsx"))
    for f in Traffic_data_directory:
        df = pd.read_csv(f)
        user_defined_dict_sensed_data_attributes_inCSV = {0: "id", 1: "fecha", 2: "tipo_elem", 3: "intensidad",
                                                          4: "ocupacion", 5: "carga", 6: "vmed", 7: "error",
                                                          8: "periodo_integracion"}
        splitted_column_data_frame = df[
            'id;"fecha";"tipo_elem";"intensidad";"ocupacion";"carga";"vmed";"error";"periodo_integracion"'].str.split(
            ";", expand=True).rename(user_defined_dict_sensed_data_attributes_inCSV, axis=1)

        print("before index setting splitted_column_data_frame_sensed_data.colums",
              splitted_column_data_frame_sensed_data.columns)
        splitted_column_data_frame = splitted_column_data_frame.set_index("id")
        print("after index setting splitted_column_data_frame_sensed_data.colums",
              splitted_column_data_frame_sensed_data.columns)
        print("printing the type of ee", type(splitted_column_data_frame))
        splitted_column_data_frame.to_csv(r'splited.csv')
        for sens_info in Traffic_sensor_information:
            print("\n", sens_info)
            sensed_info_DF = pd.read_excel(sens_info)
            print(sensed_info_DF.columns)
            sensed_info_DF = sensed_info_DF.set_index("id")

        return splitted_column_data_frame, sensed_info_DF

splitted_column_data_frame_sensed_data, splitted_sensor_info_DF = Splitting_and_concatination_of_CSV_Columns_single_tuple_first()
```

Figure 23. Script for converting raw data in CSV format to JSON

Secondly, there is a need to integrate the data from the multiple relevant files. For instance, the sensed data need to include the sensor metadata from any other sources and validate if the collected JSON is close to the architecture of the developed model. To this purpose, we have considered to develop a dictionary based on the corresponding data model (FIWARE Traffic Flow Observed), while an iterator on the sensed data puts the relevant values into the specified dictionary. The template dictionary is collected to a python-based list, afterwards the list-

[21] https://github.com/smart-data-models/dataModel.Transportation/tree/db1e11c396a7b5f6e251bab27d56b38a26068a35/TrafficFlowObserved/examples

iterator converts it in to JSON-LD format. A software function that takes the data from sensed data and sensor information (both loaded in Pandas Data frames) can be seen in Figure 24.

```
def traffic_flow_template_dict_func(splitted_column_data_frame_sensed_data, splitted_sensor_info_DF):
    aggregated_list_of_data_template_loaded_dict = []
    traffic_flow_dict_template = dict()
    for splitted_column_data_frame_sensed_data_keys, splitted_column_data_frame_sensed_data_values in splitted_column_data_frame_sensed_data.iterrows():
        print("in loop", repr(splitted_column_data_frame_sensed_data_keys),
              type(splitted_column_data_frame_sensed_data_keys))
        if int(splitted_column_data_frame_sensed_data_keys) in splitted_sensor_info_DF.index:
            print("\n\n\n true*****that ", splitted_column_data_frame_sensed_data_keys, "is in ",
                  "splitted_sensor_info_DF.index")
            print(splitted_column_data_frame_sensed_data.loc[splitted_column_data_frame_sensed_data_keys]["fecha"],
                  repr(splitted_column_data_frame_sensed_data.loc[splitted_column_data_frame_sensed_data_keys]["vmed"]))
            traffic_flow_dict_template[
                "id"] = "urn:ngsi-ld:TrafficFlowObserved:TrafficFlowObserved:" + splitted_column_data_frame_sensed_data_keys
            print("\ntraffic_flow_dict_template[id]", traffic_flow_dict_template["id"])
            traffic_flow_dict_template["dateObserved"] = {"type": "string", "value":
                splitted_column_data_frame_sensed_data.loc[splitted_column_data_frame_sensed_data_keys]["fecha"]}
            traffic_flow_dict_template["averageVehicleSpeed"] = {"type": "number", "value":
                splitted_column_data_frame_sensed_data.loc[splitted_column_data_frame_sensed_data_keys]["vmed"]}
            print("\n\n\n\n splitted_sensor_info_DF.loc[splitted_column_data_frame_sensed_data_keys]",
                  splitted_sensor_info_DF.loc[int(splitted_column_data_frame_sensed_data_keys)])
            traffic_flow_dict_template["refRoadSegment"] = {"type": "number", "value":
                splitted_sensor_info_DF.loc[int(splitted_column_data_frame_sensed_data_keys)]["distrito"]}
            traffic_flow_dict_template["@context"] = list()
            traffic_flow_dict_template["@context"].append(
                splitted_column_data_frame_sensed_data.loc[splitted_column_data_frame_sensed_data_keys]["@context_0"])
            traffic_flow_dict_template["@context"].append(
                splitted_column_data_frame_sensed_data.loc[splitted_column_data_frame_sensed_data_keys]["@context_1"])
            traffic_flow_dict_template["type"] = \
            splitted_column_data_frame_sensed_data.loc[splitted_column_data_frame_sensed_data_keys]["type"]
            aggregated_list_of_data_template_loaded_dict.append(traffic_flow_dict_template)
    return aggregated_list_of_data_template_loaded_dict
aggregated_list_of_data_template_loaded_dict = traffic_flow_template_dict_func(splitted_column_data_frame_sensed_data,
                                                                               splitted_sensor_info_DF)
```

Figure 24. A piece of script for converting sensor data to JSON-LD format

To normalize these CSV batch files or Pandas-loaded data frames in JSON-LD format, we have included all the relevant properties of the Smart Data Model (i.e., context, URN) with a Python script. An example case of converted data from Madrid city traffic into dictionary (directly exportable to JSON-LD) is shown in Figure 25.

