IW-NET

Innovation-driven Collaborative European Inland Waterways Transport Network

D1.2 – Open Security Preserving Data and Services Connectivity Components - Federation of IWT Systems

Lead Beneficiary: Konnecta Systems Limited

Delivery Date: 29/5/2023

Dissemination Level: Public

Type: Report

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 861377.

Document Information

Title:	Innovation-driven Collaborative European Inland Waterways Transport Network
Acronym:	IW-NET
Call:	H2020-MG-2019-TwoStages
Type of Action:	RIA
Grant Number:	861377
Start date:	01 May 2020
Duration:	36 Months
URL	www.iw-net.eu

Deliverable

Title	D1.2 – Open Security Preserving Data and Services Connectivity Components - Federation of IWT Systems
Work Package	WP2: IWT Infrastructure improvements and TEN-T, Sea and Inland Ports Integration
Dissemination Level	Public
Delivery Date	29/5/2023
Lead Beneficiary	Konnecta Systems Limited (KCT)
Lead Authors	Nikos Chalvantzis (ICCS), Evie Kassela (ICCS), Harris Niavis (INLE), Dr. Aristea Zafeiropoulou (KCT)

Document History

Version	Date	Modifications	Contributors
0.1	20/06/2022	Created ToC	Nikos Chalvantzis, Evie Kassela, Harris Niavis, Aristea Zafeiropoulou
0.2	14/12/2022	Received contributions in chapters 2, 3	Nikos Chalvantzis, Evie Kassela
0.3	30/1/2023	Received contribution in chapter 4	Harris Niavis
0.4	3/2/2023	Added Summary	Aristea Zafeiropoulou
0.5	7/2/2023	Added Introduction, Conclusions, ready for peer-review	Nikos Chalvantzis, Aristea Zafeiropoulou
0.6	29/5/2023	Added feedback from peer review	Harris Niavis, Nikos Chalvantzis

Executive Summary

The "Innovation-driven Collaborative European Inland Waterways Transport Network" project is supported by the European Commission under the “Moving freight by Water: Sustainable Infrastructure and Innovative Vessels” topic of the Horizon 2020 research and innovation programme under grant agreement No 861377.

The present deliverable presents the work undertaken under Task 1.2 *Open security preserving data and services connectivity components - federation of IWT systems* that aims at fulfilling the project Objective (O1), i.e., the development of an open IWT digitalisation infrastructure and accompanying services for IWT integration in multimodal transport and urban logistics.

Work towards the fulfilment of this objective has been focused on three main aspects. The first has been devoted towards the development of a publish-subscribe mechanism via Apache Kafka that will enable the information sharing and storing within the IW-NET Big Data Analytics Platform.

The second part of the work deals with the secure identity management for secure communication and information sharing within the project. The developed solution, based on Keycloak, incorporates the security aspect within the IW-NET information exchange and is deployed on top of the event-based publish-subscribe solution.

The third part of work introduces Blockchain technology to the project by enabling trusted data exchange between IW stakeholders across the entire transport chain. In conjunction with Blockchain, the use of IoT devices supports live monitoring of logistics assets which in turn enhances cargo traceability. The integration of IoT and Blockchain enables the employment of smart contracts to automate processes and improve accountability of actions and logistics events.

All three solutions are brought together to constitute the open IWT digitalisation infrastructure that aims at applying the latest technologies in IT in the field of logistics. In this manner, logistics stakeholders embrace transparent, accessible, and automated operations through the use of the latest technological advances such as Big Data, Internet of Things, Blockchain and Artificial Intelligence.

Disclaimer

Table of Contents

List of Figures

List of Tables

Glossary of Terms and Abbreviations

Abbreviation / Term	Description
API	Application Programming Interface
BDA	Big Data Analytics platform
Dn.m	IW-NET Deliverable number
EPCIS	Electronic Product Code Information Services
IAM	Identity and Access Management
IO	Input/Output
IoT	Internet of Things
IWT	Inland Waterway Transport
JSON	JavaScript Object Notation
OAuth	Open Authorization
OS	Operating System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
REST	Representational state transfer
SAML	Security Assertion Markup Language
SSCC	Serial Shipping Container Code
TCP	Transmission Control Protocol
TI	Transport Instruction
TIR	Transport Instruction Response
TSN	Transport Status Notification
WP	Work Package

1 Introduction

1.1 Focus of the Deliverable

Work Package 1 focuses on the implementation of Digitalisation Infrastructure and Services that enable the application of the latest technologies in IT in the field of logistics. The vision is a business that embraces transparent, accessible, and automated operations assisted by recent technological advances such as Big Data, Internet of Things, Blockchain and Artificial Intelligence.

All the above are only possible given a trustworthy, reliable, and resilient technical infrastructure which has the technical capabilities to support demanding applications. This report documents the work undertaken towards building the components of the infrastructural stack that implement the message exchange/communication and security aspects of the WP1 technical solution. Additionally, it presents the development of automated and trustworthy smart contracts enabled via Blockchain technology and its integration with data coming from IoT devices.

The purpose of this document is to provide the reader with a good understanding of how and why certain system design decisions were made. In the cases of the Publish/Subscribe Architecture and the IoT Streaming integration with Blockchain, it highlights some of the most significant instances of message exchange between IW-NET architectural building blocks. More specifically, the former documents the internal workings of the most common communication channel, acting as a "messaging highway" and interconnecting all the parts of our design, while the latter sheds light on a very specific, one-to-one instance of communication between IoT and Blockchain. Finally, the section dedicated to the Secure Access and Identity Management component documents the workflows that allow us to secure and protect the sensitive services, data and digital resources which are an integral part of the project's outcomes.

The document is divided into separate chapters:

- Chapter 2 is dedicated to the architecture of the Publish-Subscribe service. A more detailed technical description of the system design and deployment is included in D1.6 *Big Data analytics linked with IWT corridor data hub Version 2*.
- Chapter 3 presents the Access and Identity Management solution developed in the IW-NET connectivity layer.
- In Chapter 4 the IW-NET Blockchain component is presented in detail along with its integration with IoT data. The smart contracts, the relevant APIs and the Blockchain infrastructure are explained further in D1.7 *Synchro-modality booking and execution management dashboard and architecture extensions, Dynamic optimisation Version 1*.
- Finally, a conclusion outlines the key takeaways from the work undertaken in Task 1.2.

1.2 Mapping IW-NET Outputs

The following table (Table 1) provides a map of the deliverable D1.2 and the corresponding task descriptions to the content of this document.

Table 1: Adherence to IW-NET's GA Deliverables & Task Descriptions

DELIVERABLE		
D1.2	Report on the IW-NET security federation layer, including Identity Management, and Blockchain to implement the IW-NET data governance, and Publish-Subscribe events architecture and components and the integration of the IoT data flows layer.	
TASKS		
T1.2 Open security preserving data and services connectivity components - federation of IWT systems Leader: KCT Participants: ISL, INLE, VLTN, NGS, KUL	Develop a Secure Services Federation layer integrating, extending and using open-source components to seamlessly and securely exchange information between infrastructures and systems, enabling data harmonisation in IWT, connectivity with Inland Port Management and Transportation Management systems. Apply and configure State-of-Art security options including secure access, identity management (IAM) and Blockchain, for reliable data transfers via Publish-Subscribe, and IoT data streaming.	
IW-NET GA Component Title	**IW-NET GA Component Outline**	**Respective Document Chapter(s)**
ST1.2.1 Secure Access and Identity Management (ICCS)	Configure and use Identity Management (IAM) solutions already developed in SELIS to automate secure access and a permissions system for all information accessing, exchanging, and routing. Configure and use Keycloak or equivalent federated solution for user authentication with single sign-on, with standard protocols such as OpenID Connect and OAuth. Develop a directory service for all IW-Net IAM. Adapt SELIS Community Nodes to link IW infrastructure data and securely deploy IW related services, including the secure storage of data with end user privacy safeguards. Enable secure communications between services, secure and private communication. Introduce end-to-end cryptographic protocols, to enable cyber-Secure solutions deployment and operation. Deliver a complete environment to include and support all IW-Net Business Cases and their security needs.	In this report we discuss our implementation of the IAM solution adopted in the context of IW-NET (Chapter 3). We deploy and configure a solution based on Keycloak and expose its role in the general architecture and its interactions with other key components.
ST1.2.2 Publish – Subscribe event-based Architecture (ICCS)	SELIS Publish-Subscribe architecture components will be integrated and form the backbone of asynchronous information exchanges in IW-Net, for performance and scalability. Different deployment topologies and configurations of the publish-subscribe system will be investigated depending on the different Business Case workload scenarios, mainly to integrate Infrastructure related information exchanges and the linking to back-	In this report we introduce the IW-NET implementation of a Publish/Subscribe service based on SELIS (Chapter 2). We highlight the strengths of our architecture

	end systems. Performance will be tuned to match the needs of the IW-Net overall system topology, identifying stressed services and components and possible bottlenecks. The IW-Net solution will be tested for scalability so to accommodate the required large numbers of data consumers and producers serving the transactions of the IW-Net users including their transport means and the near real-time message routing involving River Infrastructure notifications.	designs and discuss the reasons why a Publish-Subscribe message exchange service perfectly fits the use cases we encounter in the context of IW-NET. Our system design and deployment are consistent with that of the Big Data Analytics Platform described in deliverables D1.5 and D1.6, providing scalability and support for workloads of arbitrarily large volume.
ST1.2.3 IoT data Streaming Integration with Blockchain (INLE)	Use blockchain technology to improve processes and transactions along the whole transport chain as well as the visibility of shipments and the provenance of messages and notifications as a key enabler of synchro-modality. Analyse the needs of the transactions between the IWT actors in IW-Net Business Case scenarios and produce designs and configurations to integrate blockchain technologies so to assure auditability, immutability and governance of data sharing. Build on the Blockchain components developed in CHARIoT using distributed ledger technology to simplify, standardise and streamline interorganizational workflows and the non-repudiation of milestone events and notifications.	Chapter 4 is dedicated to the IW-NET Blockchain component.

2 Publish-Subscribe Event-based Architecture

A **Publish-Subscribe** messaging system is a fundamental building block in our envisioned IW-NET architecture. As the channel through which all communication between components is materialised, the list of functional and non-functional requirements is both long and critical in nature. The communication channel required to support the intricate and diverse structure of IW-NET must exhibit characteristics such as high availability, fault tolerance, and scalability.. In the following section, we discuss the way we have achieved our goal of designing and implementing a communication mechanism that meets the high standards set by our vision of IW-NET.

Publish-Subscribe messaging is a paradigm of asynchronous service-to-service communication, that originates from the older, more classical approach of Message Queues and is typically utilised in state-of-the-art serverless and micro-service architectures. The Publish/Subscribe pattern, also referred to as **Pub/Sub**, is a design pattern that provides a framework for message exchange between services. It involves a message broker that receives and relays messages from clients that publish data, known as *publishers* or message *producers*, to clients that consume it, known as *subscribers* or message *consumers*. **Publishers** can broadcast messages in various topics, while **subscribers** declare the topics they wish to subscribe to, to receive messages published there. Messages published to a topic are immediately received by all clients that have subscribed to it. The Pub/Sub message exchange paradigm is considered ideal for enabling event-driven architectures as well as decoupling applications to increase performance, reliability, and scalability.

In the rest of this section, we will introduce the basic concepts and principles of message exchange, discuss how those are implemented by Apache Kafka – an open-source, state-of-the-art software upon which the IW-NET Pub/Sub mechanism relies – and explain the interaction between the Pub/Sub mechanism and other architectural components of IW-NET (Figure 2-1).

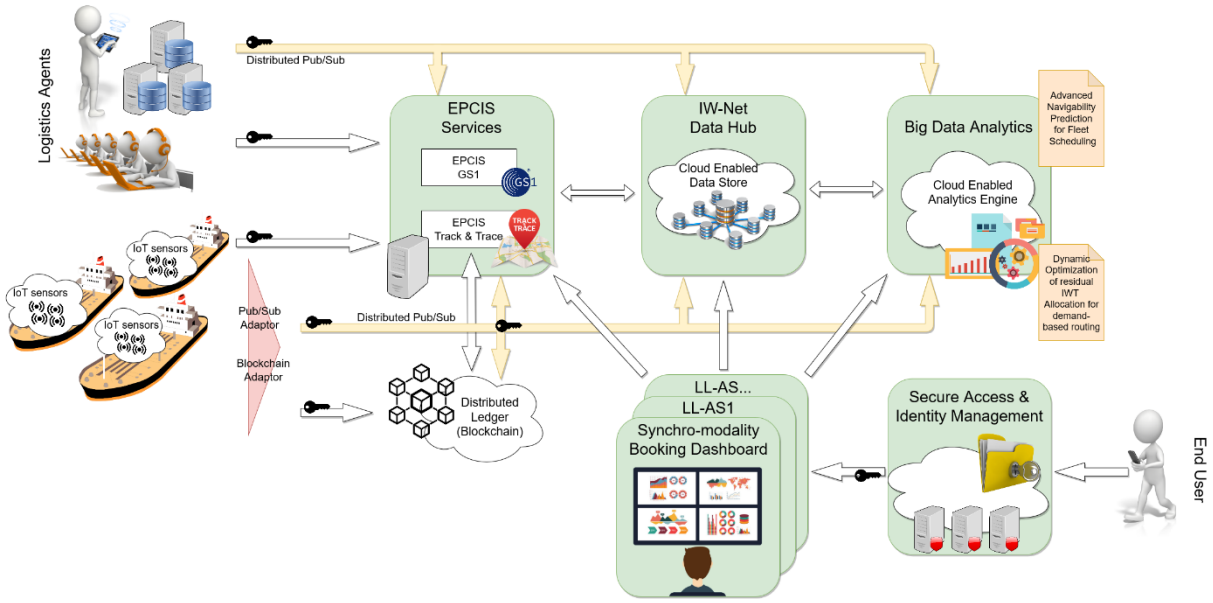

Figure 2-1: High-Level Architecture of the IW-NET ecosystem

2.1.1 The classical approach: Message Queues

The concept of message exchange is significant in modern distributed, event-driven software architectures. In such complex software designs, it is standard practice to decouple the functionality into small, independent units. Message Queues are the simplest way to implement a message

exchange service in a reliable, asynchronous manner. According to the Message Queue design, independent software components communicate by sending each other data in the form of messages rather than contacting each other directly. The concept of queueing means that messages are placed on queues in temporary storage, allowing components to function independently of each other, at different speeds and times, in different locations, and without needing a logical connection between them, thus implementing an asynchronous communication pattern. Queuing allows us to:

- Achieve cross-component communication (even across component which might each be running in different environments) without having to implement the communication.
- Select the order in which a software component processes messages.
- Balance loads on distributed systems by arranging for more than one instances of a client to service a queue when the rate at which messages arrive exceeds a threshold.
- Increase the availability of services and applications by supporting recovery routines to service the queues in cases of failures, as messages do not get lost.

