Ecosystem for COllaborative Manufacturing PrOceSses – Intra- and Interfactory Integration and AutomaTION
(Grant Agreement No 723145)

D4.5 Prototype of the Security Framework II

Date: 2019-02-28

Version 1.0

Published by the COMPOSITION Consortium

Dissemination Level: Public

Co-funded by the European Union’s Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145

Document control page

Document file:	D4.5 Prototype of the Security Framework II v1.0.docx
Document version:	1.0
Document owner:	ATOS
Work package:	WP4 – Secure Data Management and Exchange in Manufacturing
Task:	T4.1 – Security by design for cloud-based data exchange T4.3 - Knowledge Protection, IPR Protection and Trust for Collaborative… T4.4 - Cyber Security for Factories
Deliverable type:	OTHER
Document status:	☒ Approved by the document owner for internal review ☒ Approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Nacho González, Rodrigo Díaz, David Rojo, Mario Faiella (ATOS)	2019-01-02	First version of the deliverable.
0.2	Nacho González, Rodrigo Díaz, David Rojo, Mario Faiella (ATOS)	2019-02-07	Security Framework architecture update
0.3	Nacho González, Rodrigo Díaz, David Rojo, Mario Faiella (ATOS)	2019-02-14	Deployment contributions
0.4	Nacho González, Rodrigo Díaz, David Rojo, Mario Faiella (ATOS)	2019-02-20	Update on deployment contributions. Ready for internal review
0.5	Patrick McCallion (BSL)	2019-02-22	Internal review by BSL
0.6	Matteo Pardi (NXW)	2019-02-25	Internal review by NXW
1.0	Nacho González, Rodrigo Díaz, David Rojo, Mario Faiella (ATOS)	2019-02-27	Internal review comments addressed
			Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Patrick McCallion (BSL)	2019-02-22	• Update table of acronyms • Typos • Grammar and orthography
Matteo Pardi (NXW)	2019-02-25	• Typos • Issues with the architecture diagram • Reorganize the chapters

Index:

1 Abbreviations and Acronyms

Acronym	Meaning
OTP	One Time Password
SSL	Secure Sockets Layer
HTTPS	Hyper Text Transfer Protocol Secure
REST	Representational State Transfer
RaaS	RabbitMQ authentication and authorization Service
GaS	GUI authorisation Service
PAP	Policy Administration Point
XACML	eXtensible Access Control Markup Language
PA	Policy Administrator
PDP	Policy Decision Point
BGW	Border Gateway
MQTT	Message Queuing Telemetry Transport
JWT	JSON Web Token
AMQP	Advanced Message Queuing Protocol
UUID	Universally Unique Identifier

2 Introduction

2.1 Purpose, context and scope of this deliverable

This document provides an update of the implementation and deployment of the specifications and requirements of the Security Framework. The first version of this document, delivered in M20, reported the initial stages of the deployment process, and this document will update that information with a report of the changes performed due to specific needs of the project and the implementation and deployment of new and pending components of the Security Framework, aimed to address the requirements and the securitization of the different components and processes of the COMPOSITION environment.

The previous deliverables where the specification and requirements were stated are:

- D4.1 – Design of the Security Framework I (delivered in M12)
- D4.2 – Design of the Security Framework II (delivered in M18)

And the previous version of this document is:

- D4.4 – Prototype of the Security Framework (delivered in M20)

The final version of the Security Framework prototype guarantees confidentiality and integrity of the information transmitted. This is assured by three groups of security mechanisms:

- Authentication
- Access control: based on a security token included in different requests and the evaluation of security policies.
- Transport security

2.2 Content and structure of this deliverable

The structure of this document is divided into two main blocks

- **General Security Framework architecture**: description of the architecture and main updates from the first version of the Security Framework.
- **Deployment**: functional and technical report about the implementation and deployment of the different components of the Security Framework.

3 General Security Framework architecture

The Security Framework provides a complete securitization of the COMPOSITION platform components, in terms of authentication, authorization and cybersecurity of all the communications involved in the different processes.

Below we can find an update of the architecture of the Security Framework. The main differences from the previous version described in D4.4 – Prototype of the Security Framework I are:

- GaS component: aimed to address the securitization of the different graphic user interfaces.
- XL-SIEM component which will provide real-time event analysis and protection against cybersecurity threats.

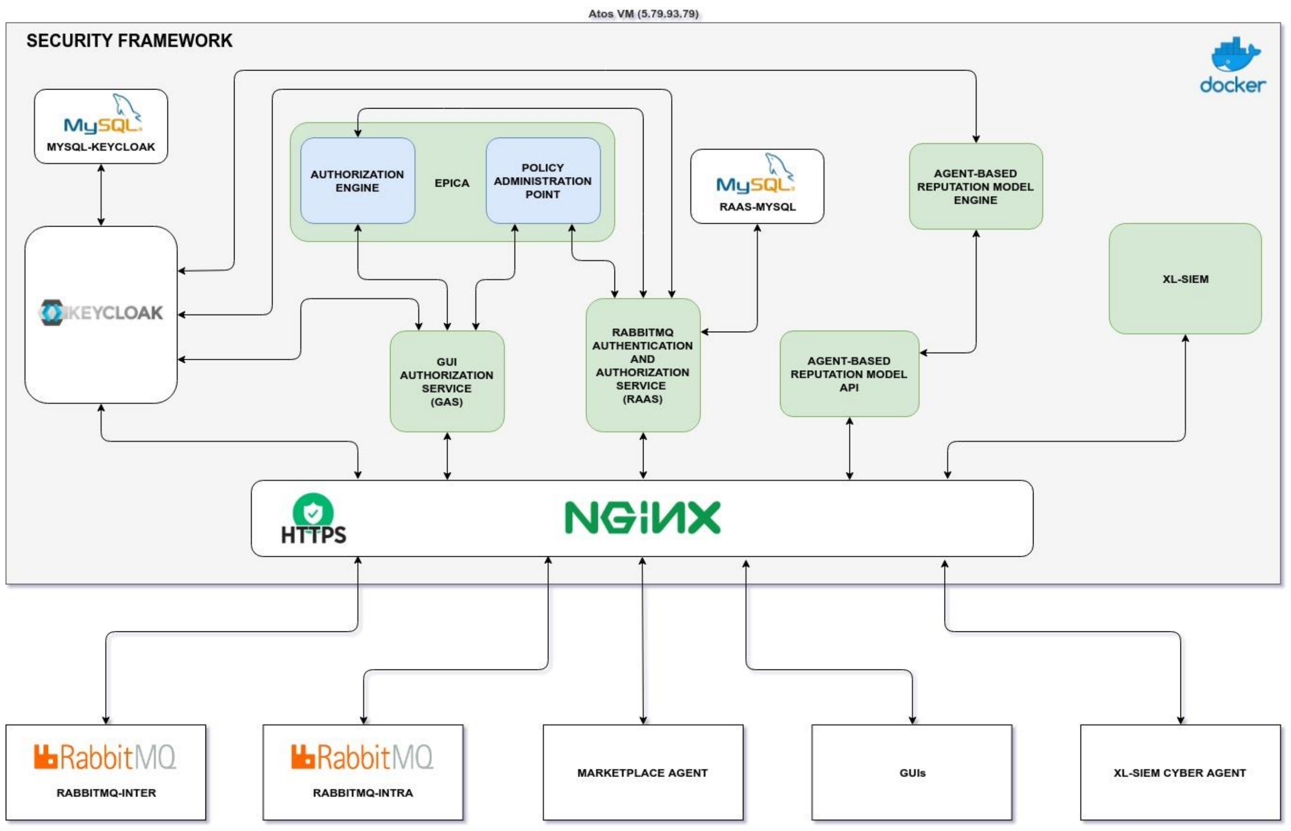


Figure 1 Security Framework architecture diagram

3.1 REST API Security with Border Gateway

All COMPOSITION components which expose RESTful APIs over the internet must enforce authentication using OpenID Connect. LinkSmart Border Gateway[1] (BGW) can secure these APIs by providing an overlay on top of all RESTful APIs, passing only authenticated and authorized requests to them.

These two authentication methods are currently available for COMPOSITION:

- Basic authentication

  1. User provides username/password in the REST request
  2. BGW intercepts the request and negotiates with an OpenID Connect server for an access token

[1] https://docs.linksmart.eu/display/BGW

3. If authenticated, BGW forwards the request to API

- Bearer token
    1. User directly negotiates with an OpenID Connect server for an access token.
    2. User provides the access token in the request
    3. BGW intercepts the request and validates the token
    4. If authenticated, BGW forwards the request to the API

Regarding authorization, BGW is able to enforce policy-based authorization based on request path and HTTP methods. The policy should be given to BGW as part of the JWT during authentication. The policies are profile attributes assigned to users and groups as part of their accounts in the OpenID Connect server. The rules format allows wildcards # and + in the same way it is commonly used for MQTT topics (see here[2] for a more elaborate documentation).

The following diagram shows the HTTP communication within the ecosystem:

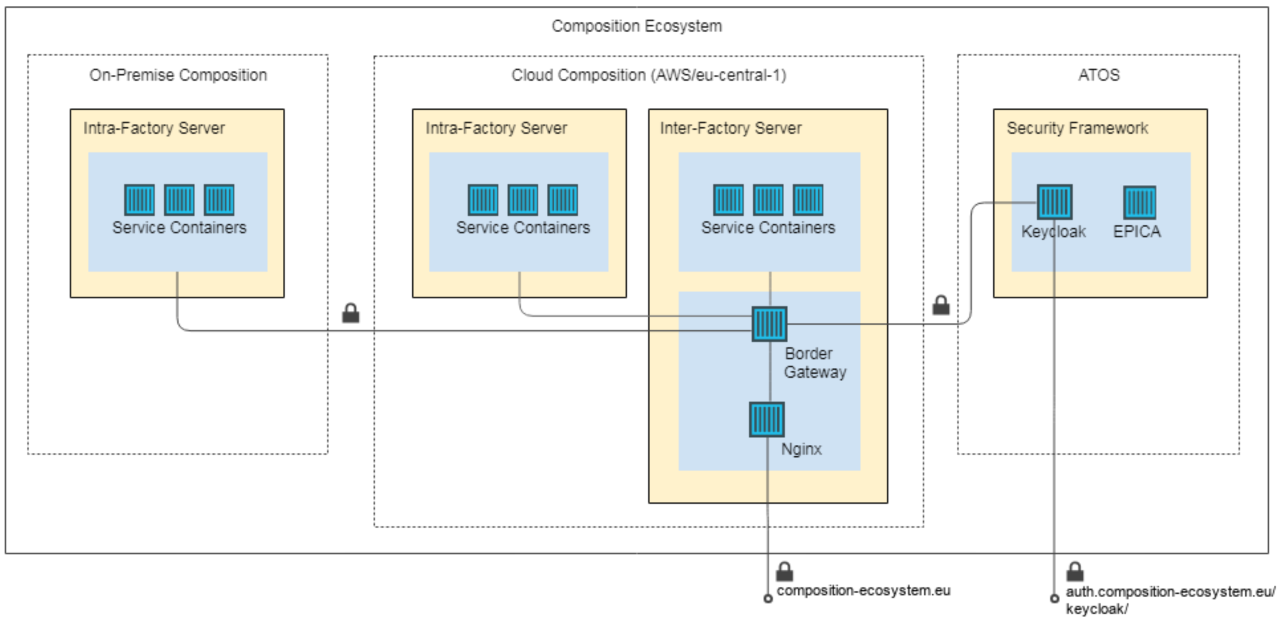


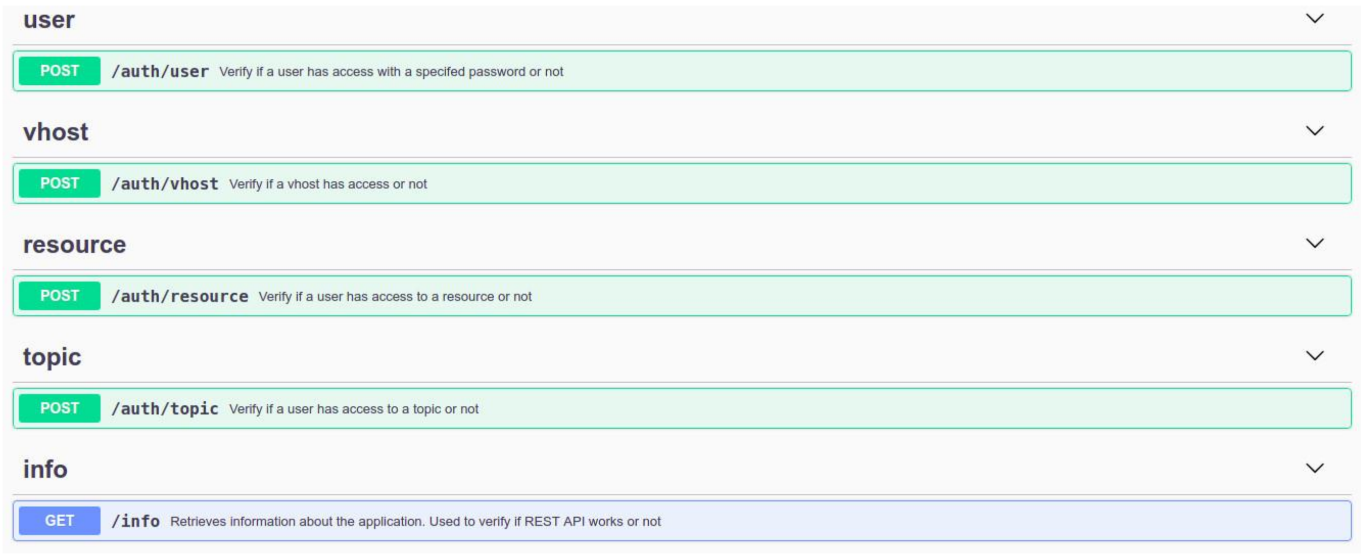
Figure 2 HTTP communication within the COMPOSITION ecosystem