```
id : urn:ngsi-ld:TrafficFlowObserved:TrafficFlowObserved:3762
dateObserved : {'type': 'string', 'value': '"2019-04-03 07:30:00"'}
averageVehicleSpeed : {'type': 'number', 'value': '"76"'}
refRoadSegment : {'type': 'number', 'value': 12.0}
@context : ['https://raw.githubusercontent.com/smart-data-models/dataModel.Transportation/master/context.jsonld', 'https://ur
i.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld']
type : TrafficFlowObserved


id : urn:ngsi-ld:TrafficFlowObserved:TrafficFlowObserved:3762
dateObserved : {'type': 'string', 'value': '"2019-04-03 07:30:00"'}
averageVehicleSpeed : {'type': 'number', 'value': '"76"'}
refRoadSegment : {'type': 'number', 'value': 12.0}
@context : ['https://raw.githubusercontent.com/smart-data-models/dataModel.Transportation/master/context.jsonld', 'https://ur
i.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld']
type : TrafficFlowObserved
```

Figure 25. A JSON-LD instance of traffic data of Madrid city

3.5.2 Dublin city

For the Dublin city, data of traffic and air pollution are being processed following the conversion and mapping flow represented in Figure 26.

Methodology

- Collect and organize data using a data collection method.
- Use the Pandas library in Python to clean and process the data in a Pandas DataFrame.
- Map the data into NGSI-LD entities.
    - Use the JSON library in Python to convert the DataFrame to a JSON object.
    - Add the NGSI-LD context and structure to the JSON object.

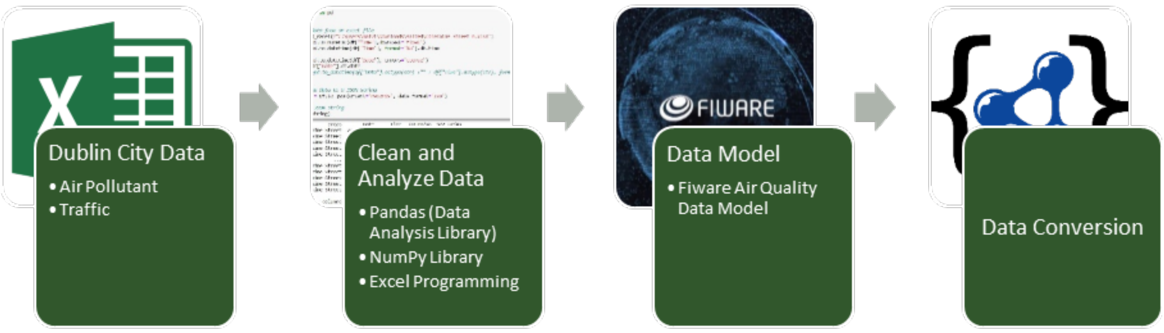


Figure 26. Dublin data conversion and mapping process

Collect and Organize Data

For the mapping purpose of air pollution data of Dublin, we collect the data from the *smartdublin* website[22] in CSV format, which is a common format for storing tabular data. Data include a key air pollutant, such as particular matter (PM 2.5 and PM 10), Nitrogen dioxide (NO_2), Sulphur dioxide (SO_2) and Carbon monoxide (CO), including values for different areas for the years of 2011 and 2012. We used the python script and EXCEL VBA code to extract the data from the source. A sample of the script is shown in Figure 27. We used the VBA code to define the data as an array or collection, by adding two libraries (Microsoft HTML Client Library and Microsoft Internet Control) and combining the all data files using MACROS in VBA.

```
import csv
import requests

url = "https://data.smartdublin.ie/dataset/8cfd5686-4d69-41d9-9fd4-e128f0b811b5/resource/eefbef8f-7c41-4281-a449-877a7676410c/

response = requests.get(url)
print(response.content)
open("dublin_pollution.csv", "wb").write(response.content)
```

```
b'Please provide hourly results in ug/m3 of PM 10 monitoring conducted in Dublin City in a tabular format by completing the t
able below. ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\nPlease provide a flag number (see comment for list) for th
e data entered if possible. The comment field is optional.,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\nIf any new s
tations have been added to the network please give details (scroll across).
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\nPlease indicate if any of the stations listed below have been reomved
from the network.,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\nDublin City Counci
l,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\nSite,Marino,,,Coleraine St.,,,Phoenix Park,,,Wood Quay (Winetavern S
t.),,,Rathmines,,,Ballyfermot,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\nDate,PM2.5(\xb5g m-3),Flag,Comment ,PM2.5 (\xb5g m-3),Fla
g,Comment ,PM10 (\xb5g m-3),Flag,Comment ,PM10 (\xb5g m-3),Flag,Comment ,PM10 (\xb5g m-3),Flag,Comment ,PM10 (\xb5g m-3),Fla
g,Comment ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n01-Jan-11,16.8,1,,17,1,,14.7,1,,20.7,1,,19.7,
1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n02-Jan-11,12.6,1,,8.9,1,,10.7,1,,13.5,1,,16.1,
1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n03-Jan-11,25.1,1,,18,1,,16.0,1,,23.1,1,,24.4,
1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n04-Jan-11,8.1,1,,9.3,1,,7.5,1,,10.8,1,,12.2,
1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n05-Jan-11,8.3,1,,10.1,1,,8.1,1,,13.8,1,,data missing,
6,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n06-Jan-11,24.6,1,,32,1,,17.6,1,,37.4,1,,data missing,
6,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n07-Jan-11,14.3,1,,19.7,1,,16.4,1,,28.6,1,,39.6,
1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n08-Jan-11,12.5,1,,10.1,1,,10.4,1,,9.6,1,,12.9,
1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n09-Jan-11,9.9,1,,12,1,,8.7,1,,13.5,1,,15.1,1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r
\n10-Jan-11,11.0,1,,11.6,1,,8.7,1,,13.5,1,,12.6,1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n11-Jan-11,48.7,1,,11.4,1,,4.7,1,,1
```

Figure 27. Reading data from Dublin data portal

Clean and Process Data

Once the data are collected, they need to be cleaned and pre-processed to ensure that they are suitable for analysis. This includes checking for missing or inconsistent data, and transforming the data into a format that can be easily analysed. We use Python with some libraries, like Pandas for reading and writing, and numpy for managing arrays. We load the CSV by using Pandas on a Jupyter notebook and perform data cleaning and extraction using the installed libraries for the conversion of data into a suitable format. We extract the relevant data from the Pandas DataFrame using *iloc* method and created a new DataFrame.

[22] https://data.smartdublin.ie

Data Mapping and Conversion

In order to get the data in NGSI-LD, we perform data mapping as a crucial step. An entity in the NGSI-LD information model represents a real-world object or concept and has a set of attributes that describe its properties. We used the FIWARE data models[23] Air Quality Observed and Traffic Flow Observed as reference models for air pollution and traffic analysis. We define the data models using NGSI-LD ontology. This ontology defines the basic entities and properties that are used to represent and manage data in a standardised way. We mapped each row in the file to an entity and columns to the attributes of that entity. The whole process is done in a Jupyter notebook by using a Python script. We used PyLD and JSON library to transform the obtained dictionary into JSON format. We added a context to the JSON data by adding an "@context" property. For the conversion from JSON to NGSI-LD format, we used a JSON-LD library that automatically transforms the data based on the mapping defined in the data modelling process. Figure 28 below shows one sample of a converted AirQualityObserved entity. An analogous approach can be adopted in other cities that have similar data.

[23] https://fiware-datamodels.readthedocs.io

