eIDAS-Node National IdP and SP Integration Guide V2.6

eIDAS eID Implementation

Exported on 04/15/2022

Table of Contents

Document history

Version	Date	Modification reason	Modified by
1.0	16/10/2017	Origination	DIGIT
2.0	10/04/2018	Document rewritten particularly in relation to separation of Generic and Specific Parts.	DIGIT
2.0.1	19/04/2018	Correction to Figure captioning EID-609. Corrected example of LightToken generation section 4.4.1 EID-611.	DIGIT
2.1	09/07/2018	Reuse of document policy updated and version changed to match the corresponding Release. Document describes how to migrate to eIDAS-Node v2.1.	DIGIT
2.3	20/06/2019	Adapted text and figures to new Cache and Ignite.	DIGIT
2.4	06/12/2019	Minor update.	DIGIT
2.5	11/12/2020	eIDAS-Node 2.5 release	DIGIT
2.6	04/2022	eIDAS-Node 2.6 release	DIGIT

Disclaimer

This document is for informational purposes only and the Commission cannot be held responsible for any use which may be made of the information contained therein. References to legal acts or documentation of the European Union (EU) cannot be perceived as amending legislation in force or other EU documentation.

The document contains a brief overview of technical nature and is not supplementing or amending terms and conditions of any procurement procedure; therefore, no compensation claim can be based on the contents of the present document.

© European Union, 2022
Reuse of this document is authorised provided the source is acknowledged. The Commission's reuse policy is implemented by Commission Decision 2011/833/EU of 12 December 2011 on the reuse of Commission documents

List of abbreviations

The following abbreviations are used within this document.

Abbreviation	Meaning
eIDAS	electronic Identification and Signature. The Regulation (EU) N°910/2014[1] governs electronic identification and trust services for electronic transactions in the internal market to enable secure and seamless electronic interactions between businesses, citizens and public authorities.
IAM	Identity and Access Management.
IdP	Identity Provider. An institution that verifies the citizen's identity and issues an electronic ID.
LoA	Level of Assurance (LoA) is a term used to describe the degree of certainty that an individual is who they say they are at the time they present a digital credential.
MW	Middle Ware. Architecture of the integration of eIDs in services, with a direct communication between SP and the citizen's PC without any central server. The term also refers to the piece of software of this architecture that executes on the citizen's PC.
MS	Member State
SAML	Security Assertion Markup Language
SAT	Solution Architecture Template
SP	Service Provider

List of definitions

The following definitions are used within this document.

Term	Meaning
Basic Setup	The basic configuration and Demo tools provided in a package to setup and run an eIDAS-Node strictly for demo purposes only.
Demo tools	Demo tools comprise the Demo SP and Demo IDP included in the package. These components are not production ready and should not be deployed or used in production environments.

1 http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2014.257.01.0073.01.ENG

Term	Meaning
eIDAS-Node	An eIDAS-Node is an application component that can assume two different roles depending on the origin of a received request. See eIDAS-Node Connector and eIDAS-Node Proxy Service.
eIDAS-Node Connector	The eIDAS-Node assumes this role when it is located in the **Service Provider's** Member State. In a scenario with a Service Provider asking for authentication, the eIDAS-Node Connector receives the authentication request from the Service Provider and forwards it to the eIDAS-Node of the citizen's country. This was formerly known as S-PEPS.
eIDAS-Node Proxy Service	The eIDAS-Node assumes this role when it is located in the **citizen's** Member State. The eIDAS-Node Proxy Service receives authentication requests from an eIDAS-Node of another MS (their eIDAS-Node Connector). The eIDAS-Node Proxy-Service also has an interface with the national eID infrastructure and triggers the identification and authentication for a citizen at an identity and/or attribute provider. This was formerly known as C-PEPS.

1 Introduction

This document is intended for a technical audience consisting of architects, developers, and those requiring detailed technical information on how an eIDAS-Node can be integrated into the National eID infrastructure.

1.1 Purpose

The purpose of this document is to describe how Member States can integrate the eIDAS-Node into their national infrastructure, which can be done in a number of ways.

This document provides guidance by recommending one way in which it can be done.

1.2 Document aims

The aims of this document are to:

- provide details of how to develop and tailor the parts that are specific to your country (Specific parts);
- describe implementation into web-based infrastructure; an
- provide information on the Protocol Engine architecture which is at the heart of all protocol related operations in the eIDAS-Node.