[2] https://docs.linksmart.eu/display/BGW/Setting+up+Keycloak+as+an+OpenID+Connect+provider

4 Deployment

4.1 RaaS Deployment

RaaS exposes different REST APIs, each of them associated to a different level of access control. The following figure describes the endpoints and methods of the API:



user

POST /auth/user Verify if a user has access with a specifed password or not

vhost

POST /auth/vhost Verify if a vhost has access or not

resource

POST /auth/resource Verify if a user has access to a resource or not

topic

POST /auth/topic Verify if a user has access to a topic or not

info

GET /info Retrieves information about the application. Used to verify if REST API works or not

Figure 3 RaaS API endpoints

RaaS is used for granting access to the rabbitMQ message broker, while GaS component, described in the next section, is used by the need of granting authorization policies to the different GUIs of the COMPOSITION platform.

Regarding the deployment, the deployment has been done on the production server. The configuration parameters can be found in the chapter 7 – Annex II and the following plugins are activated:

- rabbitmq_auth_backend_http
- rabbitmq_auth_backend_cache
- rabbitmq_mqtt
- rabbitmq_web_mqtt

4.1.1 RaaS authorization policies

RaaS component will fetch any request that is coming from the RabbitMQ[3] broker. It exposes five different APIs, summarized in Table 1. The first four of them are related to different level of access control, as well explained in RabbitMQ specification for access control[4], where also specific concepts such as 'virtual host', 'resource', 'topic' and 'routing key' are explained. Instead, the last API simply allows the user retrieving some generic information about the set of REST APIs, indeed it will not be taken in consideration from now on.

Table 1 RaaS Exposed APIs

Endpoint	Method	Path	Description
User	POST	/auth/user	Used for authenticating the user. Authorization is not involved at this step.
Vhost	POST	/auth/vhost	Verify if a user is authorized access a specific virtual host
Resource	POST	/auth/resource	Verify if a user is authorized to access a specific resource (e.g., exchange, queue)

[3] http://www.rabbitmq.com/
[4] https://www.rabbitmq.com/access-control.html

Topic	POST	/auth/topic	Verify if a user is authorized to access a specific AMQP (or MQTT) topic, associated to a specific resource (e.g., exchange, queue)
Info	GET	/auth/info	Retrieves information about the application. Used to verify if REST API works or not.

Every time a user requests the usage of a specific resource to RabbitMQ, he made four different sequential calls to the endpoints specified in Table 1, and for each of them an access control policy is created. For each request it receives, RaaS firstly interacts with Keycloak for getting the token on behalf of the user who is making the request itself. All the information needed for generating the policies can be found in this token, specified through the custom attribute "raas_authz_rules", that can contain more than one authorization rule, separated by two white spaces characters. Each rule is composed by at most four attributes, following a specific order, separated by a single white space character, and, depending on which attributes are specified, the parser can infer what kind of RabbitMQ access control level is requested.

The set of attributes that could be specified, and the related order, is the following:

- **Exchange (optional)**: name of the exchange involved in the received request (only when /topic endpoint is called)
- **Vhost (mandatory)**: name of the virtual host that want to be accessed
- **Permission (mandatory)**: type of operation that the user would like to perform on the selected resource (e.g., write, read, configure)
- **Resource_regexp (optional)**: regular expression used for matching the name of the resource (/resource endpoint) or the routing key (/topic endpoint) received through the request

For example, if just the two mandatory attributes are specified, the rule parser can infer that this authorization rule should be used when the user asks for the permission of accessing a specific virtual host (/vhost endpoint). If both the optional parameters are present, then the rule should be considered when /topic endpoint is called. Finally, if just the 'resource_regexp' parameter has been specified, together with the mandatory ones, it means that the /resource endpoint has been called, that is a user is requesting the permission to perform an operation on the selected resource belonging to a specific virtual host.

Besides, to increase the rules flexibility, special wildcards could be used, merging the guidelines of both MQTT (add footnote in confluence) and AMQP[5] protocols, which are both supported by RabbitMQ and used in COMPOSITION context. More precisely, the available wildcards are the following:

- **#:** every combination of characters is allowed starting from this character. It could be used for virtual hosts, MQTT and AMQP resources names (e.g., exchanges, queues) and topics
- **+**: if used for 'permission' parameter, every possible operation is allowed (write, read or configure for AMQP cases, while publish and subscribe for MQTT ones). Besides, it could also be used for expressing a generic single word between two '-' characters when MQTT resources and topics are involved.
- *: it could be used for expressing a generic single word between two '.' characters when AMQP resources and topics are involved.

In Table 2, some examples are described:

Table 2 RaaS Authorization Rules examples

Rule	Endpoint called	Description
vh=/ write amq.example.#	/resource	Allows write operation on every resource whose name start with "amq.example.", which belongs to the default virtual host (/).
vh=/ read mqtt-composition-example	/resource	Allows read operation (equivalent of subscribe operation for MQTT) on resources named mqtt-composition-

[5] https://www.rabbitmq.com/tutorials/tutorial-five-python.html

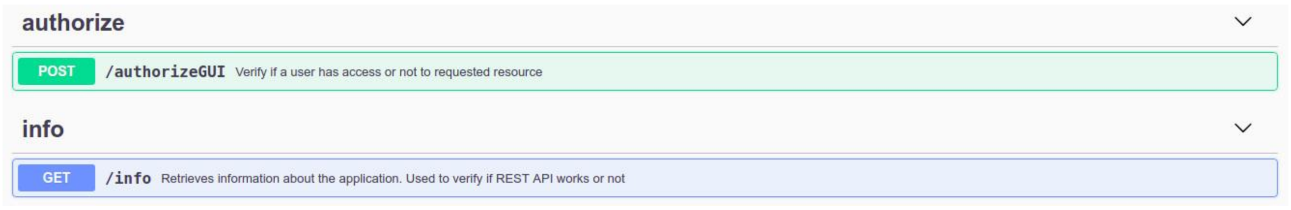
		example, which belongs to the default virtual host (/).
amq.topic vh=example + Composition.BMS.#	/topic	Allows every operations on amq.topic exchange, belonging to virtual host example, if the routing key specified in the request match the regular expression in the rule.
vh=#	/vhost	Allows the access on every virtual host.
vh=# + #	/resource	Allows every operation on every resource in every virtual host.

RaaS is able to parse each rule located in the ‘raas_authz_rules’ attribute of the Keycloak token, and create a JSON with a predefined structure needed by EPICA component, described in Section 4.4.1, for creating the final access control policy, or better, the final policy set, considering that each rule will be converted into a single policy, and all of them embedded in the same policy set. The integration between RaaS and EPICA will be explained in Section 4.4.2. Additionally, the concept of group in Keycloak will also be used for implementing a basic policy inheritance. This means that if a user belongs to a group, he will automatically inherit all the authorization rules associated to that group, specified through the group-level attribute ‘raas_authz_rules_<group_name>’, which could be also extracted from the token, but only if the user is effectively a member of that group. For more information about how groups are handled in Keycloak, the reader is advised to read Keycloak documentation.

4.2 GaS (GUI Authorization Service) Deployment

4.2.1 Overview

GaS component will be in charge of guaranteeing the authorization service for GUIs[6] It does not perform authentication, as RaaS does, indeed it expects to receive the Keycloak token in the request. The caller is responsible for retrieving the Keycloak access token and, then, forward it to GaS. It will be another secure entry-point for the Security Framework.


authorize

POST /authorizeGUI Verify if a user has access or not to requested resource

info

GET /info Retrieves information about the application. Used to verify if REST API works or not

Figure 4 GaS API endpoints

GUI authorization Service (GaS) has been added to handle authorization of Graphical User Interfaces (GUIs), used in COMPOSITION scenarios. When RaaS is involved, it means that the user is trying to access a specific resource (usually a MQTT/AMQP resource), sending the request through the RabbitMQ broker. Once the request reaches RaaS, it will interact both the Authentication Service (Keycloak) and the Authorization Service (EPICA), to authenticate the user and check if he has the rights for performing the requested operation.

When GUI are involved, instead, the workflow is slightly different. Indeed, the GUI itself must interact directly with Keycloak, for example during the login, to authenticate the user who is using the application, without passing through the broker. This is possible allowing the so-called “Implicit Flow”[7], specifying in Keycloak itself all the available URLs to which Keycloak can redirect the application after the user login.

More in details, the GUI performs a GET request to the following URL:

- https://<keycloack_domain>/keycloak/realms/<realm_name>/protocol/openid-connect/auth

[6] https://auth.composition-ecosystem.eu/gas/gas-api/index.html
[7] https://www.keycloak.org/docs/3.3/server_admin/topics/sso-protocols/oidc.html

Specifying the following parameter in the URL:

- client_id: id of the Keycloak client involved
- redirected_uri: URL to which Keycloak will redirect the application, in case of successfully login
- response_type: equal to "token". Allows obtaining the access token which could be retrieved from the callback URI provided by Keyloak in case of successfully login
- nonce: a random UUID will be specified

The redirection will end successfully if the redirected_uri, specified by the GUI, is present in the list of valid redirecting URLs, in the configuration of the used Keycloak client, where also the "Implicit Flow" must be enabled.

Finally, the GUI can retrieve the access token, from the callback URL provided by the Authentication Service. At this point, the user, through the GUI, is authenticated, but not still authorized. For this reason, GaS component has been added to the Security Framework. The GUI creates the access control request, specifying the token in the Authentication Header of the HTTP POST request, sent to GaS, together with the information of the resource that want to be accessed. This component will extract all the information from the incoming request, interacting with EPICA for checking if the user is authorized. EPICA final answer, is returned to the GUI, which, in turn can complete the overall process.

In Section 4.2.1, how GaS authorization policies are built, starting from the Keycloak access token, is described. Instead, in Section 4.4.2, the integration between GaS and EPICA, in terms of policies creation, management and enforcement will be detailed.