```
{
  "id": "urn:ngsi-ld:Data:Coleraine-Street:b009",
  "type": "Data",
  "address": {
    "type": "Property",
    "value": {
      "addressCountry": "Ireland",
      "addressLocality": "Dublin",
      "streetAddress": "Coleraine Street",
      "type": "PostalAddress"
    }
  },
  "Date": {
    "type": "DateTime",
    "value": "2011-01-01T01:00:00.000Z"
  },
  "CO": {
    "type": "Property",
    "value": 0.5162926829,
    "unitCode": "mg/m3"
  },
  "SO2": {
    "type": "Property",
    "value": 2.2,
    "unitCode": "ug/m3"
  },
  "No2": {
    "type": "Property",
    "value": 23.6559648069,
    "unitCode": "ug/m3"
  },
  "No": {
    "type": "Property",
    "value": 5.2615991416,
    "unitCode": "ug/m3"
  },
```


Figure 28. Converted Data to NGSI-LD (Dublin)

3.6 Social Media Data Injection Chain

Data Collection

The first step is data discovery and data collection. Currently, our focus is on the collection of data from Twitter. Based on the needs we have two approaches: collecting tweets through the use of specific hashtags or gathering tweets from specific public accounts.

To facilitate these data collection efforts, we begin by importing the Tweepy[24] library and setting up our Twitter API credentials. This allows us to create an API object which we can use to access the Twitter platform. When working with hashtags, we make use of the *api.search* function to extract tweet objects containing the relevant hashtags, and when working with a list

[24] https://www.tweepy.org/

of user accounts, we use *api.user_timeline* to retrieve the required tweets. Figure 29 shows this implementation.

While we have the option of specifying a limit for the number of objects returned or retrieving as many as the Twitter API allows, it is important to note that there are restrictions associated with data retrieval. These limitations are based on a time interval or specified period, during which a maximum number of requests can be made. Moreover, there are limits on the number of objects that can be returned within each request. For the latest information on rate limits, we refer to the official Twitter API documentation[25].

```
# Collect tweets containing the hashtags
for hashtag in hashtags:
    tweets_hashtag = tweepy.Cursor(api.search,
                                   q=hashtag,
                                   tweet_mode='extended').items(limit_hashtagtweets)
    for tweet in tweets_hashtag:
        tweets.append(tweet)

# Collect tweets posted by the users
for user in users:
    tweets_user = tweepy.Cursor(api.user_timeline,
                                screen_name=user,
                                # since_id=start_date,
                                # until=end_date,
                                tweet_mode='extended').items(limit_usertimeline)
    for tweet in tweets_user:
        tweets.append(tweet)
```

Figure 29. Data collection script from Twitter based on Hashtag and User

To satisfy the requirements of certain data collection tasks, it may be necessary to establish a real-time stream of Twitter data. Tweepy offers a solution for this by providing a listener object that can be configured to monitor particular hashtags or user accounts using the "track" keyword for hashtags and the "follow" keyword for user accounts. This can be seen in Figure 30. Once the listener has been set up, the filter function can be used to retrieve real-time objects that match the specified filtering criteria. Again, it is important to consider the potential difficulties associated with collecting real-time data, such as rate limiting, and to establish appropriate measures to address them.

```
user_ids = []
for user in users:
    user_ids.append(str(api.get_user(screen_name=user).id))