Typically, in micro-service architectures, there are cross-dependencies, which entail that no single service can perform its functionalities without interacting with other components. Message Queuing is a perfect fit for the microservice architectural design by **allowing components to exchange data with each other without getting blocked by responses.** Usually, messages are lightweight and contain event-related data, requests, replies to error messages etc.

In traditional Message Queue implementations, although the queue can be accessed by multiple clients, each message is strictly processed only once, by a single consumer. Therefore, this messaging paradigm is often referred to as a one-to-one, or point-to-point, communication pattern. An example of message exchange system architecture implemented using the Message Queue paradigm can be seen in Figure 2-2.

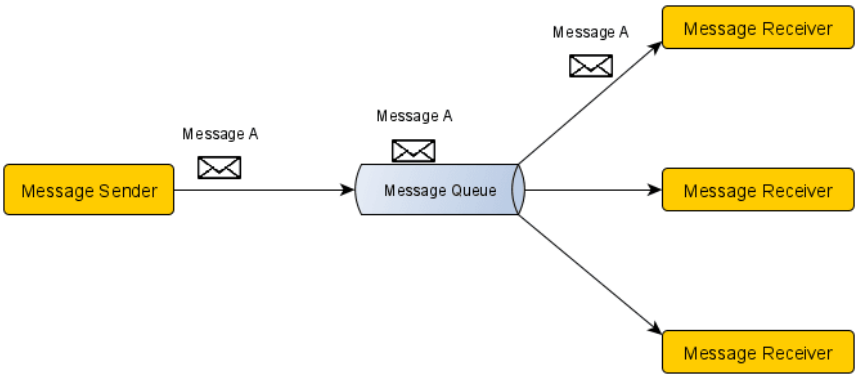

Figure 2-2: Message exchange using a simple point-to-point message queue [1].

2.1.2 Publish-Subscribe messaging

In contrast to the simpler Message Queue architecture, the Publish-Subscribe model allows messages to be broadcast to multiple receivers *asynchronously*. In the Pub/Sub paradigm, each **topic** essentially implements a concept analogous to a queue in the Message Queue model, providing:

- a lightweight mechanism to broadcast asynchronous event notifications, and
- endpoints that allow software components to connect to the topic in order to send and receive those messages.

To broadcast a message, a publisher pushes a message to a specific topic. All subscribers that are registered to the topic will receive every message published in it. In a Pub/Sub system, publishers do not need to know which subscriber consumes the messages they broadcast, and, similarly, subscribers

do not need to know which publisher is the origin of the messages they receive. Operating between the publishers and subscribers, a Pub/Sub **broker** is orchestrating the message exchange. The broker role consists of keeping a catalogue of registered topics and subscribers, receiving messages, temporarily storing them, and subsequently pushing them to the subscribed clients, therefore carrying all the complexity and intricacy of the message exchange protocol implementations.

This more flexible pattern of message exchange contrasts with the point-to-point approach we examined earlier, in which the applications that play the role of the publisher need to provide information on the expected receivers of the messages that are sent through the system.

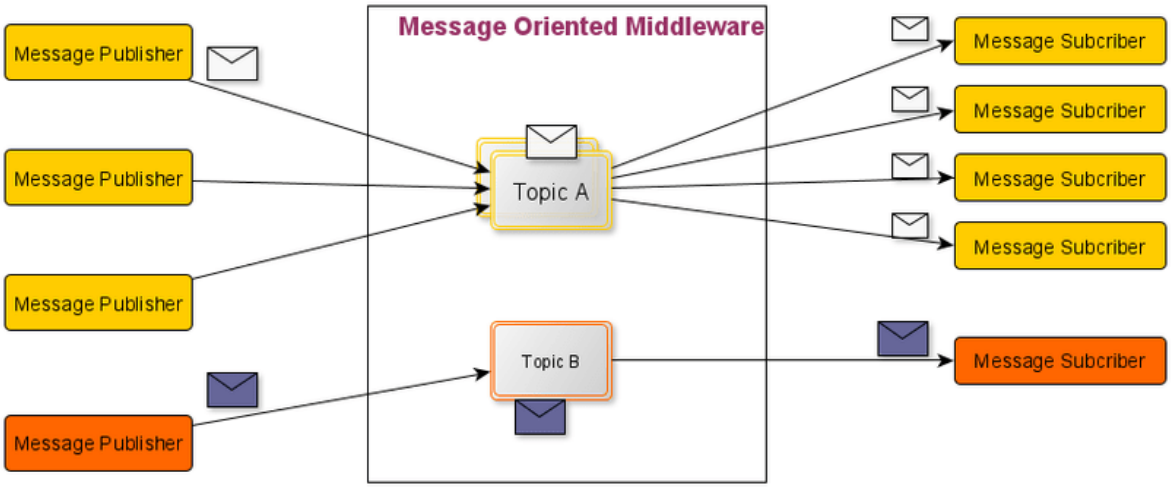

Figure 2-3: Message Exchange implemented according to the Publish-Subscribe paradigm [1].

Figure 2-3 displays an example of the architecture of a typical Publish-Subscribe message exchange system. One can notice that distinct topics essentially act as individual Message Queues – with the exception that multiple subscribers can receive the same messages simultaneously, if necessary.

2.2 The Publish-Subscribe Architecture in IW-NET

2.2.1 Moving on from the SELIS approach

In comparison to the solutions adopted in the scope of SELIS, we have chosen to move into a different direction in the context of the implementation of the Publish-Subscribe component. In SELIS we chose to utilise a research prototype developed by SELIS partners TU Dresden to implement the message exchange between components. Since the source code of that specific software solution is not part of the software that was open-sourced and made publicly available after the conclusion of the SELIS project, we were forced to reconsider our options regarding the implementation of the IW-NET Pub/Sub component. We selected to use Apache Kafka, a powerful streaming platform that excels in handling high-throughput, fault-tolerant, and scalable data streams. Its versatility and extensive ecosystem make it a popular choice for building real-time data pipelines and enabling streaming analytics in a wide range of applications and industries.

2.2.2 Introduction to Apache Kafka

In comparison to the solutions adopted in the scope of SELIS, we have chosen to move into a different direction in the context of the implementation of the Publish-Subscribe component. In SELIS we chose to utilise a research prototype developed by SELIS partners TU Dresden to implement the message exchange between components. Since the source code of that solution is not part of the software that was open-sourced and made publicly available after the SELIS project finished, we were forced to

abandon that solution. We selected to use Apache Kafka, a powerful streaming platform that excels in handling high-throughput, fault-tolerant, and scalable data streams. Its versatility and extensive ecosystem make it a popular choice for building real-time data pipelines and enabling streaming analytics in a wide range of applications and industries.

To accommodate the message exchange needs of IW-NET, we have deployed and appropriately configured a distributed Pub/Sub system implementation based on open-source, state-of-the-art software, **Apache Kafka** [2]. Apache Kafka is a distributed event streaming platform widely adopted by companies and research organizations around the globe. It is a powerful tool for the implementation of high-performance data pipelines, streaming analytics, and mission-critical applications. Kafka is based on the principles of the Publish/Subscribe model but further extends those, providing high throughput, scalability, permanent storage, and high availability.

We already described the basic theoretical principles of the Pub/Sub message exchange in the previous section. Kafka implements that architecture as a distributed, scalable system, therefore being able to serve arbitrarily high load by leveraging parallelism. More specifically, Kafka can scale, employing multiple message brokers that work in sync, forming a cluster. However, the maintenance and management of a cluster of message brokers introduces extra complexity. Problems such as synchronisation and consistency may appear, therefore appropriate mechanisms need to be in place. Kafka relies on Zookeeper [5] – i.e. a software designed to maintain configuration information, naming and to provide distributed synchronisation to distributed applications – to serve as its cluster manager. In the following sections we will discuss how our deployment of Kafka as part of the IW-NET architecture tackles such challenges.

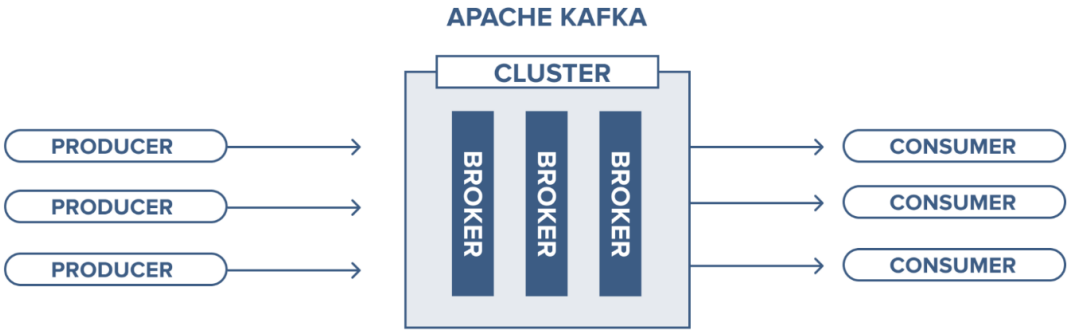

Figure 2-4: Apache Kafka architecture

2.2.2.1 Network and Request APIs

Kafka uses a binary protocol over TCP [3]. Interaction between the brokers and clients takes place through six core client request APIs, listed in Table 2.

In a typical scenario, the client initiates a socket connection, writes a sequence of request messages, and finally reads back the corresponding response message. In a distributed deployment, the client will likely need to maintain a connection to multiple brokers, as data is partitioned, and the clients will need to connect to the server where the data they want to read or write belong to. However, it should not generally be necessary to maintain multiple connections to a single broker from a single client instance (i.e., connection pooling).

The server guarantees that on a single TCP connection, requests will be processed in the order they are sent, and responses will return in that order as well. The broker's request processing allows only a single in-flight request per connection to guarantee this ordering. Despite that, clients can use non-

blocking IO to implement request pipelining and achieve higher throughput. i.e., clients can send requests even while awaiting responses for preceding requests since the outstanding requests will be buffered in the underlying OS socket buffer. All requests are initiated by the client, and usually result in a corresponding response message from the server.

Table 2: Kafka Client request API

Client request API	Description
METADATA	Describes the currently available brokers, their host and port information, and gives information about which broker hosts which partitions.
SEND	Send messages to a broker
FETCH	Fetch messages from a broker, one which fetches data, one which gets cluster metadata, and one which gets offset information about a topic.
OFFSETS	Get information about the available offsets for a given topic partition.
OFFSET COMMIT	Commit a set of offsets for a consumer group
OFFSET FETCH	Fetch a set of offsets for a consumer group

2.2.2.2 Topics & Partitions

Kafka follows the general Pub/Sub design where messages are organised into topics. Publisher clients write data to specific topics, while subscriber clients can only read messages published to the topics they have subscribed to.

Kafka topics can be divided into several partitions, which contain messages (records) in an immutable sequence. Each record is assigned to a partition and can further be identified by its unique offset. In a distributed Kafka broker deployment, having implemented multiple partitions allows topics to be parallelised by splitting their partitions across multiple brokers. Subscriber clients can read from topics with multiple partitions in parallel.

Additionally, Kafka uses data replication on a partition level, to prevent data loss in case of broker failure. Usually one or more replicas (copies) over as many brokers in the cluster are maintained for each partition. For every partition, one of the brokers acts as a leader while the rest are followers. The leader is tasked with serving all read-write requests for its specific partition, whereas followers maintain replicas of the leader's data. According to the protocol, if a leader broker fails, one of the followers replaces it as the new leader. Every time a record is published to a topic, the leader broker of the partition it belongs to handles the write request. The leader appends the record to the commit log of the appropriate topic's partition and increments its record offset. Kafka only exposes messages that have been committed and distributed to the followers.

The number of partitions into which each topic is split is predefined. Topic partitions themselves are nothing more than ordered "commit logs" numbered 0, 1, ..., P, while P is the number of partitions. The brokers do not enforce semantics of which messages should be published to a particular partition. On the contrary, it is the Kafka clients that directly control the assignment of messages to specific partitions. In order to publish messages, clients need to directly address them to a particular partition. In similar fashion, when fetching messages, those need to originate from a particular partition. If consistency between the partitioning scheme between multiple clients is desirable, they clients must use the same method to compute the mapping of message key to partition.

Requests from clients to publish or fetch data must be sent to the broker that is acting as the leader for a given partition. This condition is enforced by the broker, so a request for a particular partition to the wrong broker will result in a `NotLeaderForPartition` error code.

Obviously, because of the protocol policies discussed so far, when trying to read or write, clients need to know which broker leading of partition they should read to/read from. This happens through a process called “bootstrapping” in Kafka terminology. Clients need to issue a request for meta-data about the state of the cluster. Any active broker can be the recipient of such a request, as they all maintain and update a list of available topics, their respective partitions and the brokers that lead them. Since any broker can respond to this initial request, it is common practice for clients to cycle these meta-data requests through a known list of available brokers to balance their load more efficiently.

The client does not need to keep polling to see if the cluster has changed; it can fetch metadata once when it is instantiated cache that metadata until it receives an error indicating that the metadata is out of date. To summarize, the steps followed by a client interacting with the Kafka cluster are the following:

- Cycle through a list of Kafka brokers to complete the "bootstrapping" process, until a connection has successfully been established. Fetch cluster meta-data.
- Process read or write requests, directing them to the appropriate broker based on the topic/partitions they send to or fetch from – more on this topic in the following sub-section.
- In the case of an error that indicates cluster state has shifted, refresh the metadata and try again.

2.2.2.3 Partitioning Strategies

Having discussed the protocols that Kafka implements, we should next focus on the logic behind the topic partitioning. As highlighted earlier, it is the clients’ responsibility to enforce and maintain a consistent partitioning scheme across a specific topic. The criterion that dictates the partitioning policy depends on the type of application and data at hand. More specifically, partitioning serves two purposes in Kafka:

- Balance the data and request load over available brokers, and
- Distribute the processing load among consumer processes while allowing local state and preserving order within the partition.

A simple approach to accomplish simple load balancing could be for clients distribute requests in a round robin fashion over the list of all available brokers. Another alternative, in an environment where there are many more message publishers than brokers, would be to have each client choose a single partition at random and publish to that. The latter will additionally result in far fewer TCP connections.

In order to distribute processing loads, we would have to impose a partitioning scheme where records that would be processed as a group would be assigned to the same partition. We can achieve that using a key in the message to assign messages to partitions. By default, the hash of the key is utilised to calculate the appropriate partition but this behaviour can be overridden, if the user decides to do so.