We can synthetize what GaS does in the following diagram:

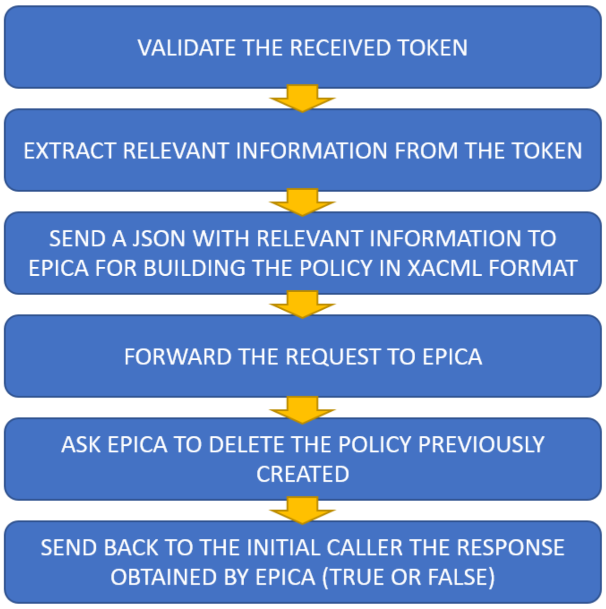

VALIDATE THE RECEIVED TOKEN
EXTRACT RELEVANT INFORMATION FROM THE TOKEN
SEND A JSON WITH RELEVANT INFORMATION TO EPICA FOR BUILDING THE POLICY IN XACML FORMAT
FORWARD THE REQUEST TO EPICA
ASK EPICA TO DELETE THE POLICY PREVIOUSLY CREATED
SEND BACK TO THE INITIAL CALLER THE RESPONSE OBTAINED BY EPICA (TRUE OR FALSE)

Figure 5 GaS activities diagram

4.2.2 GaS authorization policies

GaS authorization policies will be defined starting from the authorization rules specified in the Keycloak token, through the 'bgw_rules' attribute, in a similar way already explained for RaaS component in Section 4.1.1. As discussed in the previous section, GaS receives requests not from the broker, but directly from the GUIs, that have already interacted with Keycloak by themselves for retrieving the token, for requesting the access of a specific URI. Parameters have a predefined order, and, as for RaaS case, more rules could be specified in the same 'bgw_rule' attribute. Rules will be separated among each other by a single white space character, while the different attributes composing a single rule, by a '/' character.

The set of parameters, and the order, is the following:

- **Protocol**: transfer protocol (e.g. HTTP, HTTPS)
- **Method**: method regarding the protocol (e.g. GET, POST, DELETE)
- **Domain**: fully qualified domain name (FQDN)
- **Port**: TCP port
- **Path**: it represents resource information and may contain slashes itself. For HTTP, it is the resource path

For GaS cases, every parameter is mandatory. Also, for this component, some wildcards are allowed, for having more flexibility when defining authorization rules in Keycloak. An important difference with respect to RaaS, with respect to wildcards usage, is related to the possibility of specifying more parameters using one of these special characters. The available wildcards are the following:

- **#:** every combination of characters is allowed starting from this character. If applied to a specific attribute, it will be implicitly extended to all the subsequent attribute, considering the predefined order
- **+**: used for representing a generic word, or character combination, between two '/' characters. It means that it has a validity only for single parameters.

In Table 3 some practical examples are described:

Table 3 GaS Authorization Rules Examples

Rule	Description
HTTPS/GET/intra.composition-ecosystem.eu/443/sc/#	Allows • GET method over HTTPS • on domain intra.composition-ecosystem • port 443 • path sc as well as any other one starting with sc
HTTPS/+/intra.composition-ecosystem.eu/443/#	Allows • all methods • over HTTPS • on domain intra.composition-ecosystem • port 443 • with any path
HTTPS/+/+/443/sc/admin	Allows • all methods • over HTTPS • on every domain • port 443 • with path equal to sc/admin
HTTPS/#	Allows • all methods

	• over HTTPS • on every domain • on every port • with any path

Also, for GaS, the policy inheritance functionality has been implemented exploit how Keycloak manage groups. The approach is the same described for RaaS in Section 4.1.1. The only differences are the name of the Keycloak attribute that should be extracted, called 'bgw_rules_<group_name>' and, obviously, the syntax of the rules, as explained above.

As Raas does, this component can convert these authorization rules in a predefined JSON understandable by EPICA for allowing it building the overall policy set. The integration between GaS and EPICA will be detailed in Section 4.4.2.

4.3 Keycloak

The authentication service has been deployed using Keycloak. For management purposes, we've elaborated a cheat sheet available in Annex 1 with the most common operations to be performed by the different users of the authentication service.

4.4 EPICA overview and deployment

EPICA is an Attribute Based Access Control (ABAC) tool based on XACML v3.0[8]. The main idea behind EPICA is to provide the means to manage access control policies and then, evaluate incoming requests against these policies, when specific accesses should be granted. More specifically, the role of EPICA in COMPOSITION is to provide the Authorization service for the Security Framework. EPICA functionalities have been described in detailed in (COMPOSITION D4.2, 2018) and (COMPOSITION D4.4, 2018), focusing upon the two sub-component that characterize this tool, the Policy Administration Point (PAP), in charge of managing access control policies, and the Authorization Engine, responsible for policies enforcement task. Both offers a set of REST APIs, used for initiating the related processes.

For the sake of completeness, the overall architecture is depicted in Figure 6.

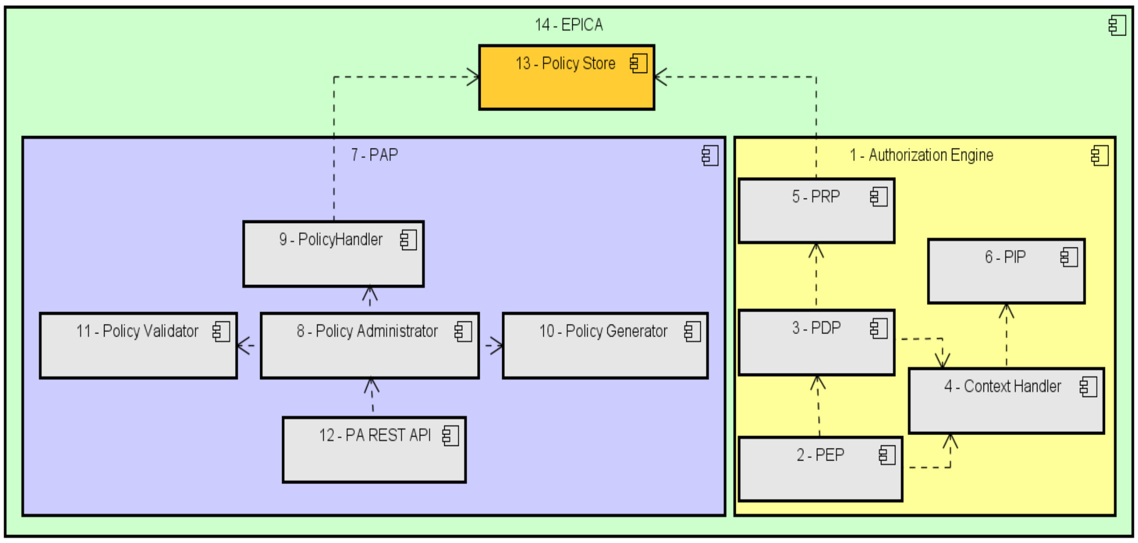


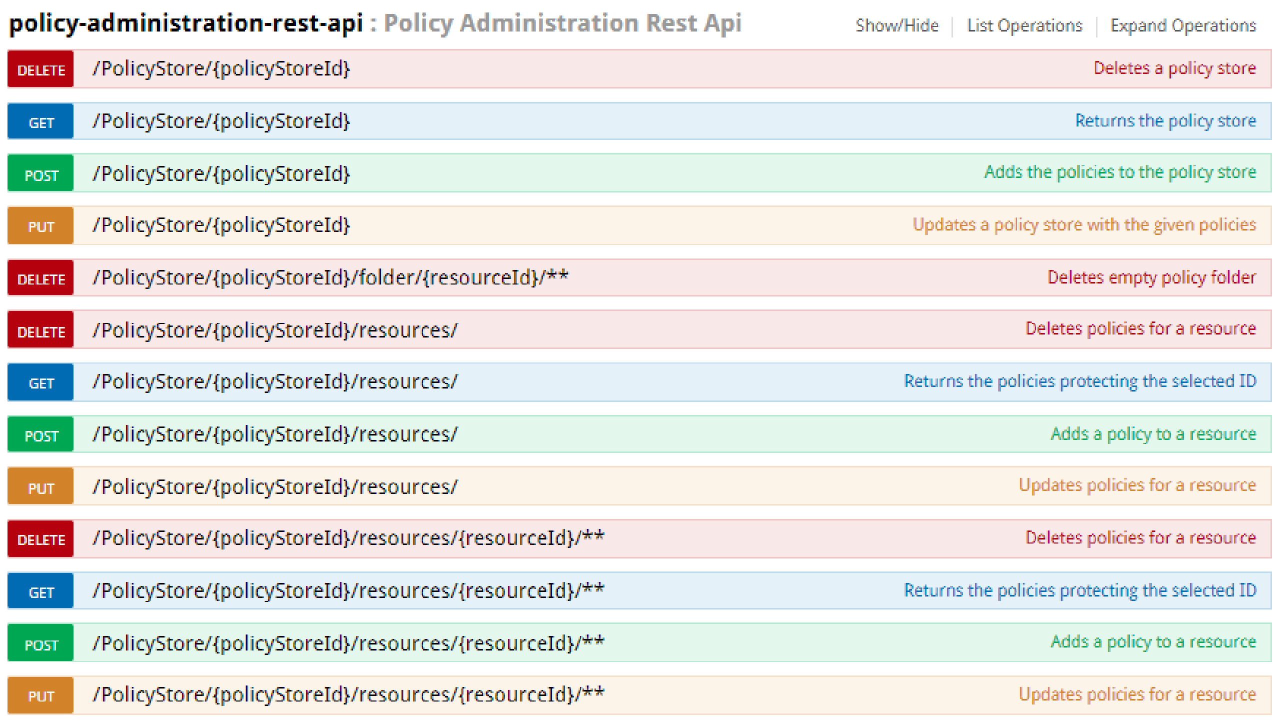
Figure 6 EPICA architecture

[8] http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

The set of REST APIs, instead, of the two sub-components are shown in Figure 7 and Figure 8. The functionalities offered by each APIs have already been detailed in (COMPOSITION D4.4, 2018), when the first prototype of the Security Framework has been introduced. With respect of those versions, just few changes have been made. Regarding the PAP, only a new endpoint has been added, allowing the requester to delete an empty folder, which previously contained one or more access control policies. For the Authorization Engine, instead, more changes have been made. Now, it is exposing two POST method, both for initializing the policy enforcement process, with the difference that the second, which has more relevance in the COMPOSITION scenario, is expecting the Keycloak token in the Authorization Header of the HTTP request, as can be seen in Figure 9. This feature is needed by the Authorization Engine for extracting specific attribute from it, to build the final XACML request.

More information about the APIs will be provided in Section 0 and Section 4.4.2, when the integration between RaaS and GaS with EPICA will be described.