# For monitoring specific accounts
myStream.filter(follow=user_ids, is_async=True)

# For monitoring specific hashtags
myStream.filter(track=keywords, is_async=True)
```

Figure 30. Real-time data collection script from Twitter

[25] https://developer.twitter.com/en/docs/twitter-api/rate-limits#:~:text=The%20maximum%20number%20of%20requests,15%2Dminute%20interval%20is%20allowed.

Data Mapping

Once the tweet objects have been obtained, the next step is to map the collected data to the NGSI-LD format. As outlined in the D2.1 document, to achieve this, the Smart Data Model for Social Media[26] is used, which features several entities including SMAnalysis, SMCollection, SMPost, SMRefLocation, and SMUser.

A tweet object includes numerous 'root-level' fields such as id, text, and created_at, and is also the 'parent' object for several child objects such as user, media, poll, and place. However, out of all the fields in the retrieved tweet objects, a significant number of them may be redundant or unnecessary, and may even be filled with nearly always empty or unhelpful values. It is essential to carefully evaluate and select the relevant fields to use as properties in the final NGSI-LD entity, which can optimize data mapping and processing, and enhance the overall efficiency of data analysis.

Therefore, in the initial phase, some basic fields from the tweet object have been selected to be used as properties of the related entities in NGSI-LD. However, as the project progresses and more specific needs and requirements are identified, additional time and effort will be required to determine which fields should be used as properties and which ones are likely to be redundant or unnecessary to keep. This will help to ensure that the collected data is accurately represented and effectively utilised within the NGSI-LD format, enabling more efficient and effective data processing and analysis. The official Twitter API documentation provides a comprehensive list of all the available fields. This API data dictionary[27] can serve as a helpful reference for selecting and utilising the relevant fields as NGSI-LD properties.

Figure 31 and Figure 32 display the SMPost and SMUser entities, respectively, with their corresponding properties and relationships. The tweet object used for the mapping was retrieved from Leonardo DiCaprio's account, with specific fields selected for the NGSI-LD data model. Additional properties are to be discussed and incorporated.


```
{'id': 'urn:ngsi-ld:SMPost:123',
 'type': 'SMPost',
 'belongsToCollection': [{'type': 'Relationship',
   'object': 'urn:ngsi-ld:SMCollection:123',
   'datasetId': 'urn:ngsi-ld:Dataset:SMCollection:123'}],
 'createdBy': {'type': 'Relationship', 'object': 'urn:ngsi-ld:SMUser:123'},
 'hasAnalysis': [],
 'hasHashtags': [{'type': 'Property', 'value': []}],
 'hasText': [{'type': 'Property',
   'value': 'Indigenous peoples safeguard 80% of our planet's biodiversity &amp;amp; are keepers of traditional knowledge
ensuring Indigenous voices lead the climate fight.     \n \nhttps://t.co/gdnhntR0lD'}],
 'hasInteractionCount': [{'type': 'Property',
   'value': {'@interactionType': 'Like', '@count': 2849}},
  {'type': 'Property',
   'value': {'@interactionType': 'Retweet', '@count': 646}}],
 'platform': {'type': 'Property', 'value': 'Twitter'},
 'postCreatedAt': {'type': 'Property',
  'value': {'@type': 'DateTime',
   '@value': datetime.datetime(2022, 11, 17, 19, 10, 16)}},
 'postId': {'type': 'Property', 'value': '1593320635022729217'},
 '@context': ['https://raw.githubusercontent.com/smart-data-models/dataModel.SocialMedia/master/context.jsonld']}
```


Figure 31. Converted data to JSON-LD for twitter (SMPost entity)

[26] https://github.com/smart-data-models/dataModel.SocialMedia
[27] https://developer.twitter.com/en/docs/twitter-api/data-dictionary/object-model/tweet