2.2.2.4 Offsets: progress tracking and Consumer Groups

Kafka brokers temporarily retain messages for a configurable retention period to account for possible failures and provide the feature of stream replay-ability, the option to repeat the messages of a specific topic, if necessary. All message data and meta-data are retained in log files, stored in the brokers’ local file systems. Subscribers are responsible for tracking the position of the records they want to read in

the log, known as the "offset". Typically, a subscriber advances the offset in a linear manner, consuming and processing messages in the order they have been sent in. However, subscriber applications can consume messages in any order, as long as they are aware of the offset of the messages they need to fetch.

In order to serve high data throughput, subscribers – also called *consumers* in Kafka semantics – can be organised in consumer groups. Consumer groups consist of multiple consumer applications the total of which can only read every record of a particular topic exactly once. Kafka can support a large number of consumers and retain large amounts of data with very little overhead. By using consumer groups, consumers can be parallelised so that multiple consumers can read from multiple partitions on a topic, allowing a very high message processing throughput. The total number of a topic's partitions dictates the maximum parallelism of its consumers, as there cannot be more consumers than partitions.

Brokers distribute read requests across the various consumer applications of a group while also keeping track the offset for the consumer group for each partition. The latter is tracked by having all consumers commit the offsets of the records they have handled. Whenever a consumer is added or removed from a group, a rebalancing procedure takes place. That causes consumers to stop, so unstable clients which often suffer time-outs or are often restart will highly affect the throughput. Consumer applications across specific groups need to be stateless, since rebalancing the load might result in different partition assignments compared to the original.

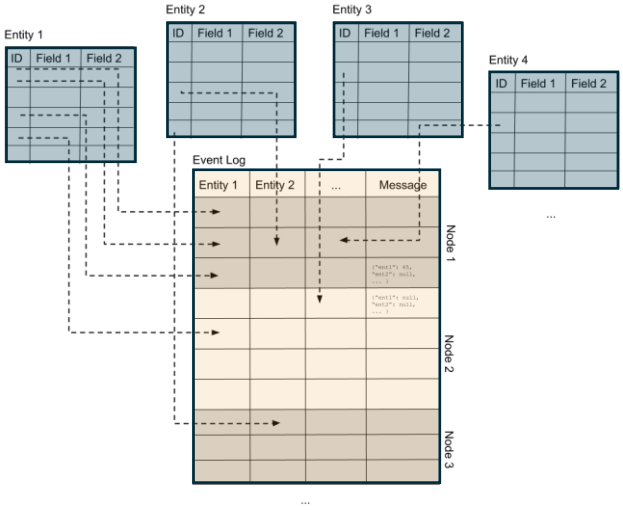

Figure 2-5: The star schema employed by the BDA.

2.2.3 Interaction with other components

In IW-NET, the Pub/Sub infrastructure plays the role of the main message exchange channel between the various system components. Components which serve as data sources publish event-related messages to designated Pub/Sub topics. Components which serve as data sinks subscribe to the topics they are interested in and receive all the messages published there. The architecture of the Pub/Sub mechanism is designed in a manner that allows Kafka to take advantage of its scaling capabilities, based on the volume of the incoming messages.

All interactions of other components with the Pub/Sub mechanism are authorised through the security component presented in chapter 3. This ensures that clients that try to establish a connection to the brokers to either submit or request data have the privileges to perform the respective actions. Unauthorised action requests are automatically blocked, as explained in detail in the next section.

2.2.3.1 Interaction with the Big Data Analytics platform

Although messages that pass through the Pub/Sub mechanism are retained for a configurable amount of time, the Pub/Sub is not the responsible component for persistent storage. Instead, the component responsible for storage is the Big Data Analytics platform (BDA), introduced in D1.5 and is illustrated in Figure 2-6. Following the specifications thoroughly discussed in that report, we design the IW-NET message exchange service to be compliant with the star schema that the BDA utilises, depicted in Figure 2-5.

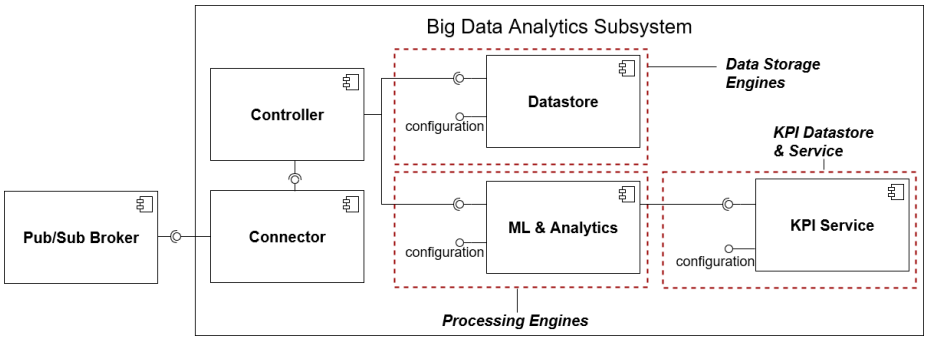

Figure 2-6: BDA low-level architecture

The **topics** created are mapped to the existing `message types`. Each message that is published to these topics passes through the brokers. It is then received by a subscriber group that resides within the Connector module of the BDA[1]. The Connector module is responsible for overseeing its smooth operation and has been tasked and configured to scale the allocated resources according to the volume of the incoming messages in order to ensure unobstructed operation of the BDA workflows. More specifically, when initialised, the Connector deploys a single instance of a pub/sub subscriber, running in a dedicated thread. The Connector module monitors the traffic and throughput of the subscriber that processes the incoming messages before they are stored and, if necessary, can deploy additional instances of the service. These additional instances also run in dedicated threads and belong to the same consumer group, therefore achieving load balancing while additionally ensuring that each message is only processed exactly once.

2.2.3.2 Interaction with EPCIS and other IW-NET components and services

Besides the BDA, which is a core component of the IW-NET architecture, we have created templates for the implementation of lightweight Publish-Subscribe clients which can play either the role of data producers or consumers in our application scenarios. The codebase of these templates has been made available to partners responsible for the development of services and components included in the IW-NET architecture as envisioned for the needs of WP1. Most notably, during the last months of the development tasks, the involved project partners have successfully achieved integration with the EPCIS services. EPCIS stands for Electronic Product Code Information Services. It is a standardized data format and interface for capturing and sharing supply chain event information. EPCIS provides a framework for tracking and tracing products as they move through the supply chain, enabling visibility and transparency into product movements, locations, and events. It is commonly used in industries such as retail, logistics, and healthcare to capture and exchange information related to the movement and status of products, including their origin, production, shipping, and receipt.

EPCIS provide a steady flow of data into the IW-NET ecosystem. However, as the work package tasks under development continue to mature, an even higher degree of integration can be expected, since the Publish-Subscribe system, protected by the Secure Access and Identity Management service described in Chapter 3 is the de facto channel of communication between distinct sub-systems and components.

2.3 System design and deployment

The system design and deployment plan for the Publish/Subscribe system of IW-NET is consistent with those of the core BDA and the Identity and Access Management (IAM) component. All three IW-NET

[1] The role of the Controller module is extensively discussed in sub-section 2.5.1 of Deliverable 1.5.

components' requirements and interactions have been considered since the first iterations of system design and deployment plan cycles.

As far as the Pub/Sub mechanism is concerned, it is worth highlighting that in the IW-NET implementation, its cluster of brokers is deployed as containerised applications leveraging Kubernetes[2]. Kubernetes, or K8s, is an open-source system for automating deployment, scaling, and management of containerised applications.

Containerisation is a deployment method that involves encapsulating an application and its dependencies into a self-contained unit called a container. Containers provide an isolated and portable environment for running software applications. They package everything needed for an application to run, including the code, runtime, system tools, libraries, and settings.

Containerisation offers several benefits as a deployment method. Firstly, it provides consistency and reproducibility across different environments, ensuring that the application behaves consistently regardless of the underlying infrastructure. Containers are also highly portable, allowing applications to be easily moved and deployed across various platforms and cloud providers.

Moreover, containerisation promotes scalability and efficient resource utilisation. Multiple containers can be executed on a single host system and can be dynamically scaled up or down based on demand. This flexibility enables efficient utilisation of computing resources and supports the development of scalable and resilient applications.

Additionally, containerisation enhances the isolation and security of applications. Containers provide a boundary between the application and the host system, preventing conflicts and dependencies. This isolation improves security by limiting the impact of potential vulnerabilities within the containerised application. The components of the IW-NET architecture that need to communicate with the Kafka cluster are given access to it and Kubernetes ensures that, if necessary, the cluster can scale out – expanding the number of its allocated resources by employing additional broker instances, in the form of containers. The configuration required has been automated to help the system suffer minimum service downtime while the transition takes place. Kafka shares a common Zookeeper cluster with the processing and storage services.

[2] https://kubernetes.io

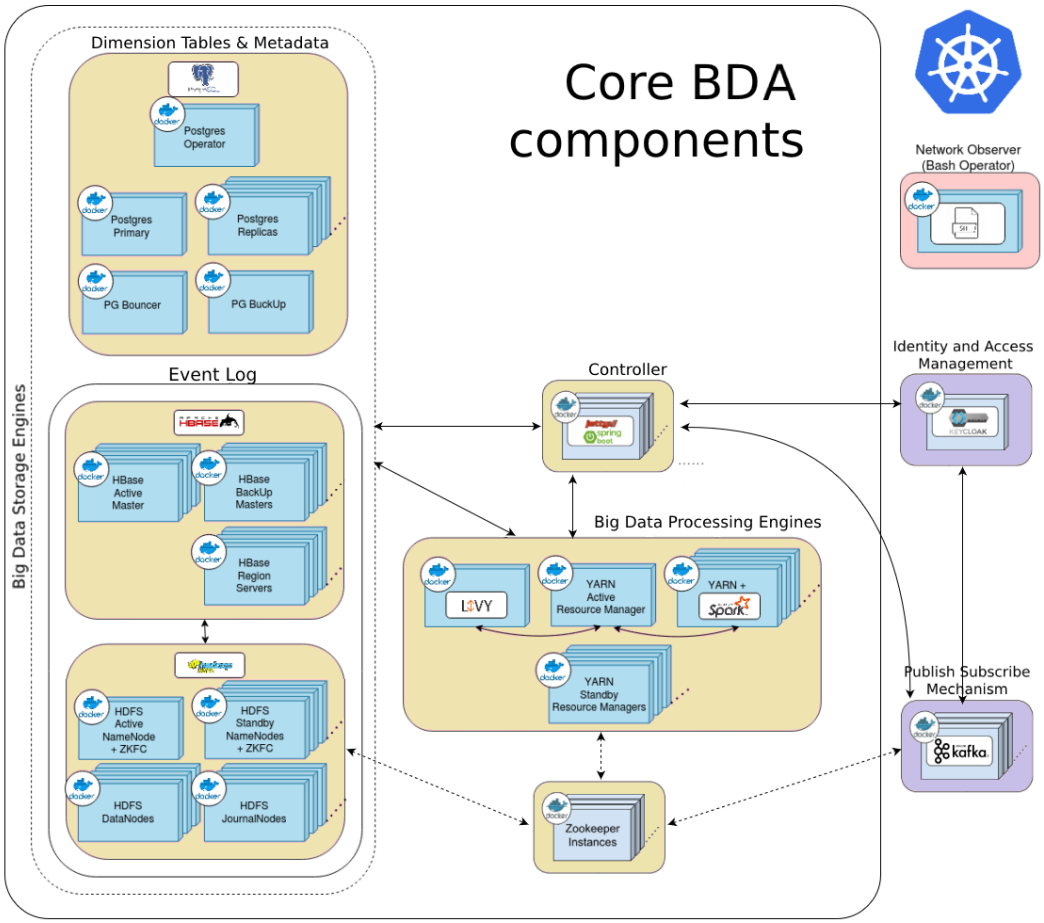

Figure 2-7: Deployment plan and interactions between the Pub/Sub, IAM and BDA components

More technical details on the system design and deployment plans available for the Pub/Sub as well as the BDA and IAM service infrastructures will be provided in D1.6, given that this analysis is more suitable to the technical nature of that deliverable. In Figure 2-7, we observe the complexity of the deployment as well as the interactions between the modules of the three co-designed IW-NET components.

Finally, a Kubernetes deployment-ready version of the software stack depicted Figure 2-6 is released in the form of open-source software for reusability purposes, in the following repository: https://github.com/iwnet/digitalization-infrastructure

2.4 Added Value for the IW-NET Architecture

The existence of a Publish/Subscribe mechanism is of critical importance in the design of the IW-NET technical infrastructure architecture. As documented in multiple deliverables and highlighted in this report, several components need to communicate with each other in a reliable manner. Thus, the task of message exchange is offloaded to the mechanism especially deployed for that purpose and the rest of the architectural components only need to implement thin clients that either send or receive messages to or from specific topics. This design ensures that the different functionalities are decoupled which is desirable for debugging, maintenance, and component/service-individual scaling reasons.

3 Secure Access and Identity Management

One of the most important tasks within the IW-NET project is the development of a Secure Services Federation layer to secure the exchange of information between the different systems that participate in the IW-NET ecosystem. This layer will be based on an open-source Identity and Access Management (IAM) Framework and on encryption protocols for enabling secure and private communications.

In general, there are two different communication channels that can be used for data exchange in IW-NET depending on the requirements. These channels are based on two different communication protocols: REST and Publish-Subscribe. The Secure Services layer will be used to secure both channels. The developed security solution will be used to support all IW-NET business cases and is currently actively used to secure the communications between the Big Data infrastructure and the EPCIS services, as well as data transfers from external data providers that push data in the Big Data storage layer. The integration of the Secure Services with the various IW-NET components is presented in detail in Figure 3-1.