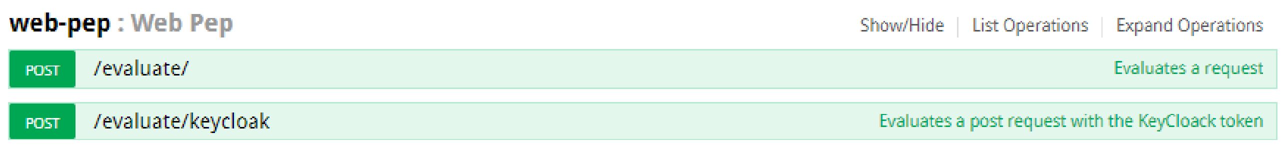
For improving the flexibility in terms of building authorization rules, which will be converted in access control policies, the usage of specific wildcards is allowed, following the guidelines for both MQTT and AMQP, as detailed in Section 4.1.1 and Section 4.2.1. EPICA must handle properly these wildcards, for this reason the matching of all the parameter (e.g., mqtt/amqp topic and resources, http resource paths) who are using them will be matched through regular expressions.


policy-administration-rest-api : Policy Administration Rest Api

Show/Hide | List Operations | Expand Operations

DELETE	/PolicyStore/{policyStoreId}	Deletes a policy store
GET	/PolicyStore/{policyStoreId}	Returns the policy store
POST	/PolicyStore/{policyStoreId}	Adds the policies to the policy store
PUT	/PolicyStore/{policyStoreId}	Updates a policy store with the given policies
DELETE	/PolicyStore/{policyStoreId}/folder/{resourceId}/**	Deletes empty policy folder
DELETE	/PolicyStore/{policyStoreId}/resources/	Deletes policies for a resource
GET	/PolicyStore/{policyStoreId}/resources/	Returns the policies protecting the selected ID
POST	/PolicyStore/{policyStoreId}/resources/	Adds a policy to a resource
PUT	/PolicyStore/{policyStoreId}/resources/	Updates policies for a resource
DELETE	/PolicyStore/{policyStoreId}/resources/{resourceId}/**	Deletes policies for a resource
GET	/PolicyStore/{policyStoreId}/resources/{resourceId}/**	Returns the policies protecting the selected ID
POST	/PolicyStore/{policyStoreId}/resources/{resourceId}/**	Adds a policy to a resource
PUT	/PolicyStore/{policyStoreId}/resources/{resourceId}/**	Updates policies for a resource

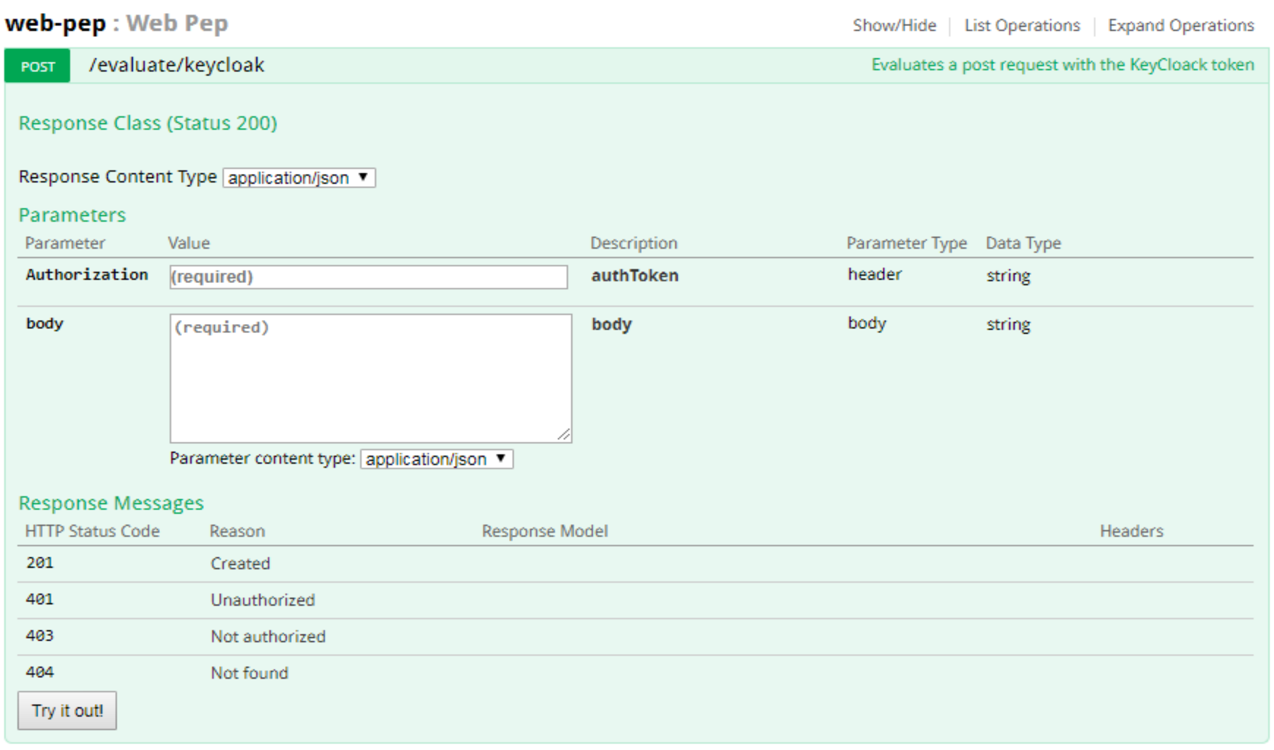
Figure 7 Policy Administration Point API


web-pep : Web Pep

Show/Hide | List Operations | Expand Operations

POST	/evaluate/	Evaluates a request
POST	/evaluate/keycloak	Evaluates a post request with the KeyCloack token

Figure 8 Authorization Engine API


web-pep : Web Pep — Show/Hide | List Operations | Expand Operations

POST /evaluate/keycloak — Evaluates a post request with the KeyCloack token

Response Class (Status 200)

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
Authorization	(required)	authToken	header	string
body	(required)	body	body	string

Parameter content type: application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Not authorized		
404	Not found		

Try it out!

Figure 9 Authorization Engine API Details

To let EPICA able to do that, the Policy Decision Point (PDP), that is based on WSO2 Balana Open Source implementation[9], adding, at runtime, a new matching function to the pool of default function that Balana is offering, to specially fit the regular expression matching needed in COMPOSITION, without modifying anything of the core software of Balana itself. More information about EPICA PDP can be found in (COMPOSITION D4.4, 2018).

About the deployment, a docker version of both the sub-component have been generated and deployed in ATOS premises using the docker-compose tool, in the same virtual network of the security framework, saving valuable information, such as logs and policy folders, in docker volumes. For having a better view about the overall deployment of EPICA, the reader is advised to read (COMPOSITION D4.4, 2018).

4.4.1 RaaS – EPICA Integration

As already explained, RaaS component is one of the entry point of the Security Framework, managing all the access requests that come from the RabbitMQ broker. First, it authenticates the user who is making the request, retrieving a token from Keycloak, using the received credentials (username and password). Then, it extracts the information contained in the token for retrieving the information needed to build a JSON compliant with the one that the Policy Administration Point (PAP) component of EPICA is expecting to receive (Section 4.1.1). Once this JSON has been built, PAP REST APIs are used for building the real XACML policy, specifying the policy store in which it will be stored, a UUID as resource identifier, and the JSON itself as the POST request body.

After that, Authorization Engine API is invoked, specifying the Keycloak token in the Authorization Header of the request. Moreover, RaaS will forward the information received by the RabbitMQ broker (e.g., vhost, resource to be accessed, permission, eventual topic), inserting it in the request body, to allow building the correct XACML request, used for matching the policy created in the above step.

For letting the Authorization Engine able to retrieve the correct policy, also the UUID previously associated to the policy is added to the body of the incoming request. Depending on the HTTP Status of the HTTP Response, RaaS can recognize the outcome of the enforcement. Moreover, the Authorization Engine, return, within the HTTP Response, a body containing a Boolean value, true if the final decision is to allow the access control request, false otherwise, which will, in turn, forwarded to the broker.

[9] https://github.com/wso2/balana

At the end of the process, RaaS invokes again the PAP REST APIs, this time for deleting first the policy in the folder identified by the previously created UUID, and, then, the folder itself.

The entire process is summarized through the activity diagram shown in Figure 10.

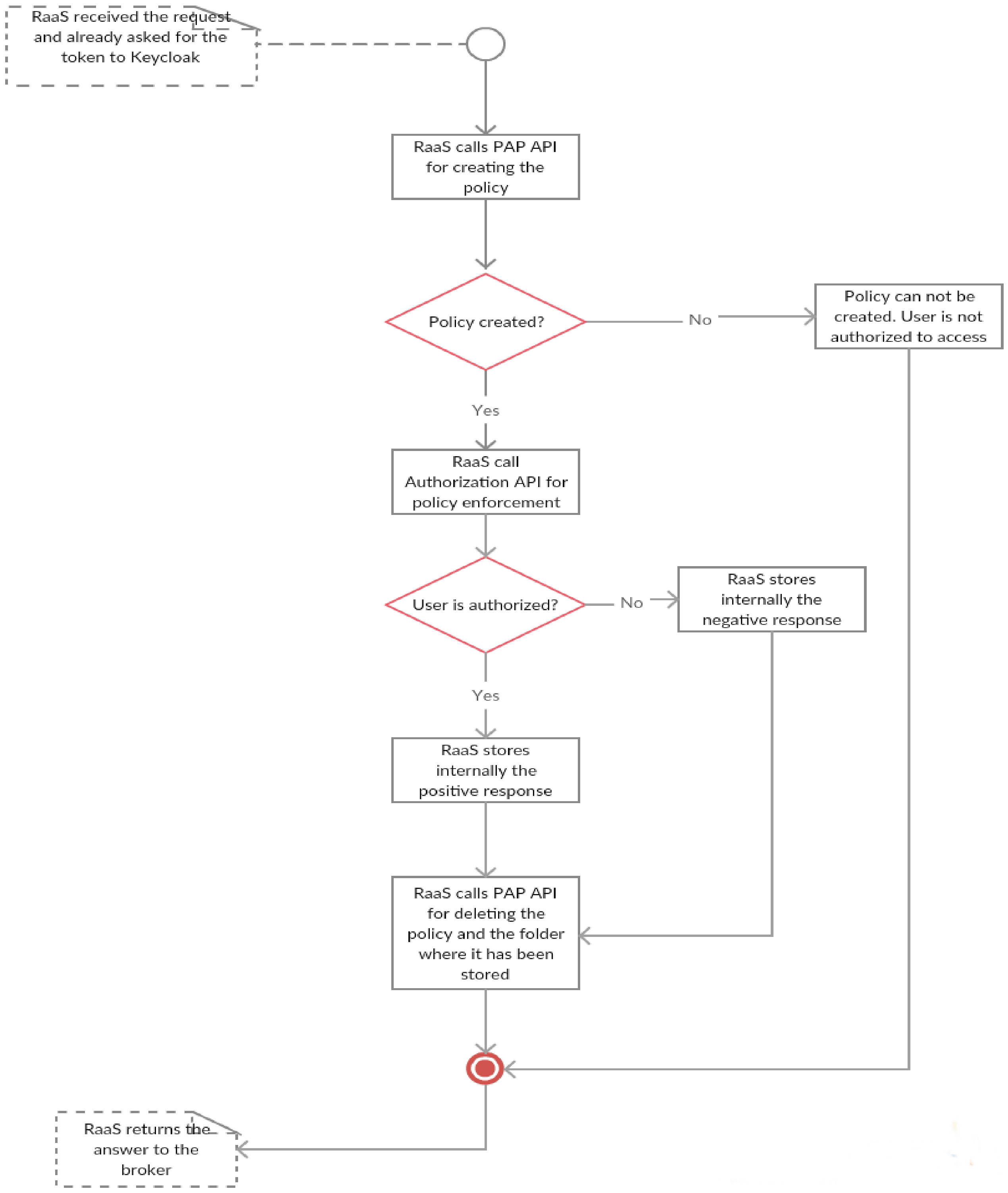

RaaS received the request and already asked for the token to Keycloak
RaaS calls PAP API for creating the policy
Policy created?
No
Policy can not be created. User is not authorized to access
Yes
RaaS call Authorization API for policy enforcement
User is authorized?
No
RaaS stores internally the negative response
Yes
RaaS stores internally the positive response
RaaS calls PAP API for deleting the policy and the folder where it has been stored
RaaS returns the answer to the broker

Figure 10 RaaS - EPICA Activity Diagram

4.4.2 GaS – EPICA Integration

As stated in Section 4.2, GaS component constitutes another entry point for the Security Framework. Differently from RaaS, it is not connected to the RabbitMQ broker, but used for GUI authentication and authorization purposes. The requests, indeed, are coming directly from the GUIs.

It is important to remind that, in this case, the component will not directly retrieve Keycloak token, as RaaS does, but this operation is done separately by the GUI itself, previously interacting with Keycloak. This means that the GUI will provide the token to GaS, which, in turn, will forward it to EPICA.

However, despite of the differences between the two use cases, the integration with EPICA will be the same, in terms of workflow. Obviously, the customized JSON to send to PAP REST APIs will consider different attributes and values, as described in Section 4.2.1. A UUID will be generated and associated to each policy and passes to PAP as resource identifier.

The same reasoning is applied when interacting with the Authorization Engine for starting the policy enforcement process. The same API is called, then it will be EPICA that will discriminate what component has performed the call, depending on the parameter received in the HTTP POST request (e.g., protocol, method, domain, port, resource path).

Finally, GaS will interact again with the PAP for deleting the created policy and the folder where it has been stored and, at the end, returns the result of the enforcement to the GUI.

For the sake of completeness, the activity diagram of the interaction between GaS and EPICA is shown in Figure 11.