```
{'id': 'urn:ngsi-ld:SMUser:123',
 'type': 'SMUser',
 'platform': {'type': 'Property', 'value': 'Twitter'},
 'userId': {'type': 'Property', 'value': '133880286'},
 'userName': {'type': 'Property', 'value': 'Leonardo DiCaprio'},
 'createdPosts': [{'type': 'Relationship',
   'object': 'urn:ngsi-ld:SMPost:123',
   'datasetId': 'urn:ngsi-ld:Dataset:SMPost:123'}],
 '@context': ['https://raw.githubusercontent.com/smart-data-models/dataModel.SocialMedia/master/context.jsonld']}
```


Figure 32. Converted data to JSON-LD for twitter (SMUser entity)

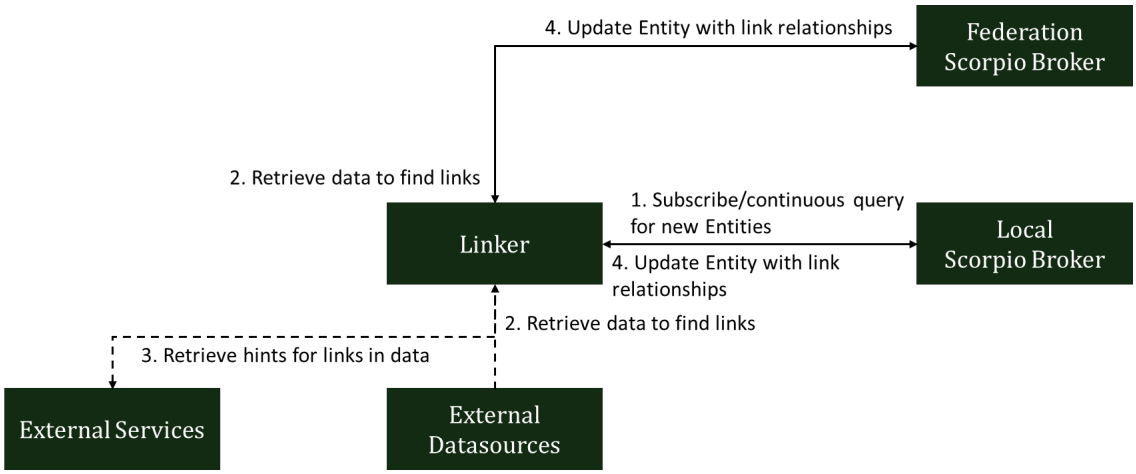
This project is co-financed by the Connecting Europe Facility of the European Union under the Action Number 2020-EU-IA-0274.

4 DET LINKERS

4.1 OVERVIEW

In the SALTED project, a Linker is in general a component that generates NGSI-LD Relationships between two or more NGSI-LD Entities. The generated Relationship will represent a link between the Entities. The meaning of the **link** is represented in the Relationship name and the used data model. Link generation is not restricted in any way but common ways are to discover links are:

- by finding commonalities in the data of the entities.
- by finding spatial associations.
- by finding commonalities in the data model of entities.



The SALTED architecture also allows Linkers to use external data sources or services to discover links.

External services can provide associations between Entities based on the values of their Properties allowing a Linker to generate new links.

Since we have to expect that not always all the data is necessarily available in the NGSI-LD format, Linkers are also able to generate new Entities on the fly in case they discover new links in the external data.

4.2 IOT DATA LINKER

The IoT Data Linker (IoTDL) establishes relationships between entities whose device of origin is the same. In IoT infrastructures, it is common to see sensors, or devices in general, that generate different types of measurements. For instance, a SmartSantander temperature sensor generates both temperature and battery status measurements. However, these two entities are not linked in any way by default. The aim of the IoTDL is to find these situations, as shown in Figure *33*, and link the entities via an NGSI-LD relationship.

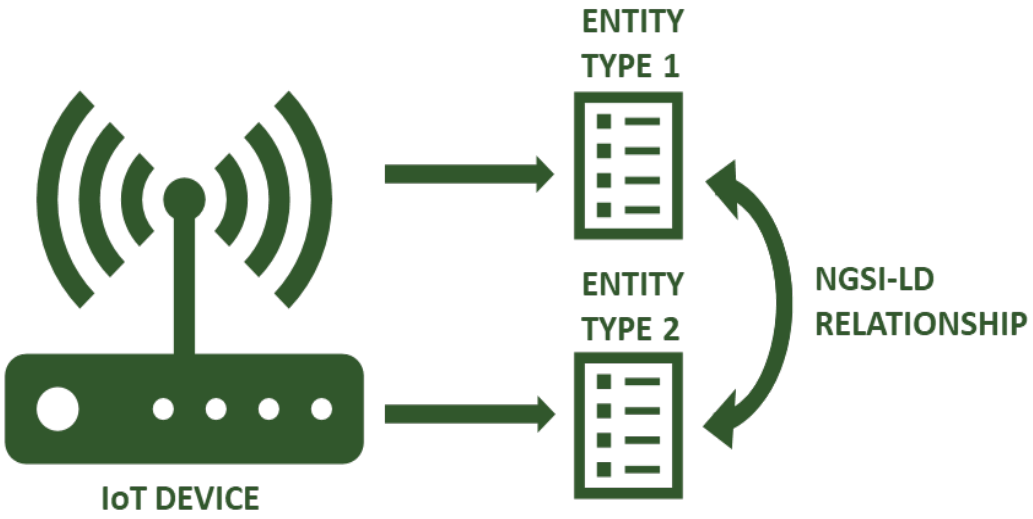


Figure 33. IoT Data Linker use case

The IoTDL works directly with data extracted from the Scorpio Broker. It can perform queries to obtain batches of data through the standard interface (an HTTP GET request to the */entities* endpoint) or subscribe to the types of data it is interested in (through an HTTP POST request to the */subscriptions* endpoint). The current implementation of the IoTDL uses a combination of both. It is written in Python, and uses the *json* library for data handling, *requests* for communication, and the *flask* and *waitress* libraries to set up the server that receives notifications from the Scorpio Broker.

The NGSI-LD entities are processed individually. The IoTDL is able to recognise entities coming from the same physical device is thanks to the *id* scheme used in the SALTED project. The entity *id* is structured in such a way that a query filter can be applied using regular expressions. The response will include entities whose *id* matches the query. In the case of the IoTDL, we can extract entities with the same device of origin as long as the original collected entities include an identification for this device. Then, we insert the related entities in an NGSI-LD relationship, creating the link.

Lastly, the NGSI-LD entity, which is now linked, is reinjected into the Scorpio Broker via an HTTP POST request to the */entities* endpoint.

The IoTDL is deployed in an Ubuntu 20.04.5 LTS Virtual Machine, and the *cron* tool is scheduling the periodic requests and performing a health check on it to make sure it is working properly.

4.3 GEOLOCATION DATA LINKER

The Geolocation Data Linker (GDL) links NGSI-LD entities to others located within a certain distance. Normally, there is no simple way to know which other entities are close to a given entity. The only possibility is to check the coordinates, manually prepare the range or the area, and perform a series of very specific requests to the NGSI-LD Context Broker. The aim of the GDL is to greatly simplify this process by automating the necessary requests and linking the entities through an NGSI-LD relationship in advance. This helps all kinds of actors: applications, other linkers and enrichers, and even users simply skimming the data.