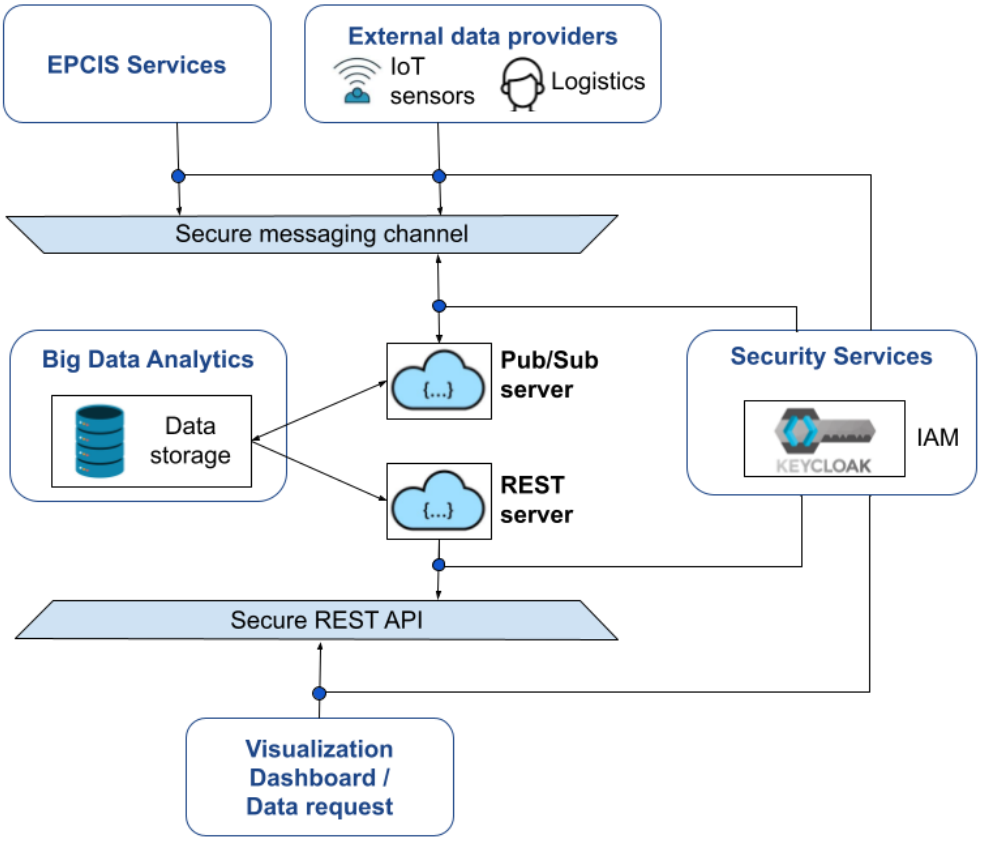

Figure 3-1: Integration of the Security Services with the IW-NET components

In the rest of this section, we will initially describe the basic principles and operation of the IAM framework that will be used with the Secure Services layer. The integration procedure of the Secure Services with both of the communication channels will also be described together with the detailed configuration that was used. Finally, we will present an example workflow for each channel of communication, which depicts the required actions for achieving an end-to-end secure data exchange.

3.1 IAM Framework

Identity and Access Management (IAM) Frameworks are designed to offer the technology required to ensure that the resources of a particular application are accessed only by authorised individuals/users. This is achieved with the setup of the appropriate policies which encapsulate information that

authenticates a user and also identifies the resources that a user is authorised to access as well as the actions that she is allowed to perform with them.

In this section we will present the architecture and configuration of Keycloak [4], which is the selected state-of-the-art IAM framework that will be used in order to secure the access to the resources of the IW-NET subsystems. An initial description of Keycloak and some high-level integration details with two of the most important IW-NET infrastructure components i.e., the Big Data Analytics subsystem and the Pub/Sub mechanism are already presented in D1.5. In this document we aim to provide a more detailed description of the initial configuration and the usage of the secured IW-NET components.

In this subsection, Keycloak is briefly presented along with its semantics and the utilities it offers which will be used in order to secure the IW-NET connectivity components. Keycloak is an open-source Identity and Access Management software which is released under Apache License 2.0.

Before we explain how the Keycloak software will be used in IW-NET, it is necessary to describe the basic entities that Keycloak encapsulates and on which it bases its operation. These entities are Users and Resources. The basic need that led to the concept of IAM software being developed is that we must be able to control whether a User is eligible to access some Resource. In order to decide whether the User is eligible or not, the need for a decision maker emerges. This decision maker is called a Resource Server since it handles access to various resources. A Resource server is considered to be a Client instance in the context of Keycloak. In general, a Client is the representative of each application that cooperates with Keycloak in order to secure itself by requesting Keycloak to authenticate a User. A Client however can also be an entity that just wants to request identity information or an access token so that it can securely invoke other services on the network that are secured by Keycloak.

During the operation of the Resource Server, the decisions about Users accessing resources are based on some defined rules, which are called Policies and Permissions. As described in D1.5, "*Permissions essentially relate Policies with Resources. More specifically, a Permission applies some Policies to a Resource. The two most common Policies that are used to define Permissions are Role and Group Policies. We will focus on the usage of Roles as they will be used for securing a REST API that serves the IW-NET data (in this case the Resources that need to be protected are the REST server URIs). A Role could be described as an abstract description for a User. Multiple Users could have the same Role and a single User could have many different Roles. A Role Policy contains one or more Roles that their users will be given access to some Resource(s) by creating the appropriate Permission(s). Thus, a User can access a Resource (i.e. URI) if a Permission exists that applies to this Resource a Policy containing their Role.*"

Another important entity inside Keycloak is the Scope. A Scope describes the access pattern for a protected Resource or a Client, for example if a Resource is only allowed to be read then a Scope for the reading capability should only be defined for this resource. Of course in case a particular User has the privilege to update or delete this Resource another Scope can be defined for this purpose and be assigned only to this User. *"Permissions can be used to relate Policies with Scopes to provide a User with access only to some Scopes. In another more generic case, a Client that should only read data from an application must be assigned the appropriate Client Scope. Client Scopes allow us to define sets of protocol mappers and roles for the Client. The application that the Client tries to access will cooperate with Keycloak and will only allow the Client to perform the operation if it has the required Client Scope. This approach is used to secure access to the Pub/Sub system that is used by IW-NET for message transportation, by creating for each client the appropriate Client Scopes for reading or writing each specific message type."*

Apart from the Resource Servers responsibility to decide if a User is eligible/authorised to access a resource, also known as authorization, there is also the need to initially identify valid Users, which is

the well-known authentication process. Keycloak supports as Authentication and Authorization protocols both SAML [5] and openid-connect [6]. We have used the openid-connect protocol for the integration with the IW-NET components, which is based on the OAuth authorization protocol [7]. The openid-connect protocol relies on the usage of tokens. There exist two types of tokens that support the authentication and authorization procedures: the identity token, which acts as a certificate for the user identity (authentication) and the access token, which indicates the Permissions of a User (authorization). The identity token essentially contains information about the User such as username, email, and other profile information. The access token is digitally signed and contains access information (like Scopes, Permissions on Resources, etc.) that the remote service can use to determine what Resources the User is allowed to access.

In the next subsection we will present in detail the configuration used with Keycloak for enabling the establishment of secure connections with both the REST and Pub/Sub endpoints that are used for data transfer. We will also present the corresponding secure access workflows for both connectivity components.

3.2 IAM Framework Configuration and Workflows

3.2.1 IW-NET IAM Configuration

In this section we will describe in detail the configuration that is applied to the Keycloak server which is used by the IW-NET components. There are two different connectivity components, as we have previously mentioned, the Pub/Sub server which is used for message transfer and the REST server provided with the BDA stack which is used for raw data transfer from within the BDA storage engines. Both components will rely on Keycloak for allowing authorised data access. Moreover, communication will be encrypted with the use of SSL protocols. Keycloak client adapters will be used in both components, which are prebuilt libraries that can be used to secure applications and services in cooperation with the Keycloak server.

3.2.1.1 Pub/Sub Clients

Starting from the Pub/Sub server which is a Kafka installation (as described in D1.5 and in section 2 of the present document), there are three different components that need to be secured individually for establishing secure Pub/Sub connections end-to-end: the Kafka broker, and any Kafka producer or consumer instance. For this purpose, the Kafka broker must be initially connected to the Keycloak server though a Client that will have the appropriate Client Scope that will authorize it to perform any administrative task. Client Scopes for Kafka will be defined using the Uniform Resource Name (URN) format: '`urn:kafka:{resourceType}:{resourceName}:{operation}`' where the '`resourceType`' field can be one of '`topic, group, cluster`' and the '`operation`' field can have any value from the following: '`read, write, create, delete, alter, describe, cluster_action`'.

Continuing with the setup of the secure broker, at first the Client Scope named '`urn:kafka:cluster:kafka-cluster:cluster_action`' is created in the Keycloak Admin User Interface as shown in Figure 3-2. Then a Client named '`kafka-broker`' is created and is assigned with the previously created Client Scope as shown in Figure 3-3 and Figure 3-4 respectively.

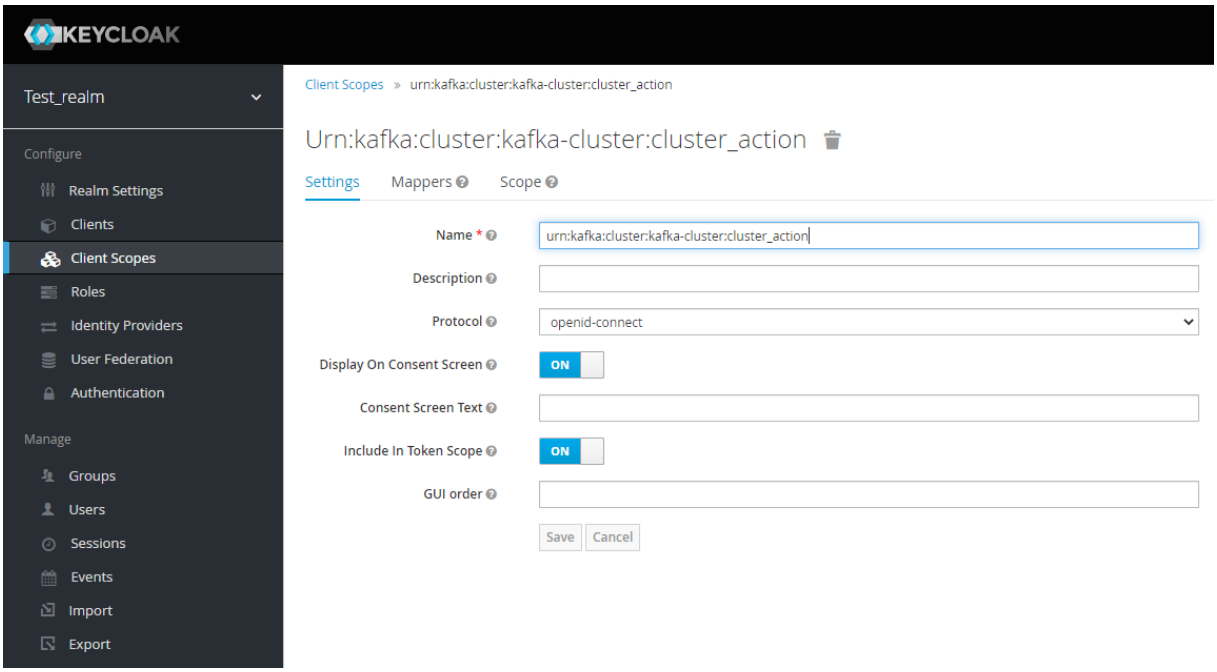

Figure 3-2: Create Client Scope for the Kafka broker

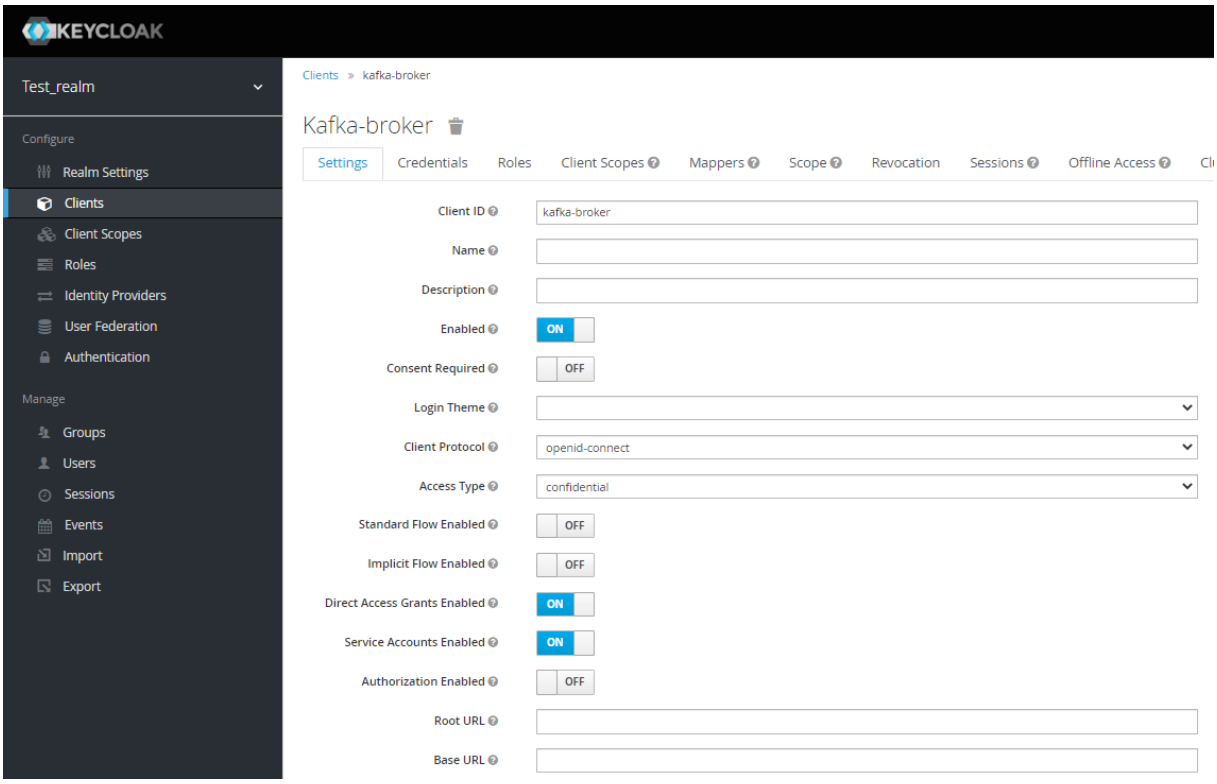

Figure 3-3: Create Client for the Kafka broker

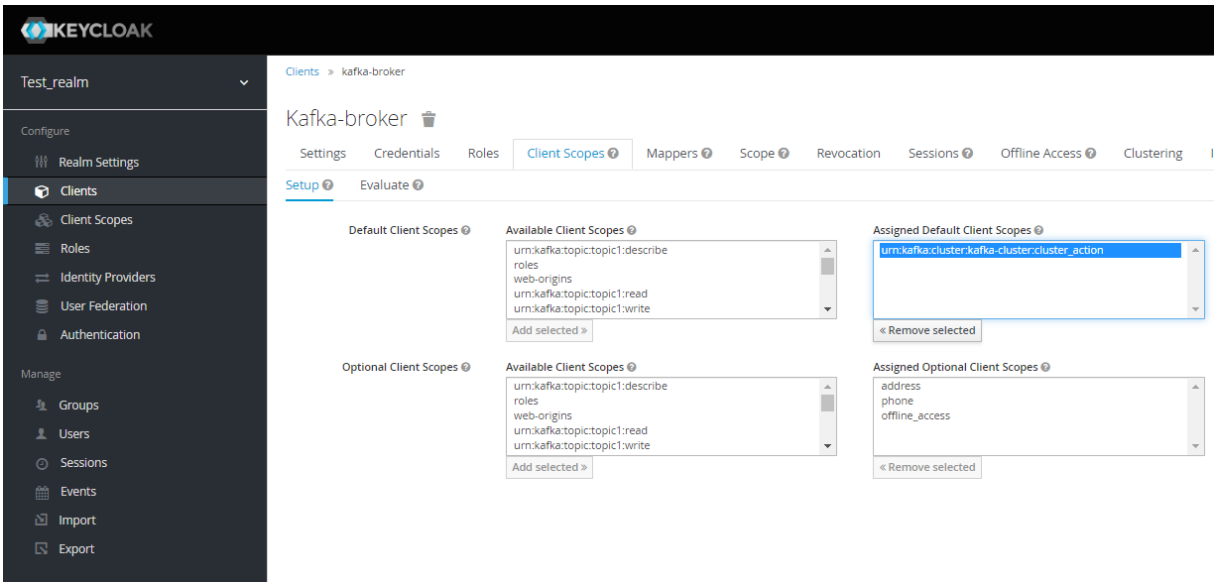

Figure 3-4: Associate client with client Scope for the Kafka broker

The final step to launch a secure Kafka broker is to retrieve the '`kafka-broker`' Client credentials from the corresponding UI tab and create a properties file consistent with the Kafka client templates developed. The Keycloak Client adapter which was used to connect the Kafka broker with the Keycloak server using the provided properties file is based on the **Kafka OAuth library** [8].