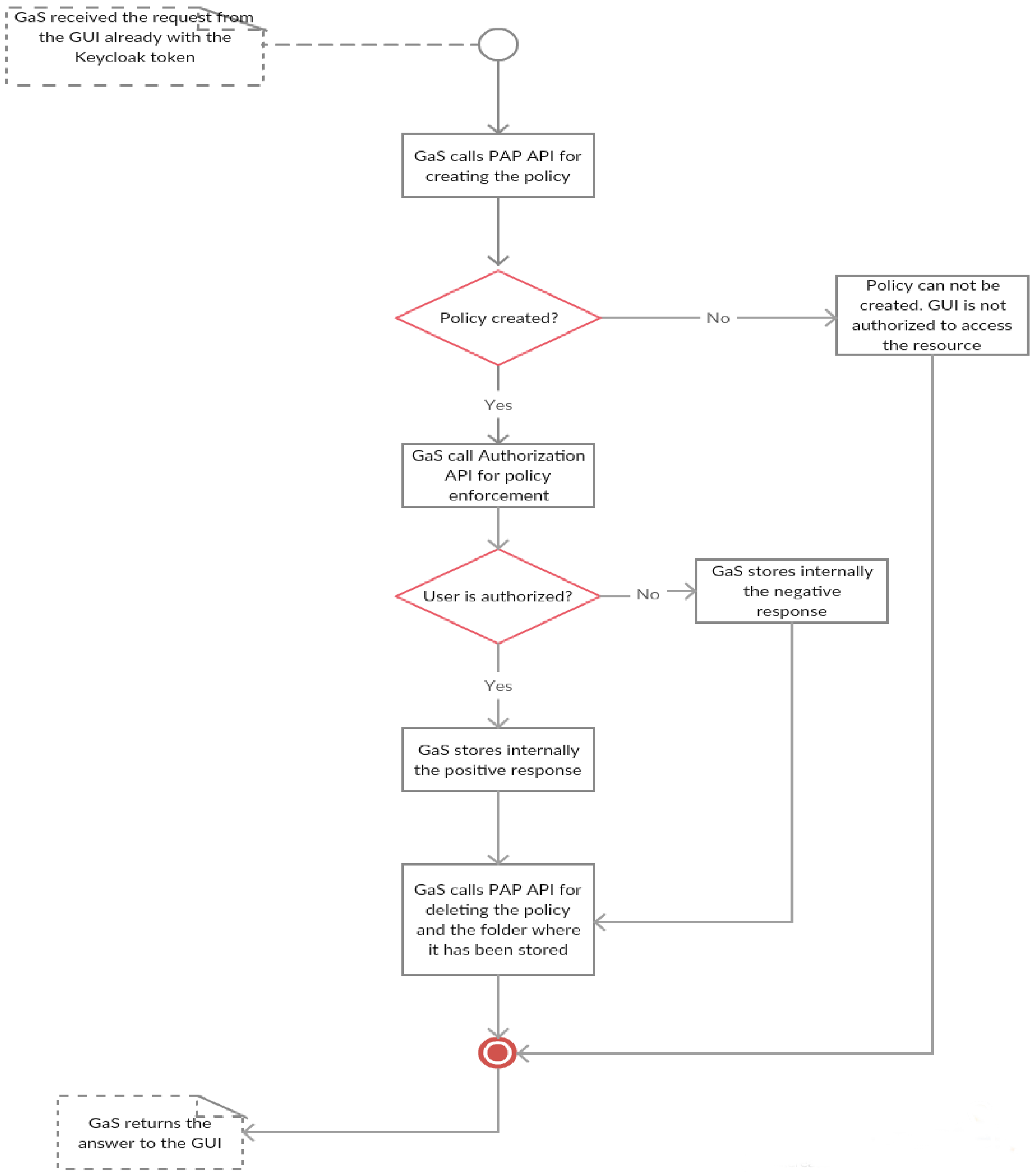

GaS received the request from the GUI already with the Keycloak token
GaS calls PAP API for creating the policy
Policy created?
No
Policy can not be created. GUI is not authorized to access the resource
Yes
GaS call Authorization API for policy enforcement
User is authorized?
No
GaS stores internally the negative response
Yes
GaS stores internally the positive response
GaS calls PAP API for deleting the policy and the folder where it has been stored
GaS returns the answer to the GUI

Figure 11 GaS - EPICA Activity Diagram

4.5 COMPOSITION Reputation Model Deployment

The aim of the COMPOSITION Reputation Model, called simply Agent-based Reputation Model, is to help Agents to make trust-based decision when choosing the counterpart with which start a new negotiation. The details about the Agents belonging to COMPOSITION context have been given in (COMPOSITION D6.3, 2018).

The requirements of the reputation model have been elicited in (COMPOSITION D4.2, 2018), while the details about the design phase have been provided in (COMPOSITION D4.4, 2018), where the decision to split it into two sub-models, called respectively Local Reputation Model and Global Reputation Model, has been explained. However, despite of this decision, the overall model implementation contains both sub-modules, which will be embedded in the Security Framework, and available on demand as a service, through a specific set of REST API, as described in Section 4.5.1.

For summarizing, the Local Reputation Model is a type of agent-level personal ranking system. Reputation is computed by a Requester Agent R with respect to a Supplier Agent S. Requesters handle these values and update them every time an interaction occurs with the same Supplier. It takes in consideration just a set of ratings, and their related timestamp, provided by R after each interaction with S. During the computation, the ratings are aggregated, weighting each of them with the associated timestamp. Indeed, newer ratings are more relevant and should have a higher weight than older ones, being compliant with the concept of reputation lifetime.

However, this approach has a huge limitation: indeed, it does not consider opinions of different Agents, which could be very valuable for having a more complete overview about the past and present behaviour of a specific Agent, or simply when the Requester has not any previous experience with the selected Supplier. From these considerations, another sub-module has been designed, called Global Reputation Model, seen more as a central and collaborative agent ranking system. This type of reputation is computed by a Trusted Third Party, the Matchmaker Agent (COMPOSITION D6.3, 2018), on behalf of a specific Requester R with respect to a Supplier S: Local Reputations that a trusted set of Requesters, from R point of view, called N_R, has with respect to S, are aggregated and weighted with the Local Reputation that R has with respect to each Requester belonging to N_R. This allows R having a global and trusted knowledge about S behaviour, depending on the opinions of other Agents.

Complete details about both Local and Global Reputation Models, especially regarding the mathematical formulas, together with some simulation tests, and the roles played by the different types of Agents, can be found in (COMPOSITION D4.4, 2018).

4.5.1 Agent-based Reputation Model Implementation and Deployment

The Agent-based Reputation Model has been implemented in Java and integrated directly within the Security Framework. A Docker[10] container has been built and deployed in Atos premises, in the same virtual network used for the Security Framework itself, as already done with the other components. It is composed by a set of APIs, and by an engine, the Agent-based Reputation Model Engine, in charge of the computation of reputation values.

The first idea was to develop it as a Java library, which could be imported directly by the various Agents involved (e.g., Requesters, Suppliers, Matchmaker). However, for facilitating the integration, the final decision was to deploy it in the Security Framework, offering a set of REST API which can be invoked by an Agent, whenever he needs. Besides, in this way, if a new version of the model is released, or simply if an update is made, there is no need, from the Agent side, to perform any kind of changes. The Agents itself will handle the creation and the storage of both ratings and timestamps and provide this value to the engine of the reputation model through the exposed APIs, shown in Figure 12.

This modification, obviously, does not affect the fulfilment of the requirements stated in (COMPOSITION D4.2, 2018). It just introduces a possible issue, related to the identity of the Agent who is making the request. Indeed, this potential problem has never been discussed before, considering the idea of deploying the model as an internal library, from the Agent point of the view. Being a remote service, the Agent-based Reputation Model Engine needs to understand if the Agents, who are requesting the computation of the reputation, effectively are the ones who are claiming to be.

[10] https://www.docker.com/

The proposed solution is to leverage on the Authentication Service provided by the Security Framework, that is Keycloak. More precisely, Agents, when interacting among each other, exchange messages through the RabbitMQ broker. To be authenticated, they provide, to the RaaS component, their username and password, or directly the Keycloak token. Indeed, it mandatory for them to have an account in Keycloak. This reasoning is valid also for the Matchmaker.

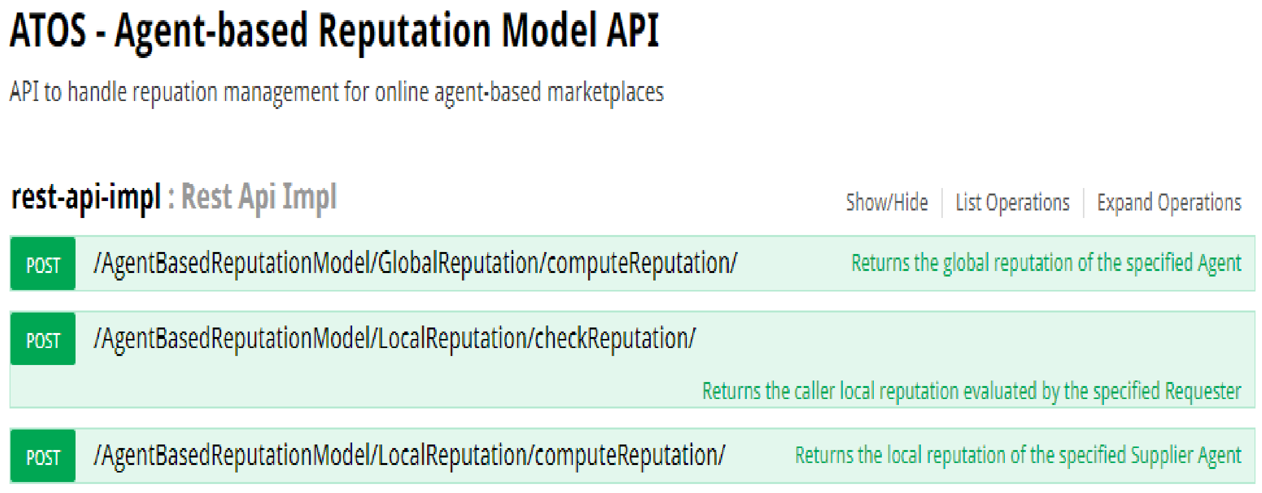
The same approach could be also for this scenario. Embedding the credentials in the body of the HTTP POST request, or directly the token in the Authorization Header, will allow the Agent-based Reputation Model Engine to infer the identity of the Agent who is asking for the service. The engine itself will retrieve the token from Keycloak, or simply validates it, if already specified in the request. Only if the authentication is performed correctly, the reputation will be computed.

The set of exposed APIs is shown in Figure 12. The first and the second are associated to the Local Reputation concept, respectively for computing the reputation of a Supplier and for allowing a Supplier to check if his reputation, evaluated on behalf of a specific Requester, is correct. This functionality has been added because ratings and timestamps are not stored by the Agent-based Reputation Model Engine itself, but they are handled by each Agent, which, potentially, could send to the engine some fake values. With this method, the Supplier could check the correctness of the reputation and, in case of some inconsistencies, ask for Matchmaker help. In (COMPOSITION D4.4, 2018) the potential crucial role of the Matchmaker has been explained in an exhaustive way.

The third one, instead, is used by the Matchmaker for evaluating Global Reputations. Considering that the Matchmaker is a trusted entity, there is no need of a checking method for this type of reputation.

All the values needed for performing the computation (e.g., ratings, timestamps, local reputations, username, password, ID of the involved agents) can be extracted from the body of the incoming HTTP POST request.

The only pending decision regarded the choice of the maximum number of ratings, called N_{max}, to consider during the computation of the Local Reputation. This value is completely application dependent, as explained in (COMPOSITION D4.4, 2018). Depending on the number of deployed Agents, as well as the expected interactions among them, a proper value will be proposed for this parameter.


ATOS - Agent-based Reputation Model API

API to handle repuation management for online agent-based marketplaces

rest-api-impl : Rest Api Impl

POST /AgentBasedReputationModel/GlobalReputation/computeReputation/ Returns the global reputation of the specified Agent

POST /AgentBasedReputationModel/LocalReputation/checkReputation/ Returns the caller local reputation evaluated by the specified Requester

POST /AgentBasedReputationModel/LocalReputation/computeReputation/ Returns the local reputation of the specified Supplier Agent

Figure 12 Agent-based Reputation Model APIs

4.6 XL-SIEM Deployment

The SIEM component aims to address cybersecurity threats by providing a real-time information and events monitoring and management. Security alerts are generated by applications and network hardware. As is described in (Mario Faiella, 2018), the Cross-Layer SIEM (XL-SIEM) overcomes the features and capabilities of the solutions analyzed, which most of them don't provide high-level security risk metrics neither strong correlation rules.