The GDL extracts the NGSI-LD entities directly from the Scorpio Broker, combining queries and subscriptions. It is written in Python, and uses the *requests* library for communication, the *json* library for data handling, and the *flask* and *waitress* libraries to set up the REST server for notifications.

Every NGSI-LD is processed individually. The GDL reads the location value of the entity and performs a geolocation-based query to the Scorpio Broker. The location of the entity is used as the centre of a circle, and the radius is obtained from a configuration parameter (called *distance*) that can be easily tuned. The response from the Broker is a list of entities within the desired range of the original location. Once obtained, we insert this list in an NGSI-LD relationship within the original entity, which completes the linking process.

Finally, the linked NGSI-LD entity is reinjected into the Scorpio Broker via an HTTP POST request to the */entities* endpoint.

The GDL is deployed in an Ubuntu 20.04.5 LTS Virtual Machine, and we use the *cron* tool to schedule the periodic requests and perform a health check on it to make sure it is up and running.

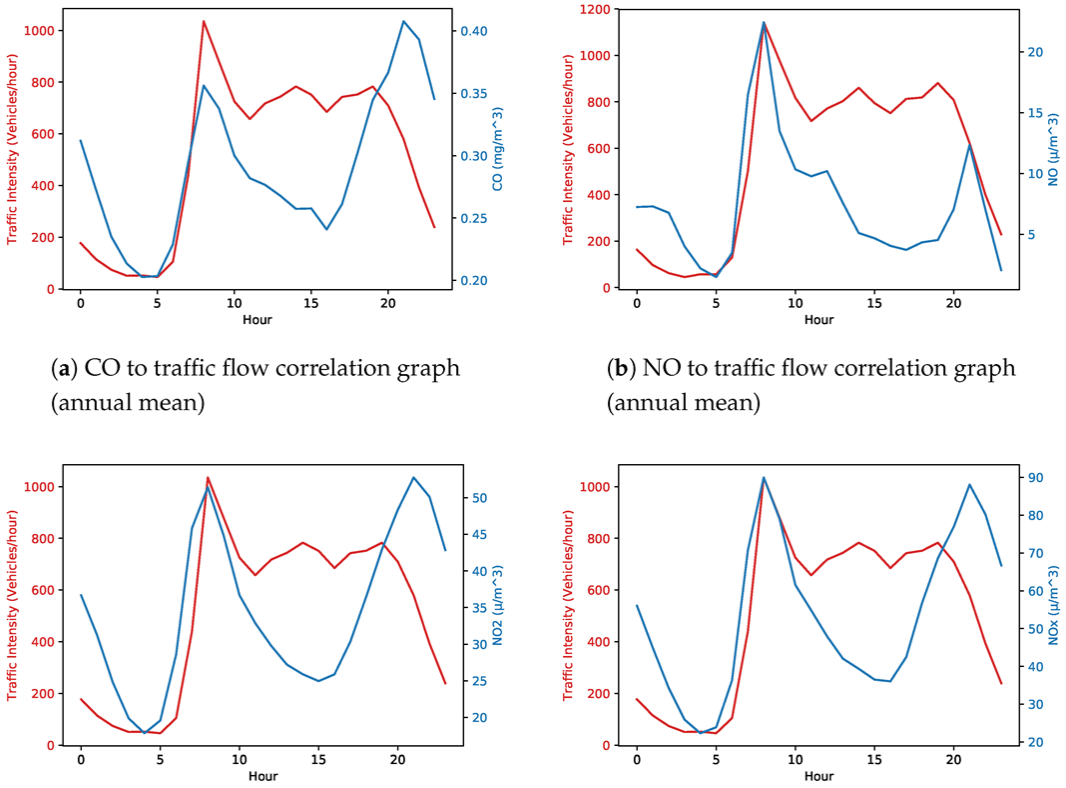
4.4 Web Data Linker

The top-down approach for web data results in the fact that we explicitly search for data needed to form a valuable representation of the entity types of interest. Further the generated entities are already filled with links, if they exist, because we know what we need to link. Of course, additional linkers (e.g. for geolocation) can additionally be used for further use case unrelated linking.

4.5 Correlation Linker

Smart Cities are collecting a large wealth of different datasets. They are intuitively related to each other. For instance, the meteorological data have an influence on how the pollution data should be interpreted: wind, temperature and pressure values clearly have an influence on many pollutants and how they spread on the environment. Traffic intensity is also one of the major contributors to the pollution. It is intuitively expected that for higher values of traffic intensity there will be a major concentration of pollutants. However, the city datasets reflect complex phenomena and it is difficult to find evidence and easy relationships between data of different datasets. There is the need to support, with tools and processes, the possibility to assess the correlation between phenomena and the data that represent them.

An example could be useful. Intensity traffic data should be strongly correlated to pollution data. Figure 34 (taken from [3]) shows the case of a comparison of a set of traffic intensity sensors and some values of pollutants taken from close pollution stations. These data represent the values of each hour of each day of the year 2019.



(a) CO to traffic flow correlation graph (annual mean)

(b) NO to traffic flow correlation graph (annual mean)

Figure 34: Correlation of traffic flow and air pollutants each hour of the day (annual mean)

As shown, the data well overlap up to a certain hour of the day and then the correlation seems to be weaker. As said, pollutants are very sensible to meteorological events and their behavior can change accordingly to different weather conditions.

Figure 35 shows the annual mean value of wind. Typically, in Madrid during the afternoon the wind almost doubles its intensity, having an impact on how pollutants spread in the environment.

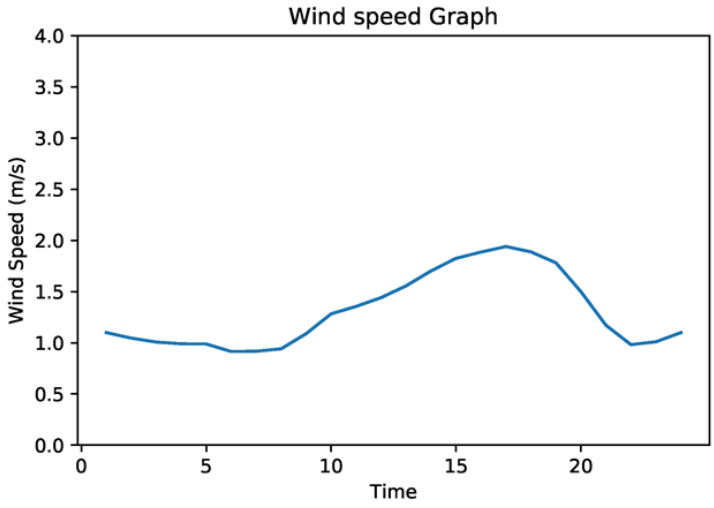


Figure 35. Wind variation in Madrid (Annual mean)

The yearly average values can be useful to understand the phenomena but are too weak to fully prove a real correlation between different phenomena.

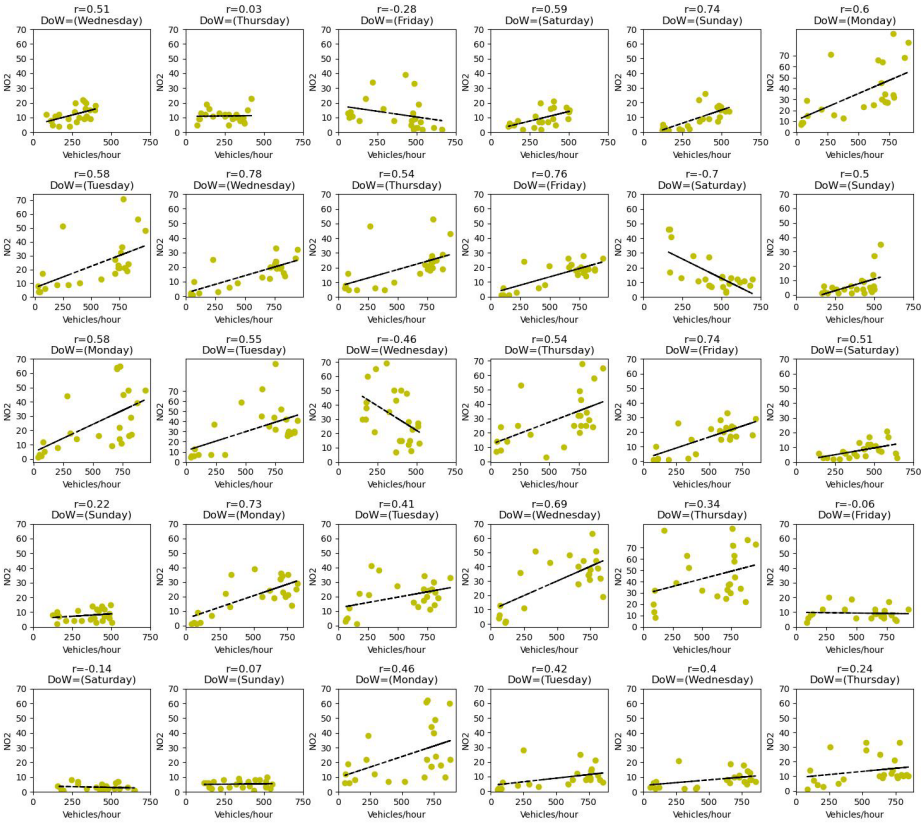


Figure 36. Monthly Correlation of traffic intensity data close to a pollution station (<500 m)