1.3 Document structure

This document is divided into the following sections:

- Chapter 1 – *Introduction* this section.
- Chapter 2 – *The eID Building Block* describes interoperability aspects of eID.
- Chapter 3 – *Designing Integration* describes considerations of the eIDAS-Node physical architecture to be taken into account when developing your integration strategy.
- Chapter 4 – *Integration Possibilities* describes the recommended integration approaches, starting with that requiring the least changes.
- Appendix A – *Diagrams and Schemas* contains diagrams covering some parts of the software architecture that are mentioned in this document.
- Appendix B – *Examples* for the: LightToken QED, Python's Ignite Thin client Specific Connector POC and Ignite's Rest API .
- Appendix C – *Ignite Advanced Configurations* elaborates on *Ignite advanced configurations*, e.g. TLS configuration.
- Appendix D – *Ignite Proposed Configuration* recommended default configuration for ignite.
- Appendix E – Message Logging features

1.4 Other technical reference documentation

We recommend that you also familiarise yourself with the following eID technical reference documents (as appropriate to your role) which are available on **Digital Home > eID :**

- *eIDAS-Node Installation, Configuration and Integration Quick Start Guide* describes how to quickly install a Service Provider, eIDAS-Node Connector, eIDAS-Node Proxy Service and IdP from the distributions in the release package. The distributions provide preconfigured eIDAS-Node modules for running on each of the supported application servers.
- *eIDAS-Node Installation and Configuration Guide* describes the steps involved when implementing a Basic Setup and goes on to provide detailed information required for customisation and deployment.

- *eIDAS-Node Demo Tools Installation and Configuration Guide* describes the installation and configuration settings for Demo Tools (SP and IdP) supplied with the package for basic testing.
- *eIDAS-Node and SAML* describes the W3C recommendations and how SAML XML encryption is implemented and integrated in eID. Encryption of the sensitive data carried in SAML 2.0 Requests and Assertions is discussed alongside the use of AEAD algorithms as essential building blocks.
- *eIDAS-Node Error and Event Logging* provides information on the eID implementation of error and event logging as a building block for generating an audit trail of activity on the eIDAS Network. It describes the files that are generated, the file format, the components that are monitored and the events that are recorded.
- *eIDAS-Node Security Considerations* describes the security considerations that should be taken into account when implementing and operating your eIDAS-Node scheme.
- *eIDAS-Node Error Codes* contains tables showing the error codes that could be generated by components along with a description of the error, specific behaviour and, where relevant, possible operator actions to remedy the error.

Other useful resources on eID, eIDAS and how the eID Building Block can be used while preserving all the characteristics and qualities that makes it a Building Block (e.g.: sustainability, reusability and replaceable parts):

- **What is eID?** helps understanding of the need and purpose of eID — https://ec.europa.eu/digital-building-blocks/wikis/display/DIGITAL/What+is+eID).
- **SAT for eID** and supporting links will help to understand the architectural framework that supports the eID Building Block — https://joinup.ec.europa.eu/release/eid-sat/v101.

Disclaimer: The users of the eIDAS-Node sample implementation remain fully responsible for its integration with back-end systems (Service Providers and Identity Providers), testing, deployment and operation. The support and maintenance of the sample implementation, as well as any other auxiliary services, are provided by the European Commission according to the terms defined in the European Union Public License (EUPL) at https://joinup.ec.europa.eu/sites/default/files/custom-page/attachment/eupl_v1.2_en.pdf

2 The eID Building Block

The eID Building Block helps public administrations and private online service providers to extend the use of their online services to citizens from other EU Member States by enabling **cross-border authentication**, in a secure, reliable and trusted way, by making national electronic identification systems interoperable.

Once this Building Block is deployed in a Member State, the mutual recognition of national eIDs becomes possible between participating Member States, in line with the eIDAS (electronic Identification and Signature) legal framework and with the privacy requirements of all the participating countries. Mutual recognition of national eIDs allows citizens of one Member State to access online services provided by public and private organisations of other participating Member States, using their own national eID.

The advantages of adopting this approach are:

- Sustainability;
- Greater Security;
- Better Scalability; and
- More Flexibility.

The eID Building Block is primarily intended for authentication and is not intended for authorisation, document transfers, etc.

2.1 eIDAS-Node software

The eIDAS-Node software is a sample implementation of the eID eIDAS Profile[2]. It is developed by the European Commission with the help of Member States collaborating in the technical sub-committee of the eIDAS Expert Group. The eIDAS-Node software contains the necessary modules to help Member States to communicate with other eIDAS-compliant counterparts in a centralised or distributed fashion.