The basic architecture of the solution is described in Figure 13.

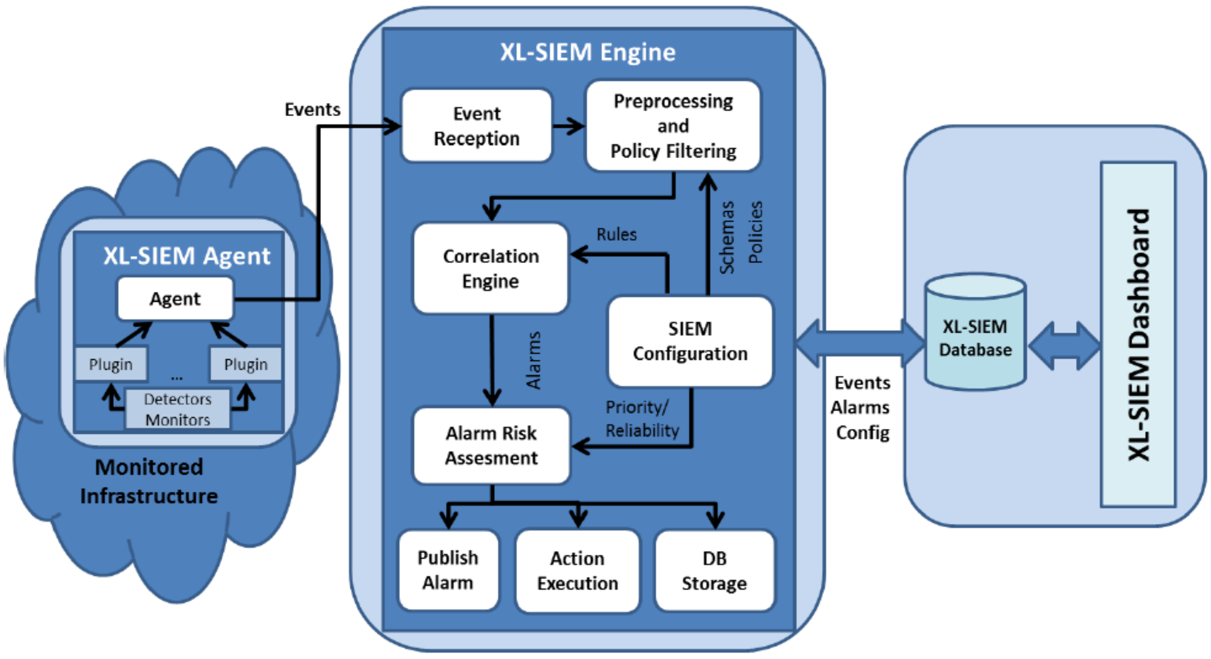


Figure 13 Cross-Layer SIEM architecture

Basically, SIEM agents monitor the infrastructure. In COMPOSITION platform, agents will be deployed in the production servers where all data are managed. The events gathered are sent to the SIEM engine, which will be deployed at Atos premises, and will be processed and correlated.

4.6.1 XL-SIEM Engine Deployment

The XL-SIEM processes and analyses the events gathered by the different XL-SIEM agents deployed in the infrastructure.

There are two different steps in the analysis process of the events:

- Pre-processing and filtering
- Correlation engine which integrates an open source high performance correlation engine (Esper). The results of this process can be grouped into:
  - Detection of patterns
  - Definition of data windows
  - Aggregation and filtering of incoming events into more complex events

The deployment of the engine is done in the same virtual machine (5.79.93.79) where the Security Framework components are running. This component is protected by the Nginx firewall for the securitization of the communications with the cyber agents.

4.6.2 XL-SIEM Agents Deployment

As was previously detailed, the agents will be deployed in the production servers gathering and collecting all data related to events in order to provide that information to the XL-SIEM engine. Furthermore, the agents will normalize and transfer the information obtained.

The communication with the XL-SIEM engine supports TLS (Transport Layer Security) certificates, in order to address all security and privacy requirements.

At this moment of the project, there are planned one agent in each use case production server, with direct connection to the XL-SIEM for securing the cybersecurity threats.

5 Conclusion

With the deployment of all the components of the Security Framework, all the communications and the information managed and shared among the different stakeholders of the supply chain are guaranteed in terms of:

- Security
- Confidentiality
- Integrity
- Availability of the managed information

Furthermore, added value features such as protection against cyber-attacks and a complete authorization, authentication and reputation model management, ensure a high level of security and protection.

6 Annex 1 – Keycloak management

6.1 Admin console

The admin console is the core component of the management of the authentication service. The way to access is via URL, to the ATOS' server in which is deployed:

https://auth.composition-ecosystem.eu/keycloak/

So we enter to the welcome page:

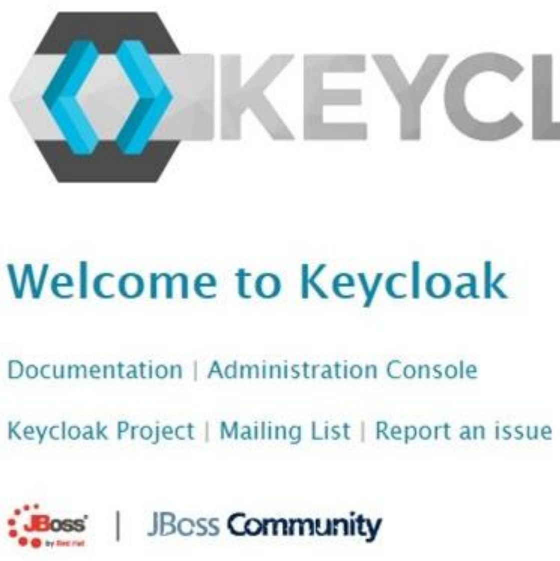

KEYCLOAK
Welcome to Keycloak
Documentation | Administration Console
Keycloak Project | Mailing List | Report an issue
JBoss | JBoss Community

Figure 14 Keycloak welcome page

So, clicking in the "Administration Console" link we proceed to enter our credentials in the login page:

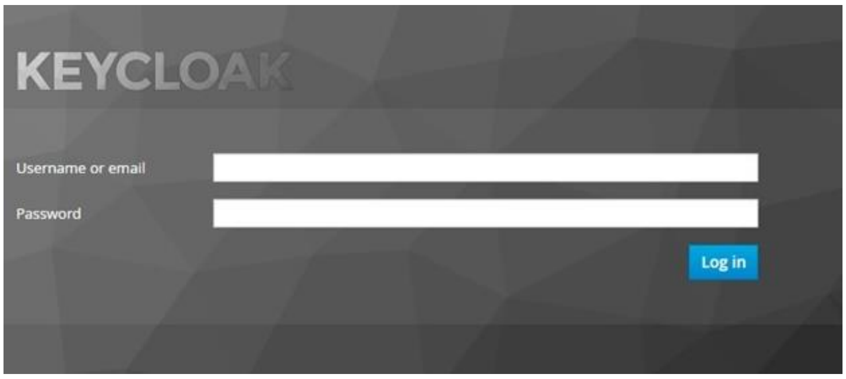
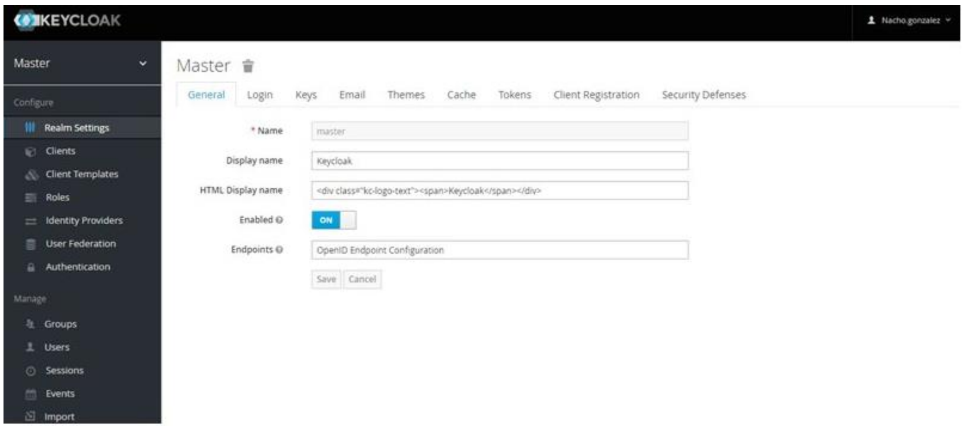


Figure 15 Keycloak administration console login page

Once we enter the admin user and password, it'll bring us to the Keycloak Admin Console:



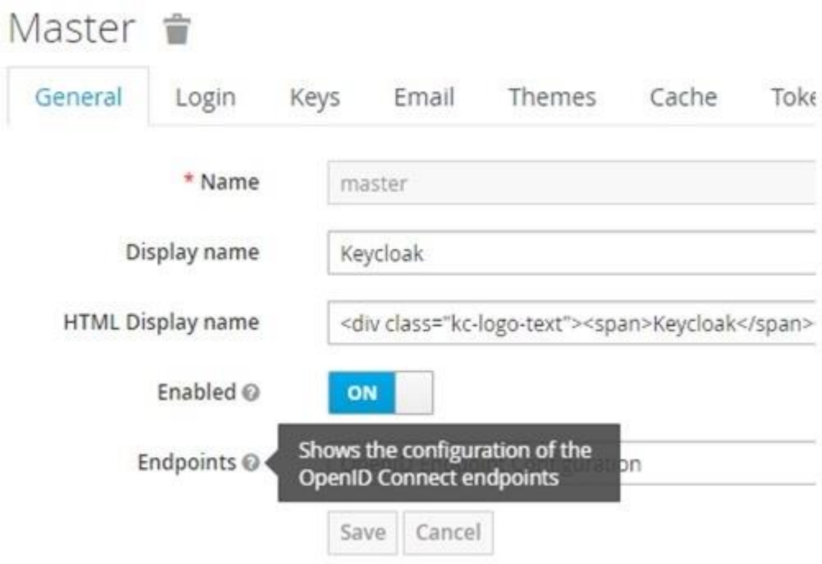
* Name: master
Display name: Keycloak
HTML Display name: <div class="kc-logo-text"><span>Keycloak</span></div>
Enabled: ON
Endpoints: OpenID Endpoint Configuration

Figure 16 Keycloak administration console

The structure of this page is quite simple but functional. There is one menu on the left, where we can manage realms (by default it manages "Master" realm).

The menu on the right, just dropping off from the user name allows us to view our user account or logout.

Furthermore, there is a help system embedded. If we're not sure about any certain field, we can hover our mouse over any question mark icon, and a tooltip text will appear providing more information about this field.



* Name: master
Display name: Keycloak
HTML Display name: <div class="kc-logo-text"><span>Keycloak</span>
Enabled: ON
Endpoints: Shows the configuration of the OpenID Connect endpoints

Figure 17 Keycloak realm management

6.2 The Master realm

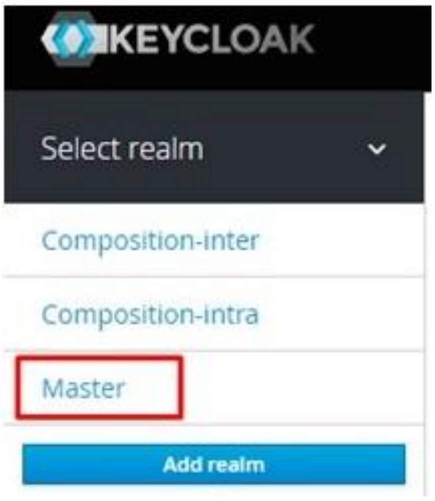


Figure 18 Keycloak master realm

The Master realm is the default realm that Keycloak creates when is booted for the first time. Is in the highest level in the hierarchy tree of the realms, and the admin accounts enclosed in this realm have permissions to view and manage any other realm created on the server.

The use of the Master realm is recommended for super admins for creation and management of other business realms in the system. This security model prevents accidental changes and follows the tradition of permitting user accounts to only those privileges and powers necessary for the successful completion of their current task.