Figure 36 represents the calculation of the Pearson correlation for a small number of traffic intensity sensors within a distance of 500 m from a pollution station. The Pearson correlation coefficient (r) is a common way of measuring a linear correlation. It is a number between –1 and 1 that measures the strength and direction of the relationship between two variables. When one variable changes, the other variable changes in the same direction. Any value of r>=0.5 shows correlation exists between two variables. The higher the value, the stronger the relationship. From left to right and top to bottom of Figure 36, each plot represents individual day of May 2019.

Some Findings:

- For around 17 days, we observed moderate to strong correlation.
- For around 6 days, we observed a weak correlation.
- For around 7 days, we found almost no relationship.

In order to better correlate the phenomena, the calculation of p-value [4] can support the identification of strong correlation. In a simple analysis of the correlation between

meteorological values and pollution shown in Figure 37, the p-value was measured in order to understand if the assumptions hold.

TABLE VIII
CORRELATION COEFFICIENT BETWEEN METEOROLOGICAL FEATURES AND TRAFFIC FLOW WITH AIR POLLUTANT CONCENTRATION IN TIME DOMAIN

	NO			NOx			PM2.5			PM10		
	2019	2020	2022	2019	2020	2022	2019	2020	2022	2019	2020	2022
Solar Radiation	-0.08	0.12**	0.02	-0.14***	0.02	-0.13**	0.1*	0.1*	-0.05	0.14***	0.31***	-0.08
Temperature	-0.07	0.11**	0.04	-0.07	0.08	0	0.41***	0.3***	0.36***	0.39***	0.55***	0.37***
Wind Speed	-0.29***	-0.22***	-0.24***	-0.38***	-0.3***	-0.28***	-0.26***	-0.22***	-0.24***	-0.1**	-0.09*	-0.16***
Traffic Flow	-0.12**	0.27***	-0.24***	-0.26***	0.18***	-0.33***	0.05	-0.03	0.03	-0.11**	0.09*	-0.08

KEY FINDINGS

- Wind speed is highly correlated with all air pollutants in every year
- Temperature has significant positive correlation with PM2.5 and PM10 every year
- Traffic flow is correlated with only NO and NOx every year

Figure 37. Correlation study between traffic, pollution and meteo values on anual average in Madrid

The linkage of different datasets should be supported by evidence that the two data sets are correlated. Otherwise, the application of Neural Networks techniques or other AI based analysis could lead to misinterpretation and mistakes. On the other side, the process for analysing the correlations is very problem domain specific (e.g., traffic intensity, pollutions and meteorological data) that it should be sorted out during the problem definition and analysis. However, there are many tools that can be used in order to investigate the correlation. For instance the Pandas tools offers some functionalities related to statistical analysis, while some specific Python libraries (*scipy.stats* or *numpy*) can be used and exploited. In addition, data of the different datasets should be comparable in terms of timing. For instance, traffic intensity is measured in 15min slots while pollution in 60 min slots in Madrid. There is the need to prepare the data for the correlation analysis and then to wisely apply a few statistical techniques for determining the value of linking the data.

The point here is that SALTED can provide clean data that can be processed by simple functions (by means of additional tools) in order to assess the correlation and to prepare the datasets for the linking. The applications operating on the “well-formatted” data will take charge of the expected results.

4.6 SEMANTIC LINKER

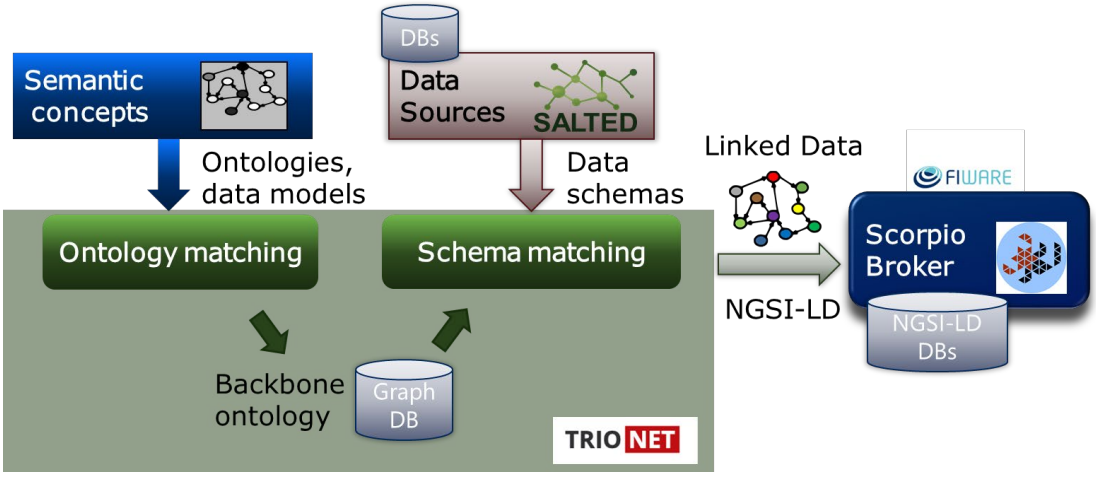


Figure 38. Data linking using NEC component TrioNet

The Semantic Linker is mapping relevant raw data sources with existing ontologies or data models such as the Smart Data Models. NEC previously developed a prototype called "TrioNet". Reference [5] explains the high-level architecture of this component and [6] includes experimental analysis for ontologies from the building domain. Deliverable D2.1 [1] includes the initial descriptions of this service with examples of linking between IoT data sets.

The data sources of SALTED can come from EDP or any other open/propriety databases. The sources are heterogeneous in their nature, for instance, smart city-related data may include various sensor data as well as records published online by the city government.

TrioNet is used for matching semantic concepts to each other and creating a "backbone ontology" that merges the relevant semantic concepts by the ontology matching. As example to semantic concepts, ontologies or data models can be considered to be merged. The backbone ontology can be created using only one or multiple ontologies or data models. This ontology is stored in a graph database. The TrioNet architecture is shown in Figure 38.