The sample implementation is composed of the following:

- **eIDAS-Node**: an implementation of the eID eIDAS Profile able to communicate with other nodes of the eIDAS Network. The eIDAS-Node can either request (via an eIDAS-Node Connector) or provide (via an eIDAS-Node Proxy Service) cross-border authentication;
- **Testing tools (demo Service Provider and demo Identity Provider)**: additional tools for setting up a demo environment for testing purposes.
  Note: The testing tools should be used for testing purposes only and should not be deployed in a production environment.

Each eIDAS-Node Connector and eIDAS-Node Proxy Service consists of two parts:

- Generic part; and
- Specific part.

Generic part
A sample implementation of the Generic part, conforming to the eIDAS Technical Specifications, is developed by DIGIT. Code in the Generic part (modules eIDAS-Node, eIDAS-Metadata, eIDAS-SAMLEngine) is developed to communicate with other eIDAS-Nodes in other Member States (Connectors and Proxy Services) using the strictly defined eIDAS protocol. This ensures compatibility and interoperability between eIDAS-Nodes in the eIDAS network.

Specific part
The Specific part is **designed and implemented by individual Member States** to suit their specific national requirements. This custom Specific part communicates with the Generic part by defined interfaces.

2 https://ec.europa.eu/digital-building-blocks/wikis/display/CEFDIGITAL/eIDAS+eID+Profile

This separation of functionality between the Generic part and the Specific part defines a clear domain boundary and provides independence where, for example, Service Providers are not aware of the technology used in cross-border transactions.

Because of this separation, some functions that are required by the Interoperability Specifications must be implemented by Member States in their Specific part (or in the Member State 'Hub', see Figure 1), these are:

- MS Selection provided by the Connector solution; and
- Scheme selection provided by the Proxy Service solution.

Note: The integration package includes a sample Specific part for demonstration only. This is not production-ready and should not be seen as a template for how functionality should be implemented.

Disclaimer: The users of the eIDAS-Node sample implementation remain fully responsible for its integration with back-end systems (Service Providers and Identity Providers), testing, deployment and operation. The support and maintenance of the sample implementation, as well as any other auxiliary services, are provided by the European Commission according to the terms defined in the European Union Public Licence[3] (EUPL).

2.2 Architecture of a typical eID solution

The following diagram, based on eID Solution Architecture Template (SAT 1.0.1), illustrates the application level approach for a typical eID/authentication solution.

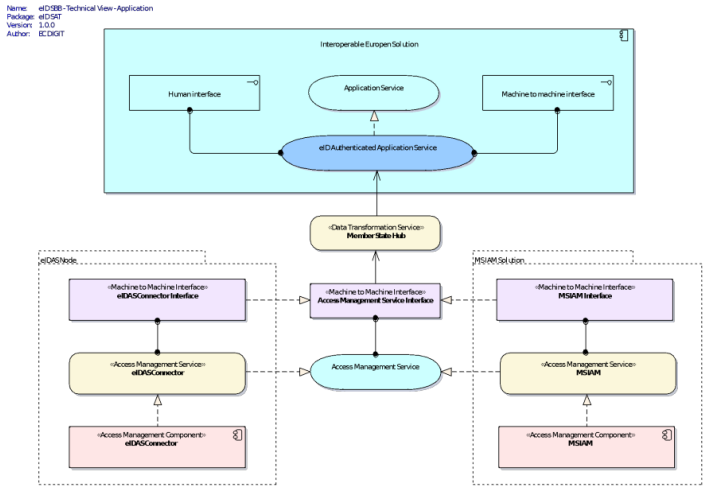

Figure 1: Architecture of a typical eID solution with MS IAM

This is a very high-level ArchiMate® abstract, introducing only a concept, and not the physical level solution.

The packaging component (Interoperable European Solution) implements an Application Service using eID Authentication, therefore realising an eID Authenticated Application Service.

For example, opening a bank account requires identification of the person, which can be done with eID-based authentication. For the above architecture, strictly for the authentication (so not including any other processes like Anti-Money-Laundering) it is indifferent to whether the eID is local or foreign. The solution (like the online bank account) uses the local authentication ecosystem offered by eGovernment services.

This is the Member State Hub responsible for hiding the cross-border nature, but making the necessary conversion and routing between the two different implementations: eIDAS-Node Connector and (the existing) government

3 https://ec.europa.eu/digital-building-blocks/wikis/download/attachments/46992716/eupl1.1.-licence-en.pdf?api=v2&modificationDate=1496243904284&version=1

Identity and Access Management (IAM) services. The Member State Hub is abstract, it is not necessarily a physical component, it is a service (composed of functions like the conversion and routing), that can be centralised or decentralised.