6.3 Creating a new realm

For creating a new real, we've to click in the "Add realm" button that appears in the real drop-down menu on the left:

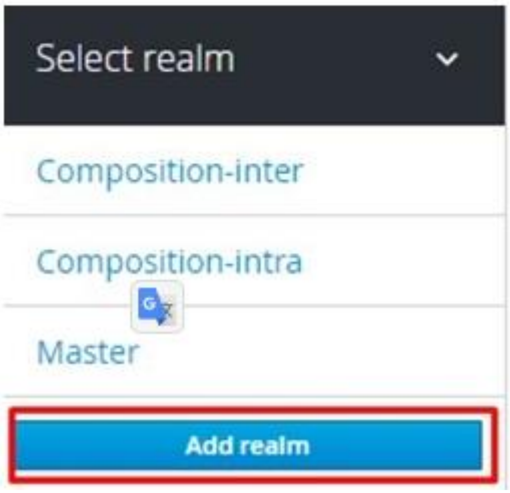


Figure 19 Keycloak add realm

Then we complete the information in the Add Realm page. The information we have to provide here is only the realm name. In other way, we can import a JSON document that defines the new realm.

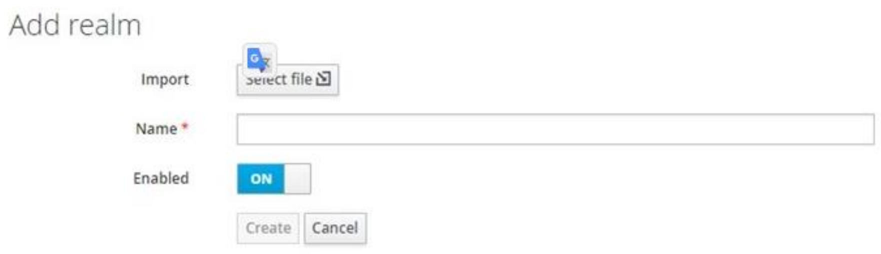


Figure 20 Keycloak add realm details

6.4 User search

For searching a specific user, we just have to click in the "Users" menu option. A "Lookup" tab will appear and we can introduce the search information in the textbox:

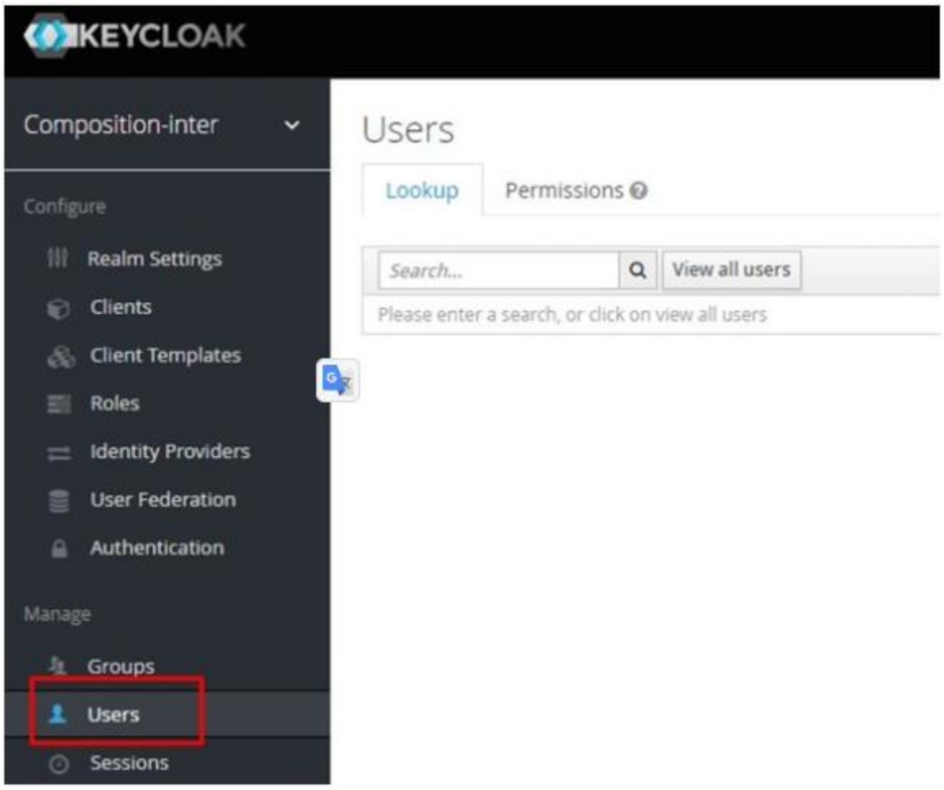


Figure 21 Keycloak user search

6.5 Create new user

In the “Users” page (same page for the search user option), we can click on the “Add user” button on the right for creating a new user:

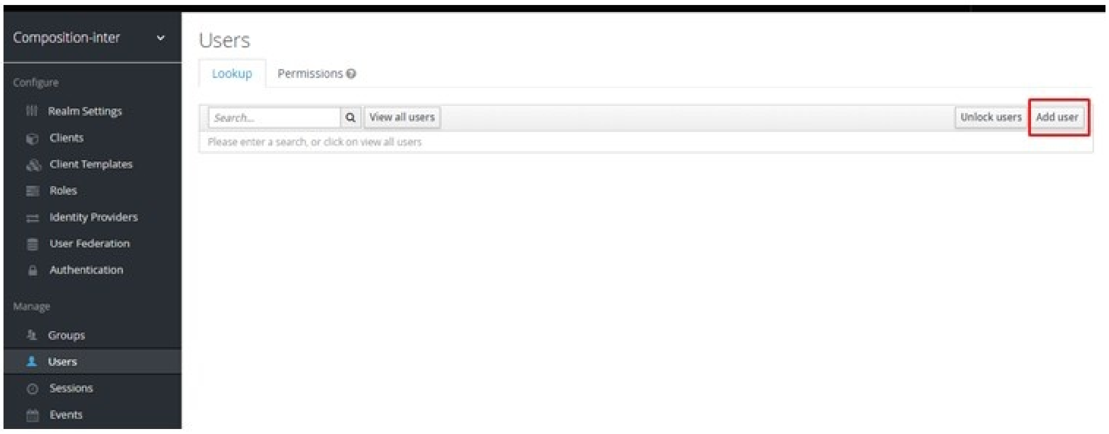
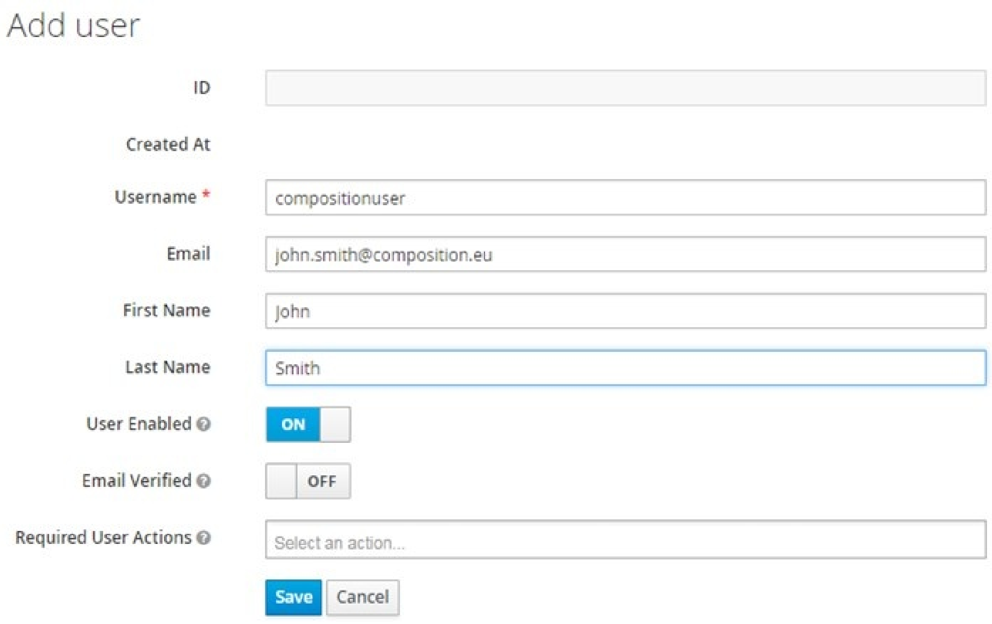


Figure 22 Keycloak new user

So, we enter the page for entering the information of the new user:

- ID
- User name (only field required)
- Email
- First name
- Last name
- User enabled
- Email verified
- Required user actions

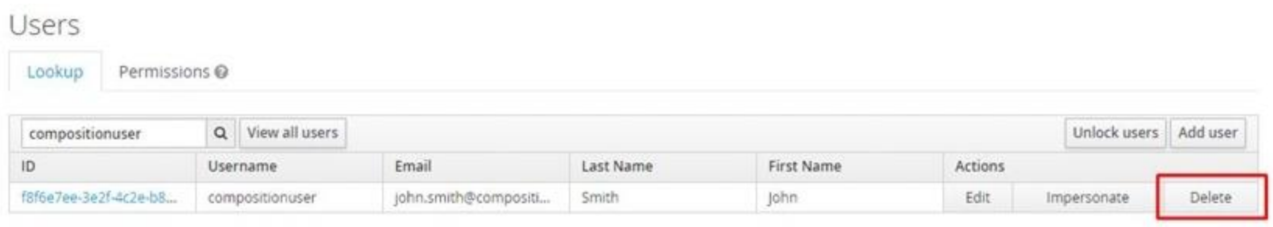


ID	
Created At	
Username *	compositionuser
Email	john.smith@composition.eu
First Name	John
Last Name	Smith
User Enabled	ON
Email Verified	OFF
Required User Actions	Select an action...

Figure 23 Keycloak new user details

6.6 Deleting users

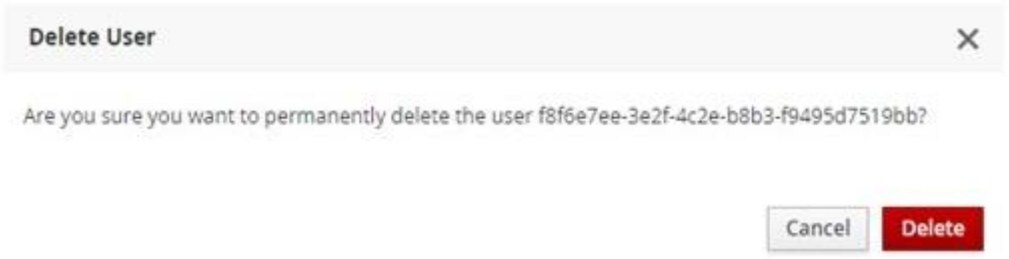
For deleting one user, we'll have to search first as described in section 4.1.2.1 the user we want to delete. For example, we're going to delete the user with username "compositionuser". Once we find the user, we can click on the "Delete" button on the right side of the row:



ID	Username	Email	Last Name	First Name	Actions		
f8f6e7ee-3e2f-4c2e-b8...	compositionuser	john.smith@compositi...	Smith	John	Edit	Impersonate	Delete

Figure 24 Keycloak delete user

We'll be asked for our confirmation:



Are you sure you want to permanently delete the user f8f6e7ee-3e2f-4c2e-b8b3-f9495d7519bb?