Having the backbone ontology, various data sources with existing schemas (e.g., CSV files, JSON files) can be annotated with the matching concepts in the backbone by the schema matching. There is not a requirement that every data source and all its schema-related entities can be matched to the backbone ontologies, whereas a subset of the entities can be matched with the existing backbone.

The above steps of ontology matching and schema matching is semi-automated. In the first step, a weakly-supervised machine learning algorithm is matching the relevant semantic concepts (ontology or schema concepts). Later, a human uses the estimations as well as the distance metrics provided by TrioNet for manual annotation.

After the annotation steps, the data linking converts the annotated data sources to the standard NGSI-LD data format. The NGSI-LD data is finally published to the FIWARE Scorpio Broker. FIWARE Scorpio Broker would make the data available to the subscribers or through asynchronous queries. The linked data by the semantic linker will include the semantic concepts (e.g., Ontology concept URI, link to the data model) along with the available data (e.g., sensor measurements).

5 CONCLUSIONS

In this document, we provide the D2.2 deliverable on the SALTED Data Modelling and Linking, as well as the current implementation and deployment of the architecture. Firstly, we describe the Federation setup of the SALTED architecture, including a thorough review of the implemented framework. Moreover, we outline the communication flows that can occur both internally (between DET components) and externally (when users or applications are involved). We also elaborate on the enforced security mechanisms protecting the SALTED architecture. These include the deployment of a KeyCloak instance and the development of a PEP proxy.

The Data Modelling segment is focused on the implementation of Injection Chains and the addition of new Smart Data Models to the SALTED ecosystem. In fact, we have contributed to this initiative with the DataQualityAssessment data model. The Injection Chains implemented, generally composed of Data Collection, Data Mapping and Data Curation, have been classified and comprehensively described according to the nature of the data sources.

Regarding the Data Linking, we provide a review of several DET Linkers following different approaches. The output NGSI-LD linked entities are a key contribution towards the expected goals of the SALTED project.

This deliverable shows the progress accomplished within the SALTED architecture and, more specifically, the DET components involved in the Data Modelling and Data Linking areas. The aim has been put in the implementation and deployment of the outlined DET components, as well as the key elements of the architecture. The NGSI-LD entities currently injected into the Scorpio Broker as a result of the mechanisms elucidated in this report, which will be available in the EDP as stated in the SALTED requirements, demonstrate an alignment with the expected outcome of the project.

6 BIBLIOGRAPHY

[1] SALTED, "D2.1: Report on Data Linking and Enrichment Architecture," 2022.

[2] Context Information Management (CIM) ETSI Industry Specification Group (ISG), "NGSI-LD API," 08 2022. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.06.01_60/gs_CIM009v010601p.pdf. [Accessed 13 02 2023].

[3] F. Awan, R. Minerva and N. Crespi, "Improving road traffic forecasting using air pollution and atmospheric data: Experiments based on LSTM recurrent neural networks," *Sensors,* 2020 Jul 4.

[4] M. Thiese, B. Ronna and U. Ott, "P value interpretations and considerations," *Journal of thoracic disease,* 2016 Sep.

[5] C. Garrido-Hidalgo, J. Fürst, B. Cheng, L. Roda-Sanchez, T. Olivares and E. Kovacs, "Interlinking the Brick Schema with Building Domain Ontologies," *Twentieth ACM Conference on Embedded Networked Sensor Systems,* pp. 1026-1030, 2022 Nov 6.

[6] G. Solmaz, F. Cirillo, J. Fürst, T. Jacobs, M. Bauer, E. Kovacs, J. R. Santana and L. Sánchez, "Enabling data spaces: existing developments and challenges," *1st International Workshop on Data Economy,* pp. 42-48, 2022 Dec 9.