After creating the secure Kafka broker, let us assume a scenario in which we need to publish a message for the topic '`topic1`'. In this case a Client for a Kafka producer needs to be created that will have the appropriate Client Scope that will allow it to publish in this particular topic. We start again by creating in a similar way the necessary Client Scopes as shown in Figure 3-5, which are in this case '`urn:kafka:topic:topic1:describe`' and '`urn:kafka:topic:topic1:write`'. Then a Client named '`kafka-topic1-producer`' is created and assigned with these two Scopes as shown in Figure 3-6 and Figure 3-7 respectively.

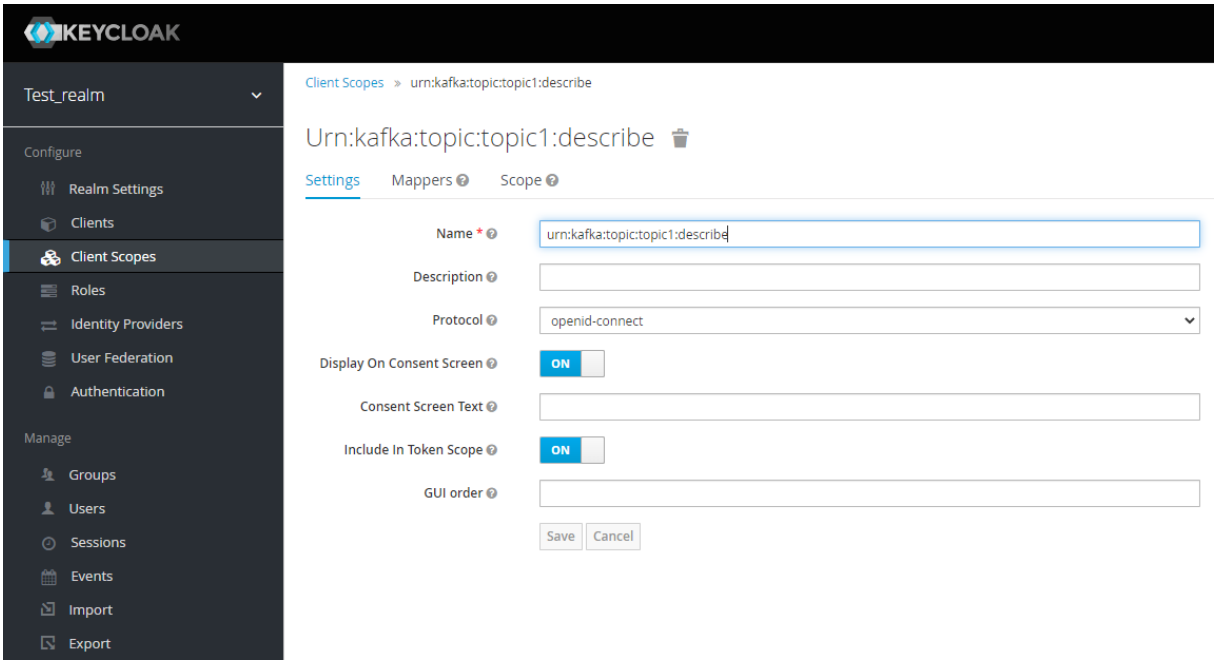

Figure 3-5: Create Client Scope for Kafka producer

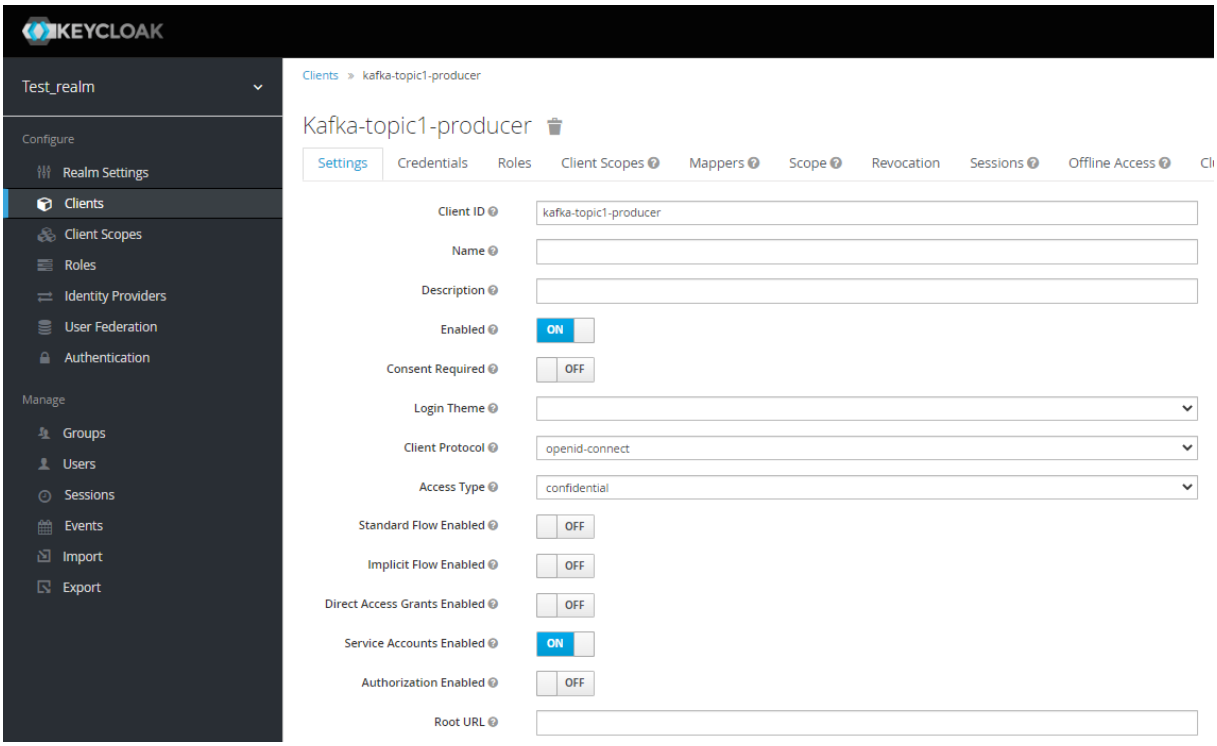

Figure 3-6: Create Client for Kafka producer

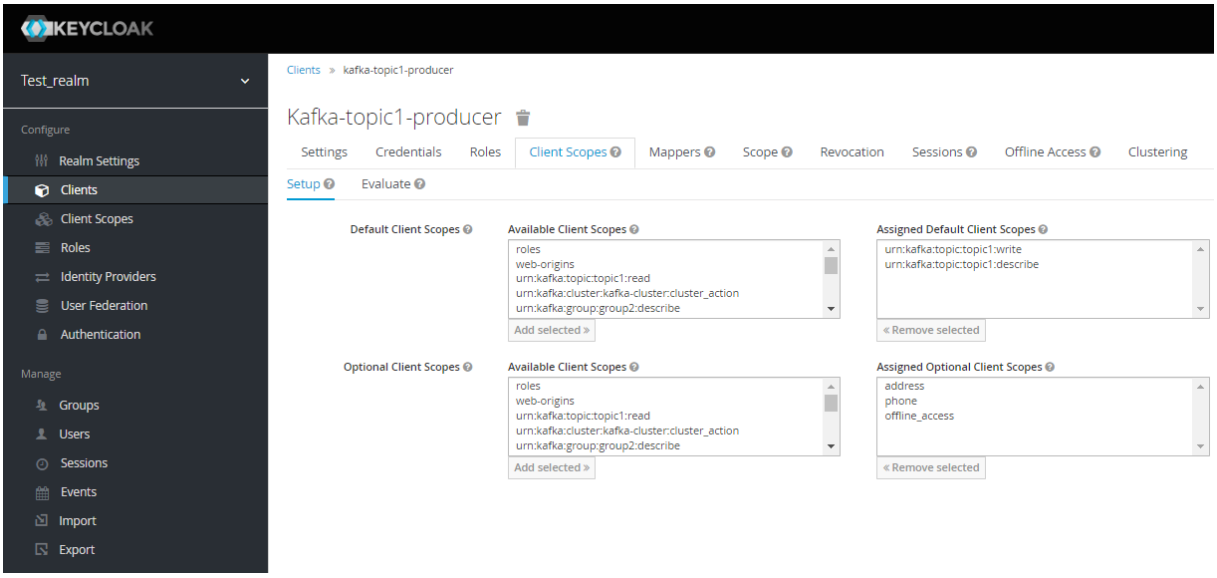

Figure 3-7: Associate client with the required Scopes for the Kafka producer

Finally, for reading messages from '`topic1`' with the corresponding secure Kafka consumer the procedure of creating the necessary Client is similar with the producer with two differences: first a Client Scope named '`urn:kafka:topic:topic1:read`' is assigned to the client instead of the '`urn:kafka:topic:topic1:write`' Scope, and secondly two more Client Scopes must be created and assigned to the Client that correspond to the Kafka group that consumes this topic ('`group1`') named '`urn:kafka:group:group1:read`' and '`urn:kafka:group:group1:describe`'.

3.2.1.2 REST API Client

For securing the data transfer that happens through the REST server of the BDA infrastructure of IW-NET, the procedure is different since we aim to protect different URIs in this case. However, similarly

with the Kafka broker, the REST server must initially be connected with the Keycloak server using a Keycloak client adapter. The client adapter used is the **Spring Security adapter** since the REST server is developed as a Spring-boot application. For the initial connection of the adapter with the Keycloak server we start by creating a Client named '`bda_client`' which is a Bearer Only Client meaning that it will only be used to evaluate User Access Tokens before serving Users with the requested URIs. The client is created as shown in Figure 3-8, and we can observe that we must define in the field 'Valid Redirect URIs' the base URL of the REST API.

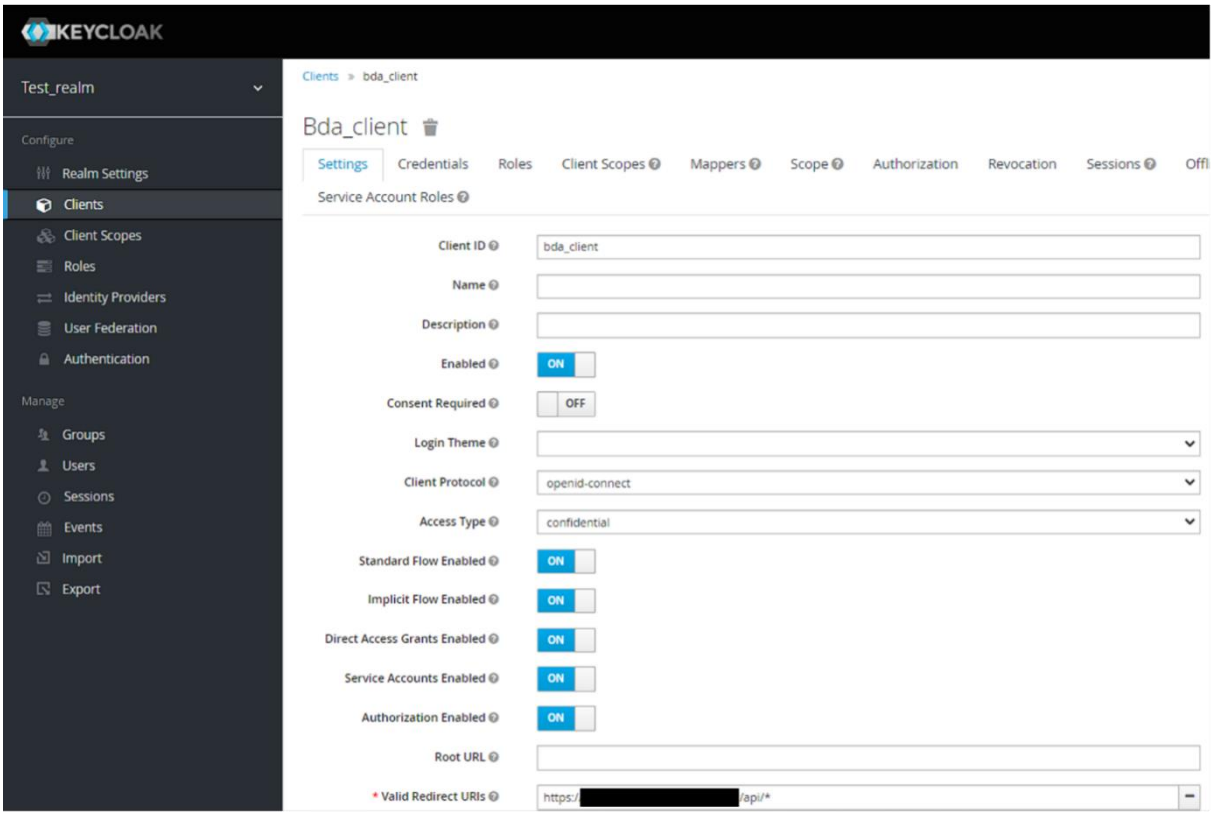

Figure 3-8: Create a Keycloak client for the REST server

In order to launch the secure REST server, we first retrieve the '`bda-client`' credentials from the corresponding UI tab and, knowing the Keycloak server URL, we update the REST server properties file as shown in Figure 3-9. The parameters of Figure 3-9 will be used by the Spring adapter to connect with the Keycloak server.

```
keycloak.enabled                = true
keycloak.auth-server-url        = https://[redacted]:8443/auth/
keycloak.realm                  = test_realm
keycloak.resource               = bda_client
keycloak.credentials.secret     = [redacted]
keycloak.use-resource-role-mappings = true
keycloak.bearer-only            = true
keycloak.cors                   = true
keycloak.policy-enforcer-config.enforcement-mode=enforcing
keycloak.policy-enforcer-config.lazy-load-paths=true
```

Figure 3-9: Update the properties of the REST server

After launching the secure REST server, we can start defining each available URI as a Resource to Keycloak and access to it shall be provided with a specific Role via the corresponding Policy and Permission. Each User that will be related to this Role can have access to this URI. In this way different Users can have access to different URIs of the BDA REST server according to their Role. Moreover, multiple URIs can be defined inside a single Resource for allowing access to all of them with a specific

Role. We will next present a detailed example to better illustrate the usage of Roles, Policies and Permissions in securing URIs.