Figure 25 Keycloak delete user confirmation

6.7 User configuration

When we enter into a user profile, we access to a menu where we can edit different options of the user:

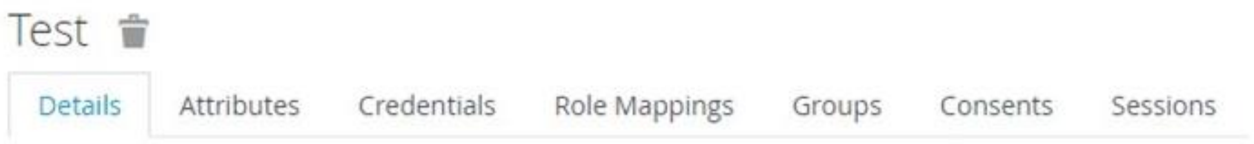
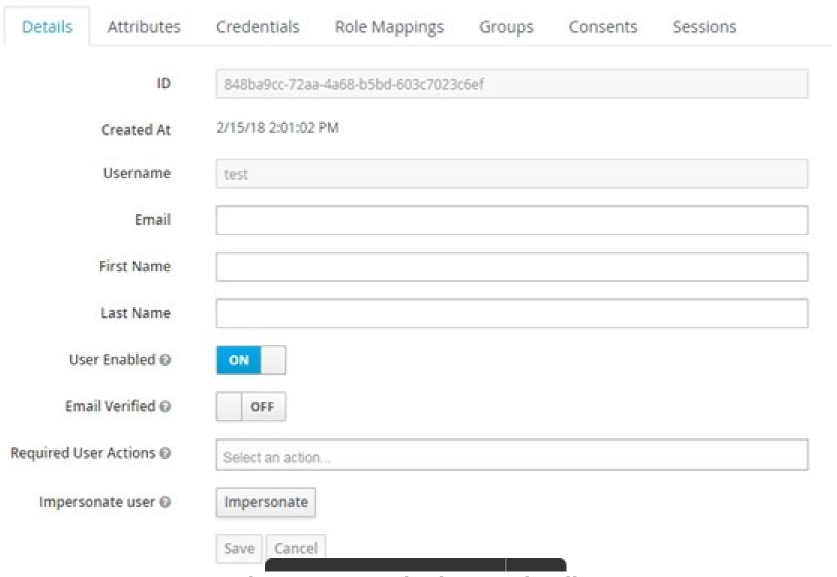


Figure 26 Keycloak user configuration

6.7.1 Details

Here we can modify some general options of the user, basically the same we specified in the creation process of the user (email, first name, last name…)

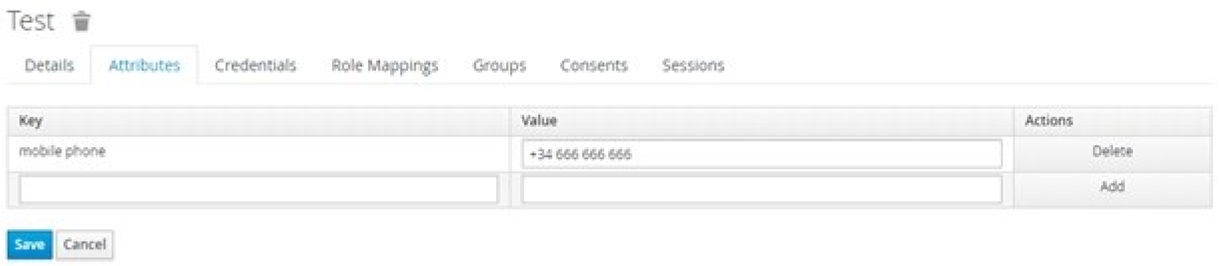


ID	848ba9cc-72aa-4a68-b5bd-603c7023c6ef
Created At	2/15/18 2:01:02 PM
Username	test
Email	
First Name	
Last Name	
User Enabled	ON
Email Verified	OFF
Required User Actions	Select an action...
Impersonate user	Impersonate

Figure 27 Keycloak user details

6.7.2 Attributes

In this tab, we can manage specific arbitrary user attributes:

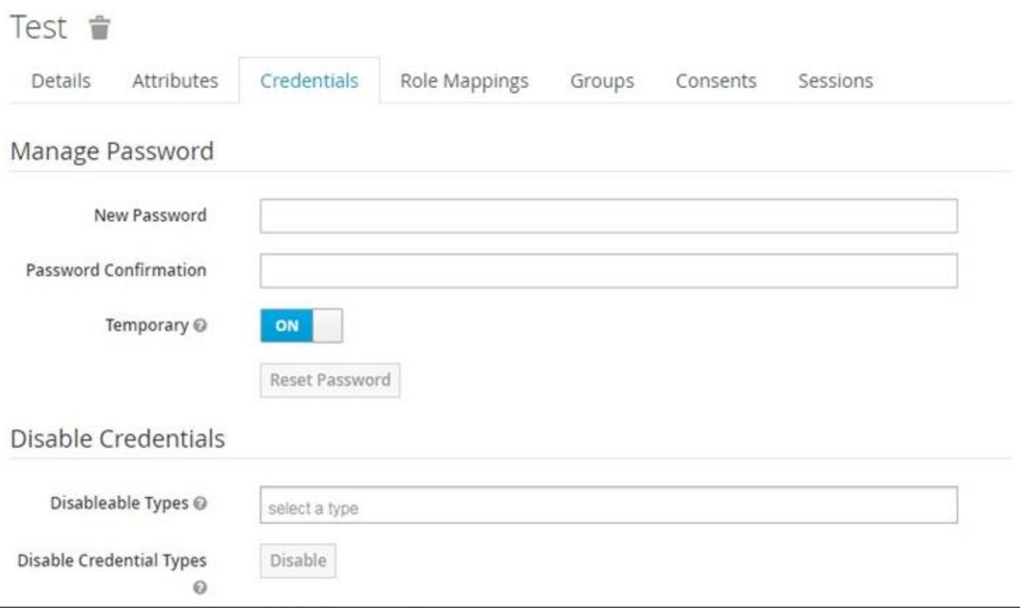


Key	Value	Actions
mobile phone	+34 666 666 666	Delete
		Add

Figure 28 Keycloak user attributes

6.7.3 Credentials

In this tab, we can manage the passwords and the OTP of the users:



Test

Details | Attributes | Credentials | Role Mappings | Groups | Consents | Sessions

Manage Password

New Password:

Password Confirmation:

Temporary: ON

Reset Password

Disable Credentials

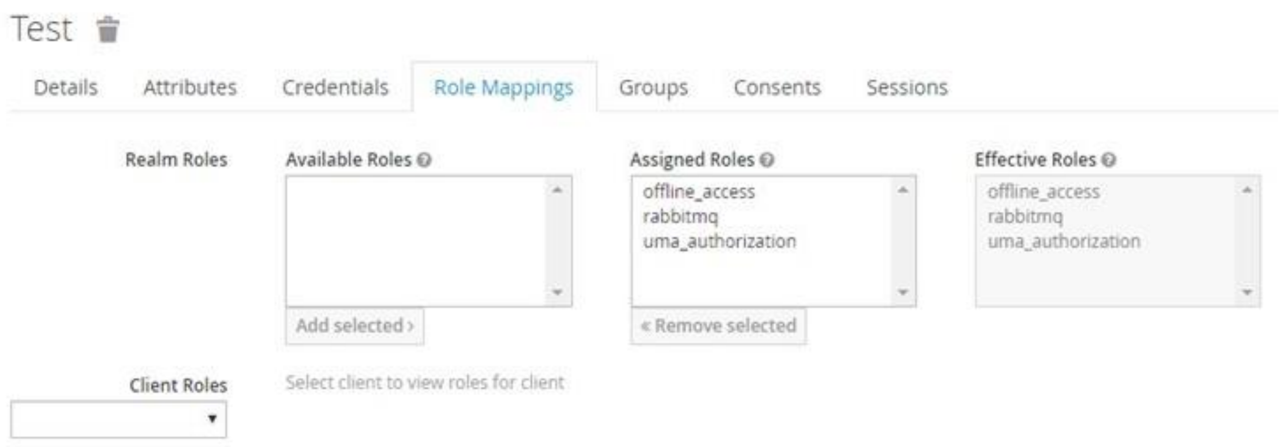
Disableable Types: select a type

Disable Credential Types: Disable

Figure 29 Keycloak user credentials

6.7.4 Role mappings

Here we can manage the different roles (a role is a predefined group of actions and behaviours to be applied to a set of users). Due to organisational and business requirements, sometimes it'll be helpful to group different users into a specific role:



Test

Details | Attributes | Credentials | Role Mappings | Groups | Consents | Sessions

Realm Roles

Available Roles: (empty) — Add selected ›

Assigned Roles: offline_access, rabbitmq, uma_authorization — « Remove selected

Effective Roles: offline_access, rabbitmq, uma_authorization

Client Roles

Select client to view roles for client

Figure 30 Keycloak user role mappings

6.7.5 Groups

In this tab we can manage the grouping of users:

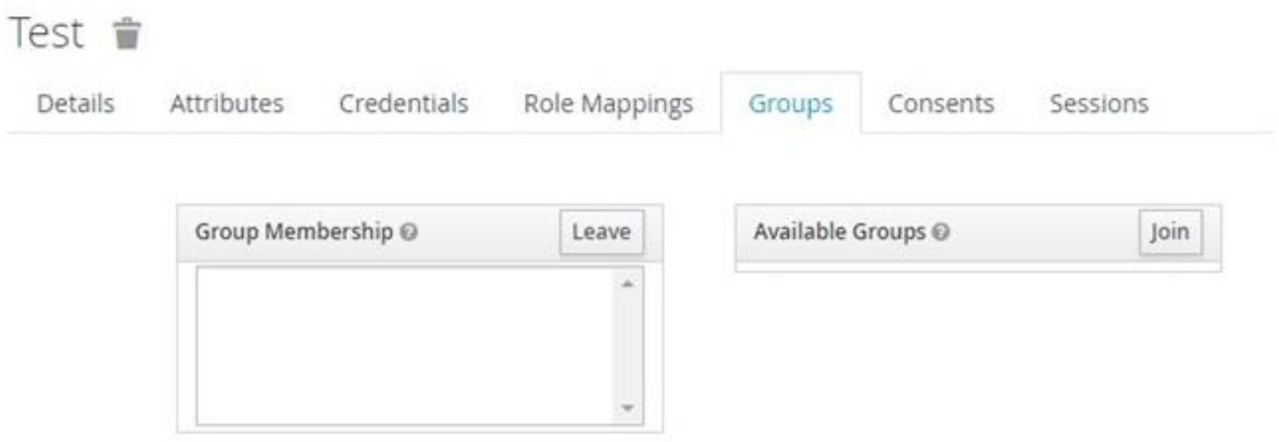


Figure 31 Keycloak user groups

7 Annex II – RabbitMQ configuration parameters

```
[
        { rabbit, [
                { loopback_users, [ ] },
                { tcp_listeners, [ 8182 ] },
                { ssl_listeners, [ ] },
                { default_pass, <<"xxxxxxxxxxxxxx">> },
                { default_user, <<"xxxxxxxxxxxxxx">> },
                { hipe_compile, false },
                {auth_backends, [rabbit_auth_backend_cache, rabbit_auth_backend_internal]}
        ] },
        { rabbitmq_management, [ { listener, [
                { port, 8181 },
                { ssl, false }
        ] } ] },
         {rabbitmq_auth_backend_cache, [
                 {cached_backend, rabbit_auth_backend_http}
         ] },
        {rabbitmq_auth_backend_http, [
                 {http_method,   post},
                {user_path,     "https://auth.composition-ecosystem.eu/raas/raas-rest/auth/user"},
                 {vhost_path,    "https://auth.composition-ecosystem.eu/raas/raas-rest/auth/vhost"},
                 {resource_path,                    "https://auth.composition-ecosystem.eu/raas/raas-
rest/auth/resource"},
                 {topic_path,    "https://auth.composition-ecosystem.eu/raas/raas-rest/auth/topic"}
        ]},
        {rabbitmq_mqtt, [
                {allow_anonymous,  false},
                 {vhost,            <<"/">>},
                 {exchange,         <<"amq.topic">>},
                 {subscription_ttl, 1800000},
                 {prefetch,         10},
                 {ssl_listeners,    []},
                 {tcp_listeners,    [8183]},
                 {tcp_listen_options, [{backlog,   128},
                                       {nodelay,   true}]}
        ]},
         {rabbitmq_web_mqtt, [
                 {tcp_config, [{port, 8184}]},
                 {cowboy_opts, [{idle_timeout, 180000}]}
         ]}
].
```

8 List of Figures and Tables

8.1 Figures

8.2 Tables

9 References

Gustavo González-Granadillo, M.F., S. G.-Z. (2018). *Towards an Enhanced Security Data Analytic Platform.* Atos Research and Innovation, Cyber Security Department.

(COMPOSITION D4.2, 2018) D4.2 Design of Security Framework II

(COMPOSITION D4.4, 2018) D4.4 Prototype of the Security Framework I

(COMPOSITION D6.3, 2018) D6.3 COMPOSITION Marketplace I