The important message from this diagram is the concept that eIDAS cross-border authentication is an alternative to the locally provided IAM services.

Reducing the solution to a cross-border-only service scenario results in this diagram:

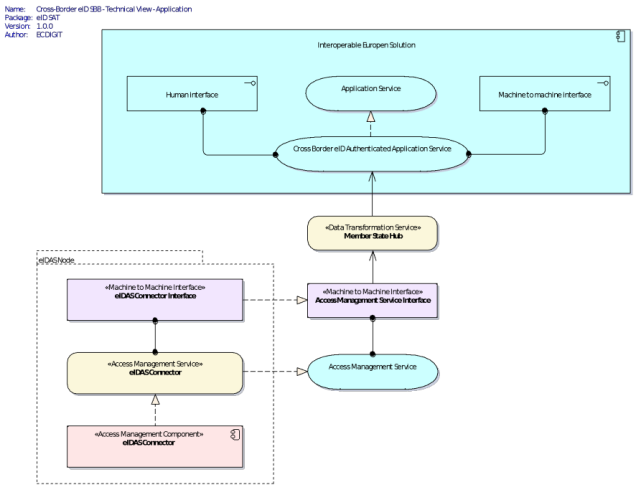

Figure 2: Architecture of a cross-border-only service

3 Designing Integration

There are multiple ways the eIDAS-Node can be integrated with a national network. Before going into detail, there is a need to understand the architecture of the system and set the integration strategy accordingly. Therefore, this section describes the general architecture considerations.

3.1 eIDAS-Node Connector and eIDAS-Node Proxy Service in one application instance

The delivered or custom built EidasNode.war web application contains the functionality of both eIDAS-Node Connector and eIDAS-Node Proxy Service. The roles are activated by configuration. It is possible to have both roles activated in one application instance (as shown in the Basic Setup) though this is not recommended and it is better to have two different instances for the following main reasons:

- the roles are very different, and since the eIDAS-Node Proxy Service issues identities, the security level is higher;
- besides the security level, uptime and business continuity requirements will be different, especially if bilaterally agreed; and
- there can be, and most likely there will be, multiple eIDAS-Node Connectors in the future for different purposes and sectors.

3.2 Deployment models

There are two deployment methods supported:

- Standard deployment; and
- Monolithic deployment.

3.2.1 Standard Deployment method

The Standard Deployment is the recommended approach for most Member States because it offers more flexibility. This is the default build target for the provided MAVEN Project Object Model.

Using the Standard Deployment the eIDAS-Node is built as a separate, independent, ready-to-use web application, that can be deployed directly. In this deployment model the Specific part may not exist at all, as the Node can be integrated closely to a domestic identity management infrastructure. There are several advantages of choosing this method:

- It is platform independent, using the provided Ignite technology stack (an alternative to Ignite can be used, see section 4.2.6 — *Back-end communication with alternative to* although reconfiguration and repackaging will be required). Integration with C++, .NET or custom infrastructure is also possible.
- Seamless upgrade to new versions. Upgrade of minor eIDAS-Node versions, where the interface does not change does not affect MS' Specific or infrastructure. Compatibility with major version change may also be better, just needing to adjust the background data communication.
- Impact of Java library dependencies is less in case of reusing some of the provided and supported libraries (like EIDAS-Commons), and may not exist at all if integrating with XML directly.

3.2.2 Monolithic Deployment method

The Monolithic Deployment is similar to the architecture delivered in eIDAS Node v1.4 and earlier versions. The Specific application parts (Specific Connector and Specific Proxy Service) are built as part of the application, as included JAR files. However, the communication interface is the same as in the Standard Deployment. By default the Monolithic Deployment method results in a non-distributed application at the communication interface level, as the communication cache does not default to a shared map, it uses internal memory cache. However, it can be changed by editing the Maven build profile.

The Monolithic Deployment method is offered for testing and familiarisation but can be useful for those who plan to operate the eIDAS-Node as one standalone application, and accept the following restrictions:

- All Java libraries included in the Generic part are also dependencies of the Specific, so upgrade or replacement possibilities are very limited.
- Any change in these commonly used libraries introduced in future eIDAS releases forces the implementer to do an impact analysis, and to take the necessary actions before upgrading.
- Overall performance of the application is dependent on Specific implementation, therefore eID support possibilities may be limited.
- Troubleshooting and problem solving by eID Team is limited. All incidents must be investigated locally to make sure, only those issues are being escalated to eID support, that originate from the provided software, and not the customised or added parts.