Let us again assume a scenario where a set of URIs must be accessed only by a set of Users that are considered to be the '`scn_slug`' Users. We start be creating a Role that describes these Users and we name the Role '`scn_slug_user`' as shown in Figure 3-10. This Role is initially set to contain two Keycloak Users, the '`admin`' and '`some_user`' as shown in Figure 3-11, but more Users can also be added later.

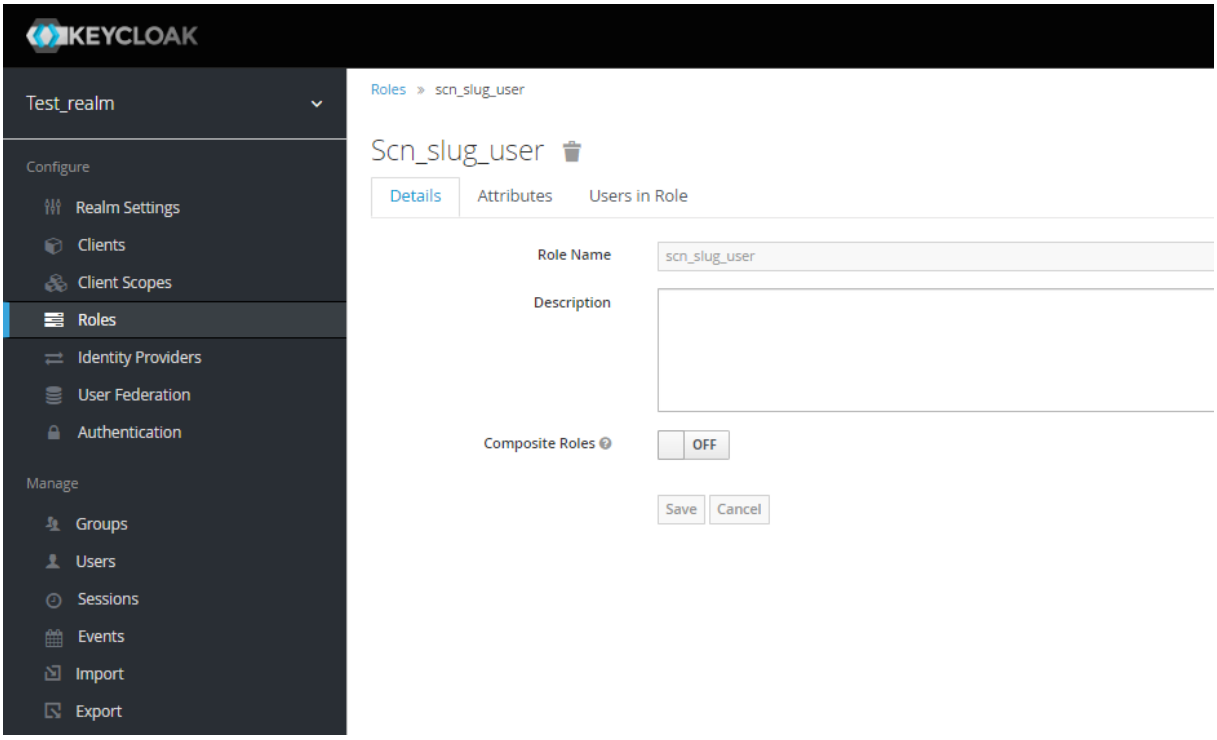

Figure 3-10: Create a Role describing the Users

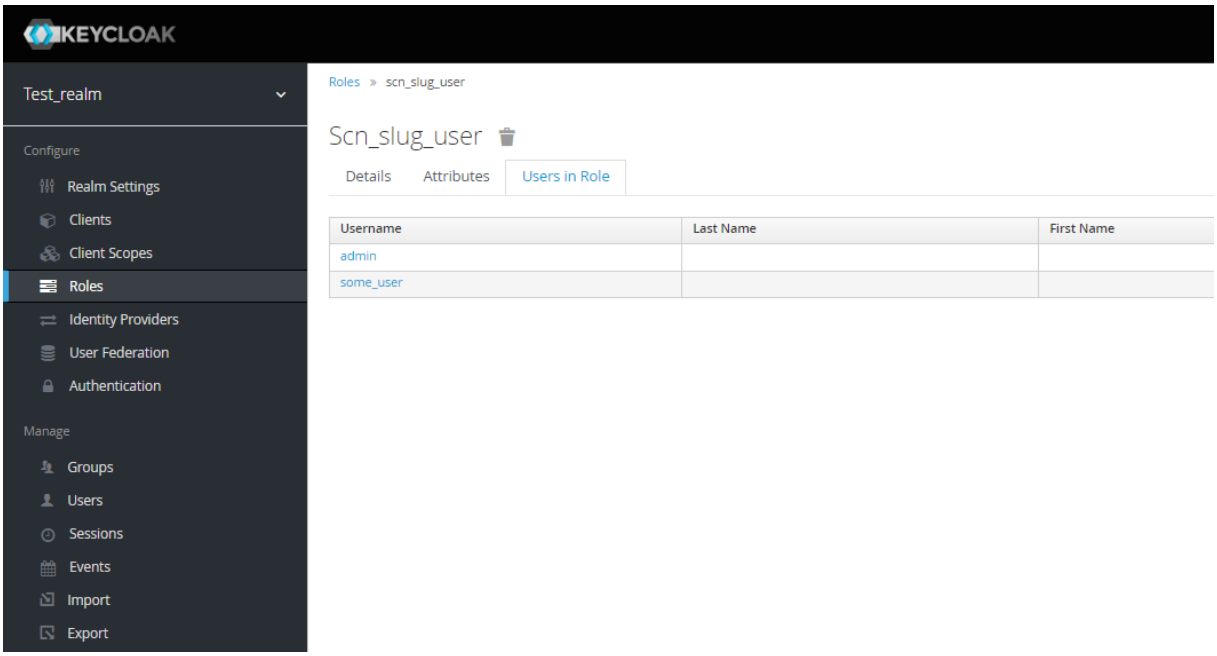

Figure 3-11: Define the Users in the Role

The next step is to define inside the '`bda-client`' a new Resource under the authorization options of the client. We place 6 different URIs inside this Resource which is named '`slug resource`' as we can see in Figure 3-12.

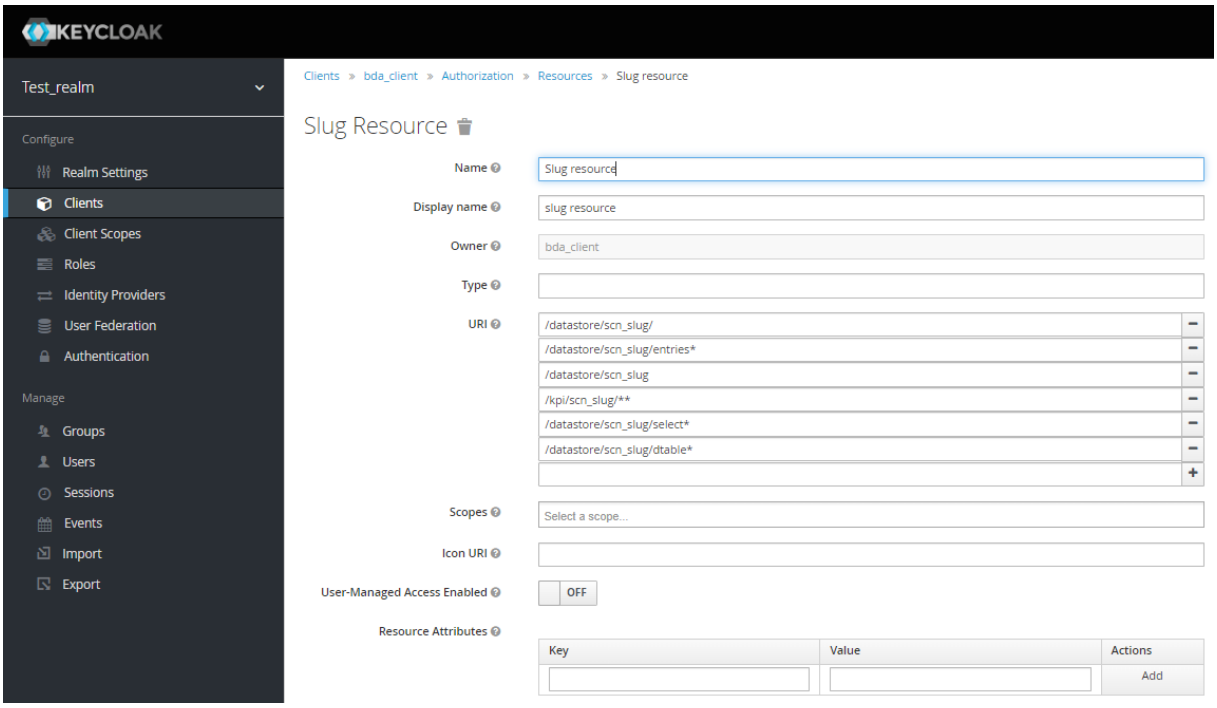

Figure 3-12: Create a Resource that contains some URIs

Following, again in the authorization options of the client, we select to create a **Role** policy named '`slug policy`' that contains the '`scn_slug_user`' role that we previously created (Figure 3-13). The final step to grant access on the URIs of the 'slug resource' to the Users of the '`scn_slug_user`' role is to create a **Resource-based** Permission inside the client that will associate the selected Resource with the corresponding policy containing this role as shown in Figure 3-14.

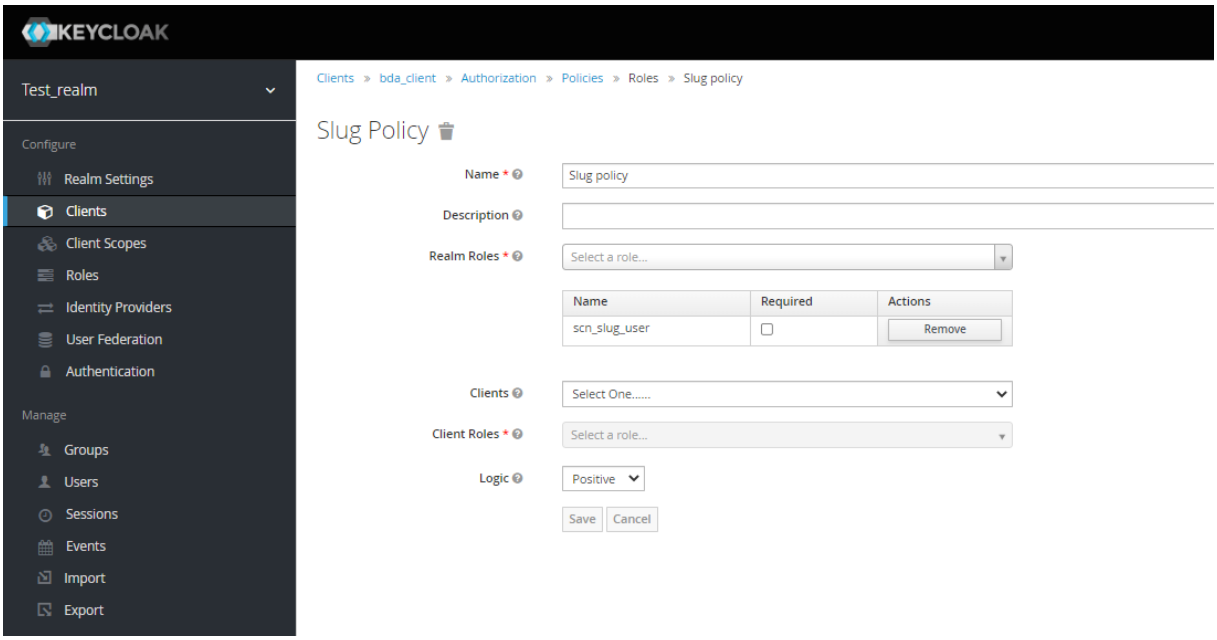

Figure 3-13: Create a Role Policy for the previously defined Role

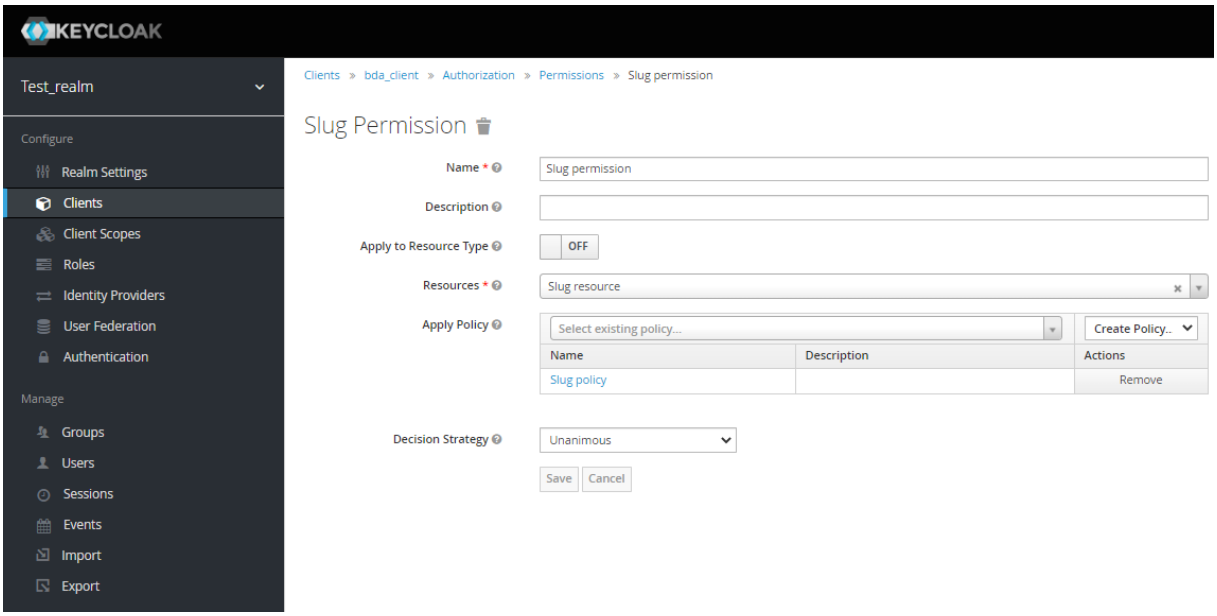

Figure 3-14: Create the Permission that will associate the Resource with the Policy

After completing the previous steps, both Users '`admin`' and '`some_user`' will be authorised to get a valid response from the BDA REST server for any of the URIs included in the resource.

3.2.2 Secure Access Workflows

In this section we will present the authentication and authorization workflows for both connectivity components of IW-NET.

3.2.2.1 Pub/Sub Workflows

Based on the two basic actions that one can perform with the Pub/Sub system which are to publish and receive a message, we will analyse in this subsection two separate workflows: one focusing on the producer's side and one on the consumer's. The workflows are very similar in both cases.

Initially we assume that a User wants to publish a message for the topic '`topic1`' using a Kafka producer. A Client must exist for the producer ('`producer-client`') with the corresponding credentials so that the producer can connect with the Keycloak server and request an access token for accessing the broker. After the Keycloak server receives the producer client credentials, it authenticates the client, and the client then receives the access token. If the client credentials are invalid the client instead receives an Invalid Credentials message.

The producer can now request to publish a message for a particular topic in the broker using this access token. The broker server is already running and is secured from the start, by connecting with the Keycloak server using the '`broker-client`' credentials. When the broker receives the request from the producer along with the access token, the broker's client contacts Keycloak to introspect this Token. In particular, it verifies the signature of the token in cooperation with the Keycloak server, then decides based on access information within the token whether or not to process the request. In this case the '`broker-client`' is trying to determine if the client has the required Scope for this action. Considering such information contained in the access token, the broker either accepts the action (`200 OK Response`) or otherwise with a `401 Unauthorized message`. In case the producer client is not even correctly authenticated (for example with an invalid/expired token) the broker server responds with a `403 Forbidden message`.

The complete workflow that we described in the previous paragraphs is graphically presented in Figure 3-15.

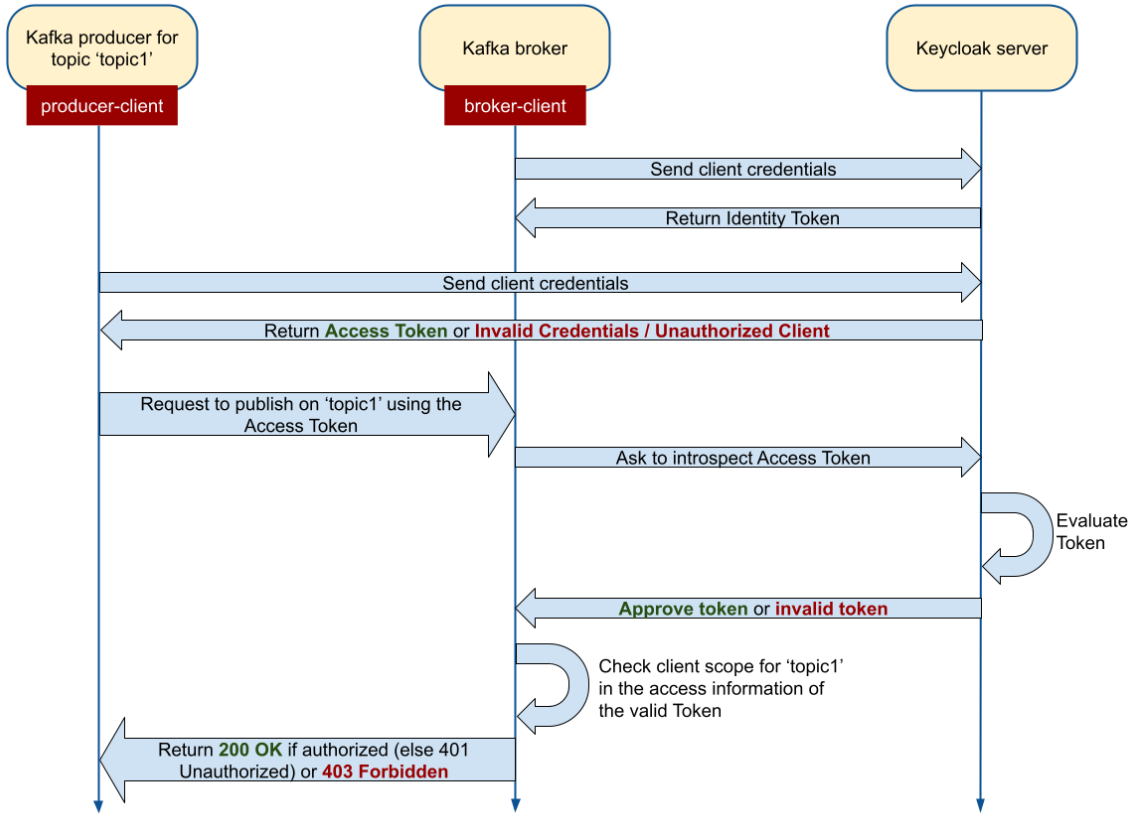

Figure 3-15: Publish message workflow

The workflow for a Kafka consumer that wants to subscribe to a particular topic to be able to receive relevant messages is almost identical and is presented in Figure 3-16. We will omit the workflow description in this case since the procedure is the same as with the producers.

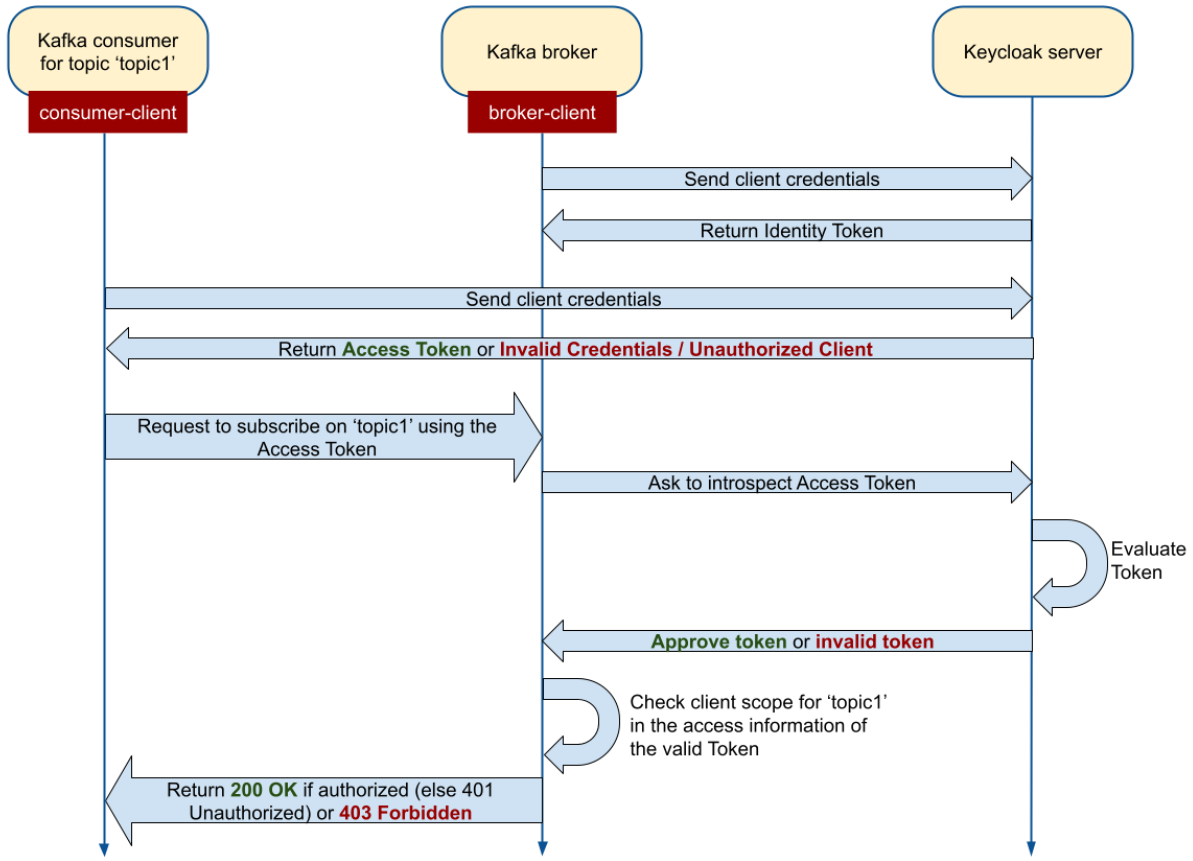

Figure 3-16: Subscribe to message workflow

3.2.2.2 REST API Workflows

In this subsection we will describe thoroughly all the actions that are involved in the simple case that a User wants to access a specific Protected Resource of the REST server i.e., a particular URI. Initially we assume that a user named '`User 1`' is defined as a Keycloak User and owns some credentials and a Client also exists ('`client-1`') with the corresponding credentials so that the User can connect with the Keycloak server and request an access token for 'User 1' to use it for accessing some other service. After the Keycloak server receives the User and client credentials, it authenticates the user, and the client then receives the access token. If the User credentials are invalid the client instead receives an Invalid Credentials message.

'`User 1`' can now make REST invocations on the BDA REST server using this access token. The REST server is already running and is secured from the start, by connecting with the Keycloak server using the '`bda-client`' credentials, which is a bearer-only client as we mentioned in subsection 3.2.1.2. The User '`User 1`' decides to request access to a specific URI of the REST server providing her access token along. At this step, the REST server's client contacts Keycloak to introspect this token. In particular, once the REST service extracts the access token from the request, it verifies the signature of the token in cooperation with the Keycloak server, then decides based on access information within the token whether or not to process the request. In this case the '`bda-client`' is trying to determine if the User has the Permission to access the Protected Resource that includes this particular URI. Taking into account such information contained in the access token, the REST server either responds with the requested Resource ('`200 OK Response`') if the token information indicates that '`User 1`' is authorised for accessing it or otherwise with a '`401 Unauthorized message`'. In case the User is not even correctly authenticated (for example with an invalid/expired token) the REST server responds with a '`403 Forbidden message`'.

The complete workflow that we described in the previous paragraphs is graphically presented in Figure 3-17.

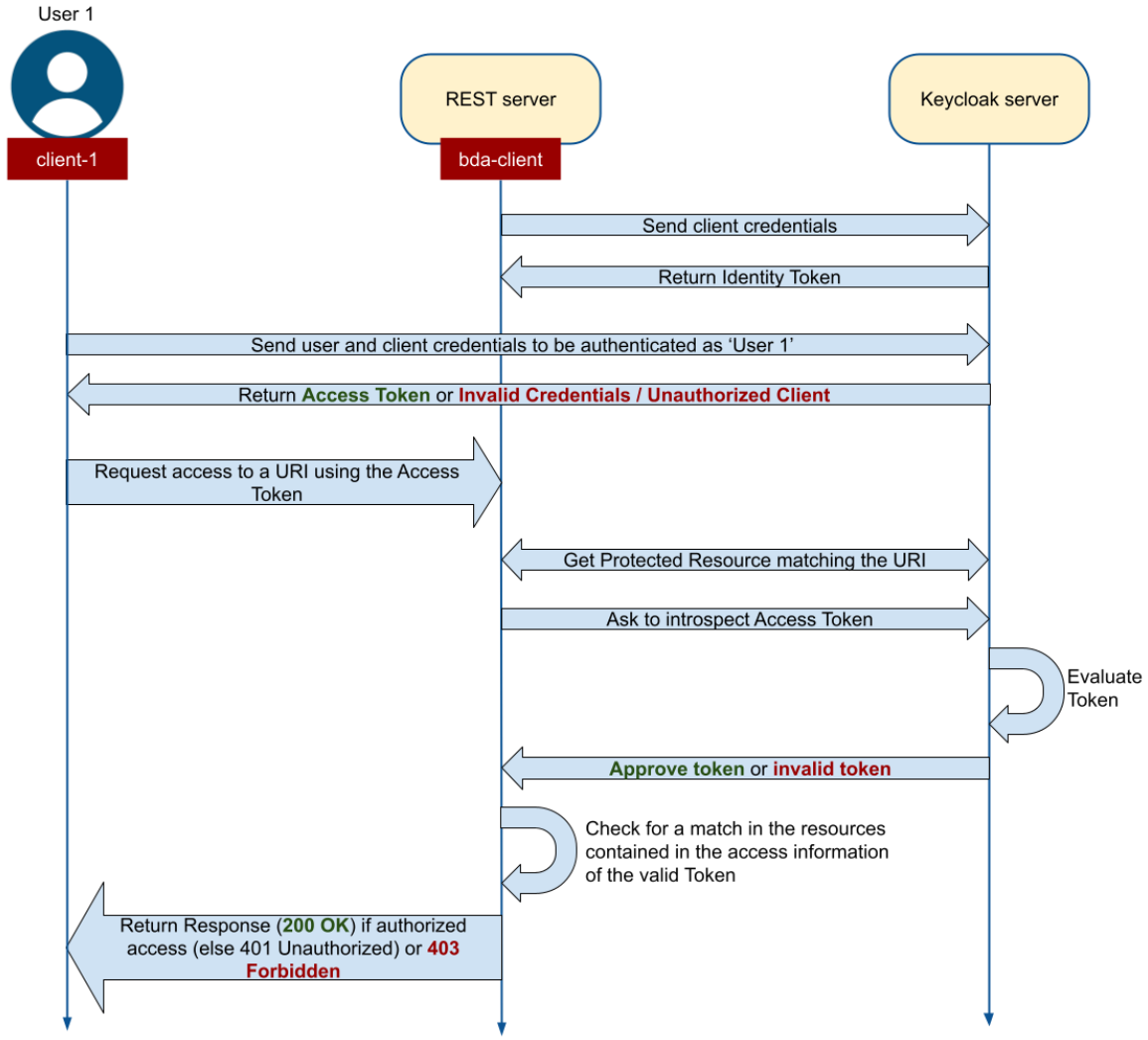

Figure 3-17: Workflow for REST API secure access

3.3 Deployment

As already discussed in the relevant sub-section on the Publish/Subscribe mechanism deployment and system design, the IAM solution has been designed alongside the other main components that support and enable Big-Data magnitude of data volume to be processed. The deployment has also been designed in association with that of the BDA and Pub/Sub components. A deployment plan can be seen in Figure 2-7. More technical details on the deployment plans available for the Pub/Sub as well as the BDA and IAM service infrastructures will be provided in D1.6, given that this analysis is more suitable to the technical nature of that deliverable.

A deployment-ready version of the software stack pictured in Figure 2-7 will be publicly available as open-source software to serve reusability purposes. The repository can be found in Github: https://github.com/iwnet/digitalization-infrastructure

3.4 Added Value for the IW-NET Architecture

The benefits of adding an IAM component to the IW-NET infrastructural architecture can be summarised as follows. The presented solution enables the management of data and Resource access

patterns, therefore dictating workflows and dataflows between components of the IW-NET architecture – regardless of whether these take place via the Pub/Sub mechanism or in a bilateral one-to-one communication pattern. Taking into consideration how sensitive some of the operational data can be, the need for secure access and identity management in the applications is paramount.

4 IoT Data Streaming Integration with Blockchain

The Open IWT Platform enables flexible configurations of Inland Water Transport (IWT) operations and trusted sharing of information between the stakeholders through blockchain. It uses the blockchain technology to improve processes and transactions along the whole transport chain as well as to increase the visibility of shipments and the provenance of messages. IoT data is a key-enabler for the trustworthy monitoring of cargo in the IWT domain that serve as an extra step of logistics events verification to increase the accountability of the stakeholders' claims.

4.1 Blockchain and IoT

Blockchain is a digital notebook that is shared among many computers, where important information is stored to make sure it's secure and undisputable. In the convergence of blockchain and IoT, the digital notebook keeps track of what all the connected IoT objects are doing and makes sure everything is working correctly and safely [9], [10], [11]. It resembles a super smart notebook that can help all the things in a house, factory or even the supply chain work together better. More specifically, blockchain and IoT intersect in several ways:

- Smart Contracts: Smart contracts can be used to automate processes and facilitate secure communication between IoT devices. For example, a smart contract could be used to automatically trigger a payment when a certain condition is met, such as a sensor detecting a low temperature in a refrigerator.
- Decentralised Identity: Blockchain can be used to provide decentralised identity to IoT devices, ensuring that only authorised devices can access certain networks or resources.
- Supply Chain Management: Blockchain can be used to track the movement of goods through a supply chain, providing transparency and reducing the risk of fraud. This can be particularly useful in the case of IoT-enabled devices, such as shipping containers fitted with sensors.
- Data Management: Blockchain can be used to securely store and share data collected by IoT devices, providing transparency, and reducing the risk of data breaches.
- Secure Communication: Blockchain can be used to establish secure and private communication channels between IoT devices, allowing them to share data and interact with one another without the need for a centralised intermediary.

Overall, the integration of blockchain with IoT has the potential to improve the security, efficiency, and trustworthiness of IoT-enabled systems and devices. In IW-Net, IoT enables live monitoring of logistics assets to enhance the traceability of cargo while the integration of IoT and Blockchain enables the employment of smart contracts to automate processes and improve accountability of actions and events.

4.2 The IW-Net Blockchain connector

The IW-Net blockchain connector is the component of the architecture that integrates the IoT infrastructure with the blockchain. As shown in Figure 2-1, IoT measurements are ingested in the IW-Net platform through a Pub/Sub mechanism (Apache Kafka) and once the data reach the Pub/Sub service, the blockchain connector forwards them to the smart contracts through the dedicated REST API endpoints of the blockchain component.

The design of the IW-Net architecture, API specifications and integration aspects are described in detail in the rest of the WP1 deliverables. Moreover, the architecture of the blockchain component is described in D1.7, presenting the smart contracts, the APIs and the blockchain infrastructure that was deployed to support non-repudiation of transactions, events and notifications data in the IW-Net application scenarios.

This document focuses on the design and development of the blockchain connector, which is an integrator between the IoT data and the blockchain component. It handles the ingestion of IoT data from the distributed Pub/Sub mechanism to the blockchain by registering to specific topics of the Kafka service. Consequently, only pointers to the IoT data are stored on the ledger, forming a timestamped and immutable chain of transactions, through the IoT API of the blockchain component.

The following use-cases are studied, where the blockchain connector is employed towards guaranteeing the authenticity of the data:

- late delivery of goods,
- damaged goods,
- suspicion of a break in the cold chain.

In particular, the IoT sensors are continuously feeding the platform with timestamped data about the location of the goods and about environmental metrics during the transport i.e., temperature, humidity, turbulence. These metrics should be trustfully validated through smart contracts. They may automatically trigger other actions involving multiple stakeholders such as the alarm notification in case of a violation of a contract term. Moreover, the traceability of information flows by all the involved actors in the supply chain should be guaranteed to enable non-repudiation and automatic settlement of disputes.

In terms of privacy, as little data as possible must be stored on chain (hashes of large documents/files are stored off-chain) to ensure the highest level of trust and privacy between supply chain stakeholders.

4.3 Design and Architecture

As shown in Figure 4-1 the Blockchain Connector takes advantage of the POST */event* endpoint of the Blockchain Component API to push pointers and metadata of batched IoT data to the blockchain, such as a list of temperature recordings (for cold chain control), photographs, signatures and/or comments by the stakeholders (proof of delivery, damaged goods). The data for off-chain storage are pushed back to the distributed Pub-Sub and consequently stored in the IW-Net Data Hub.

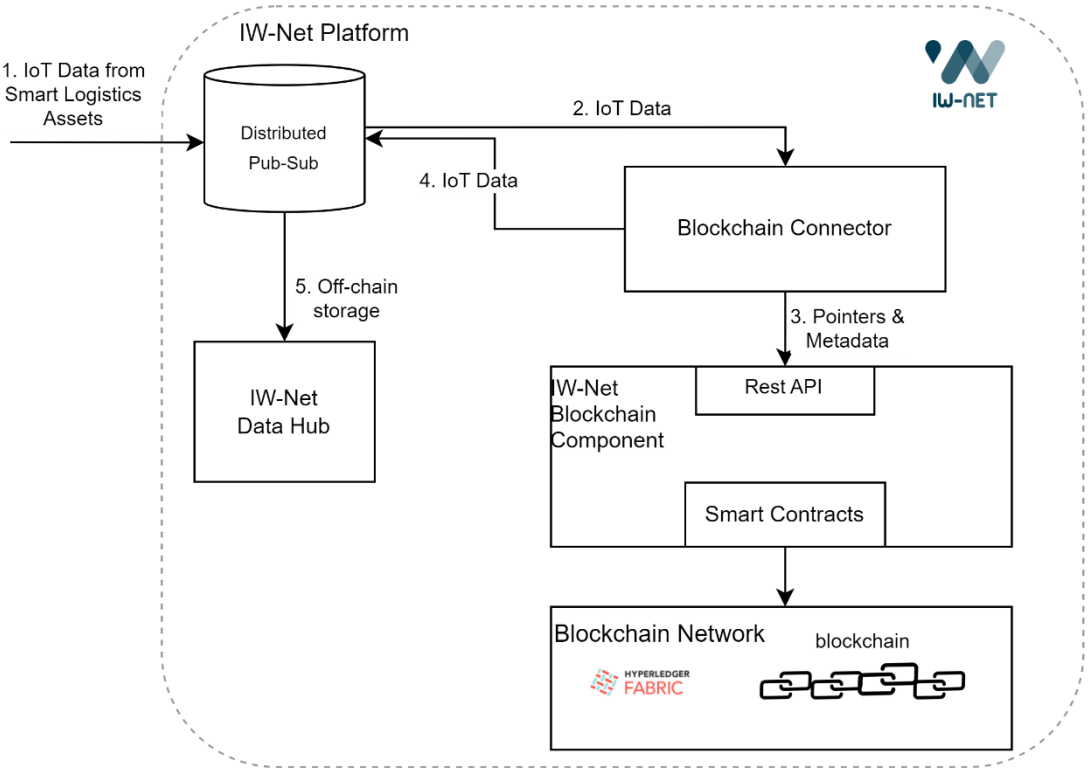

Figure 4-1. Blockchain Connector Component

4.4 Implementation

The Blockchain Connector takes advantage of the heterogeneous datasets within the IW-Net Platform to trigger smart contracts and inform decisions by storing their metadata on chain. To this end, it acts both as a producer and consumer of the distributed Pub/Sub, namely the Kafka deployment in the IW-Net Platform.

As per the consumer role, the Connector subscribes to a Kafka topic and reads IoT data from sensors by employing advanced techniques for the efficient management and filtering of the IoT messages according to their type, such as environmental measurements, arriving and departing events etc. After the validation of each message to be in the correct JSON format, the module searches the messages in multiple parallel threads and minimises unnecessary access to the Kafka topic for increased performance. The producer's module of the Blockchain Connector pushes two types of messages to the topic, either trusted events produced by the smart contracts such as alarms for contract violation or batches of IoT data to be stored off-chain in the IW-Net Data Hub.

All the interactions happen through the REST API of the Blockchain Component that is extensively documented in D1.7 and it is briefly described below for the completeness of the document. The endpoints handle the forwarding of data to the underlying smart contracts of the blockchain network and are grouped into three categories according to the three phases of the project application scenarios, namely:

- *planning* for the operations related to Transport Instructions (TIs) and Transport Instruction Responses (TIRs)
- *execution* for all the operations related to Transport Status Notifications (TSNs) and Events
- *analysis* for exposing functionalities to analyse the data

Table 3 lists all the endpoints of the REST API of the IW-Net Blockchain Component.

Table 3: Blockchain REST Server Endpoints

API Endpoint	Group	Description
POST /ti	planning	Adds a new TI record
POST /tir	planning	Adds a new TIR record
POST /event	execution	Adds a new event record
POST /tsn	execution	Adds a new TSN record
GET /analysis/bysscc	analysis	Returns all SSCC linked records in the ledger
GET /analysis/bygsin	analysis	Adds a new Transport Instruction record

The */event* POST endpoint is the one employed by the Blockchain Connector to send data to be stored in the blockchain. The full description of the endpoints, the responses and the data schemas of the REST API are developed using the OpenAPI specification[3].

Finally, the Blockchain Connector is configurable through the general configuration file of the IW-Net Blockchain Component. The Distributed Pub-Sub to be connected to (the Kafka Broker URL), together with the topic used by the Blockchain Connector, are exposed in this file and the communication between the Connector and the Kafka service is protected using openid-connect authentication, which is a token-based authentication mechanism to allow secure client requests, as described in D1.5.

4.5 Added Value for the IW-Net Architecture

The Blockchain Connector together with the entire Blockchain Component are beneficial to the IWT domain by providing increased transparency and accountability to supply chain operations, security of transactions, efficiency in operations and traceability in the entire supply chain. More specifically, it provides:

- Transparency throughout the supply chain, allowing all parties involved to see the movement of goods and the status of shipments in real-time. This can improve collaboration and communication among supply chain partners and reduce the risk of fraud.
- Security to supply chain operations by encrypting sensitive IoT data and providing tamper-proof records. This can help prevent data breaches and protect against cyber-attacks.
- Efficiency in supply chain operations by automating processes and reducing the need for intermediaries through smart contracts. For example, smart contracts are used to automatically trigger payments or release goods when certain conditions are met. This can help to speed up the movement of goods and reduce costs.
- Traceability by providing an immutable record of the entire journey of a product. This can help identify any issues as well as improve accountability.

[3] OpenAPI specification v3.0, https://swagger.io/specification/

5 Conclusion

This document presents the work undertaken in the context of Task 1.2 *Open security preserving data and services connectivity components - federation of IWT systems*. The purpose of this task has been the development of a secure services federation layer that integrates and extends open-source components to allow secure connectivity with:

a) external resources, such as Inland Port Management and Transportation Management systems,
b) internal resources, i.e., components of the IW-NET infrastructure.

The system design, implementation and deployment methods follow state-of-the-art standards set by industry-leading practices, which are highlighted in this report. A working, deployment-ready version of the solution including the Publish/Subscribe mechanism and Secure Access and Identity Management system are publicly available as open-source software in a dedicated Github repository.

The integration of IoT with Blockchain technologies enables the secure and trustworthy monitoring of cargo across the IWT chain and increases accountability of all IWT stakeholders. On top of that, the employment of smart contracts offers an automated and accountable solution for contract negotiation and application.

The different components presented throughout this report along with the remaining components of WP1 - such as Big Data analytics and Machine Learning - constitute the different technical solutions that were developed in close collaboration among all WP1 partners and are brought together to deliver the Open IWT Platform.

6 References

[1] “POINT-TO-POINT AND PUBLISH/SUBSCRIBE MESSAGING MODEL,” *DEV Community* 👩‍💻👨‍💻. https://dev.to/tranthanhdeveloper/point-to-point-and-publish-subscribe-messaging-model-41j0 (accessed Nov. 29, 2022).

[2] “Apache Kafka,” *Apache Kafka*. https://kafka.apache.org/ (accessed Nov. 01, 2021).

[3] “Transmission Control Protocol,” Internet Engineering Task Force, Request for Comments RFC 793, Sep. 1981. doi: 10.17487/RFC0793.

[4] “Keycloak.” https://www.keycloak.org/ (accessed Nov. 01, 2021).

[5] “Security Assertion Markup Language,” *Wikipedia*. Oct. 27, 2021. Accessed: Nov. 01, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Security_Assertion_Markup_Language&oldid=1052189680

[6] “OpenID Connect | OpenID,” Aug. 01, 2011. https://openid.net/connect/ (accessed Nov. 01, 2021).

[7] “OAuth 2.0 — OAuth.” https://oauth.net/2/ (accessed Nov. 01, 2021).

[8] “Lib-Kafka-OAuth.” kafka-security, Nov. 01, 2022. Accessed: Dec. 05, 2022. [Online]. Available: https://github.com/kafka-security/oauth

[9] H. -N. Dai, Z. Zheng and Y. Zhang, "Blockchain for Internet of Things: A Survey," in *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8076-8094, Oct. 2019, doi: 10.1109/JIOT.2019.2920987.

[10] Md Ashraf Uddin, Andrew Stranieri, Iqbal Gondal, Venki Balasubramanian, A survey on the adoption of blockchain in IoT: challenges and solutions, Blockchain: Research and Applications, vol. 2, i. 2, 2021, https://doi.org/10.1016/j.bcra.2021.100006.

[11] M. A. Ferrag, M. Derdour, M. Mukherjee, A. Derhab, L. Maglaras and H. Janicke, "Blockchain Technologies for the Internet of Things: Research Issues and Challenges," in IEEE Internet of Things Journal, vol. 6, no. 2, pp. 2188-2204, April 2019, doi: 10.1109/JIOT.2018.2882794.