3.3 Required profile and flows

The eIDAS protocol is based on SAML2, implemented with a web profile, so whatever is used in the national infrastructure, the eIDAS-Node will require an HTTP request, and will do Redirects or form Posts containing SAML messages. It requires a client browser capable of understanding HTTP, but does not require cookies or JavaScript support by default.

Control and Data flows are decoupled. The control is done by the user agent (browser) on the public Internet, while the data is happening in the background between back-end interfaces.

3.3.1 High level flow

The following diagram demonstrates the high level flow during an originated Request (Receiving MS) scenario, assuming there is a Specific Connector (can be anything in the MS infrastructure, including a Hub), and the eID provided SpecificCommunication API is used for the background communication.

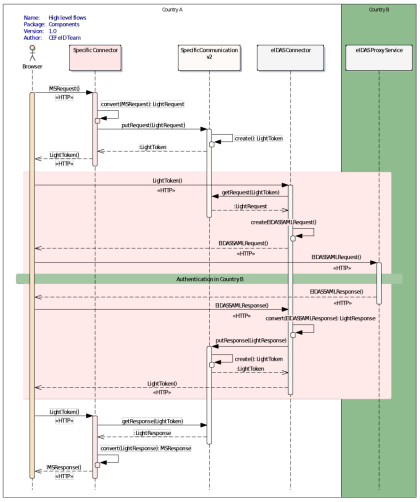

Figure 3: High level flows

The scope is restricted to the invocation of cross-border eID authentication service. The highlighted area contains the flows described in the next sections.

3.3.1.1 Outbound from MS

This invocation pattern restricts the requester to the use of HTTP. To invoke operation on the eIDAS Network (both issuing and consuming identities), the eIDAS-Node will require a browser hit (HTTP GET or POST) from an endpoint located in the national infrastructure.
The HTTP request needs to contain a reference token to a data package that is exchanged in the background. See section 4.4.1 — *Implementing the LightToken* for details on this token.

The incoming HTTP request is held, while processing is performed (synchronous way). The token-referenced data package (LightRequest/LightResponse) will be retrieved by the Node and an eIDAS SAML Request/Response is created based on the provided information. This SAML message is placed into the HTTP response to the corresponding HTTP request coming with the initial control flow. The user agent is redirected to the intended eIDAS Node (Proxy-Service) in the Network.

If it is a Request, originated from the MS, the Node will cache the Request in order to correlate with the incoming Response.

If it is a Response, then the Node will recover the original Request from its internal cache, to do the correlation.

3.3.1.2 Inbound to MS

When the eIDAS Network initiates an invocation (either response or request), the Node receives and processes the eIDAS SAML message. The incoming user agent HTTP request is held.

The resulting LightRequest/LightResponse is placed on the background communication bus, and a referencing token is created and written to the HTTP response of the initial user agent HTTP request. Then the agent is redirected to a configuration specified URL, that is interpreted as the URL of the MS-Specific.

If the incoming eIDAS message is a Response, the original Request is recovered for correlation from the internal cache.

If the incoming eIDAS message is a Request, it is stored to be recovered from the cache when the MS-based authentication is done, and the control flow returns with the response.

3.4 Requirements from MS

As seen in section 3.3 — *Required profile and flows* the eIDAS-Node requires the MS implementation to handle Service Provider requests to:

- supply a user-agent redirect to Connector Node with a Request referencing token;
- push the referenced Request data object via back-end communication to the Connector Node;
- supply an HTTP endpoint where the eIDAS Node redirects the user-agent with a Response referencing token; and
- pull the referenced Response data object from the Connector Node via background communication.

The pattern is the same for implementing cross-border identity issuing in the Proxy Service. Here the MS is responsible for:

- supplying an HTTP endpoint where the eIDAS Node redirects the user-agent with a Request referencing token;
- pulling the referenced Request data object from the Proxy Service Node via background communication;
- supplying a user-agent redirect to the Proxy Service Node with a Response referencing token; and
- pushing the referenced Response data object via back-end communication to the Proxy Service Node.

The technical details and possibilities are described in the following section.

4 Integration Possibilities

This section describes the recommended integration approaches, starting with the most straightforward.

4.1 Provided stack

The eIDAS-Node includes a sample, demo implementation of a Member State Specific Part, that is designed to help the understanding of the recommended integration pattern. The stack itself is customisable by implementing the provided/required interfaces.

The following diagram shows the complete delivery structure (including sample modules) with dependencies. Note that the modules shown in red are labelled 'DO NOT USE' in the legend, this means use only as samples for demonstration purposes, do not use in production.

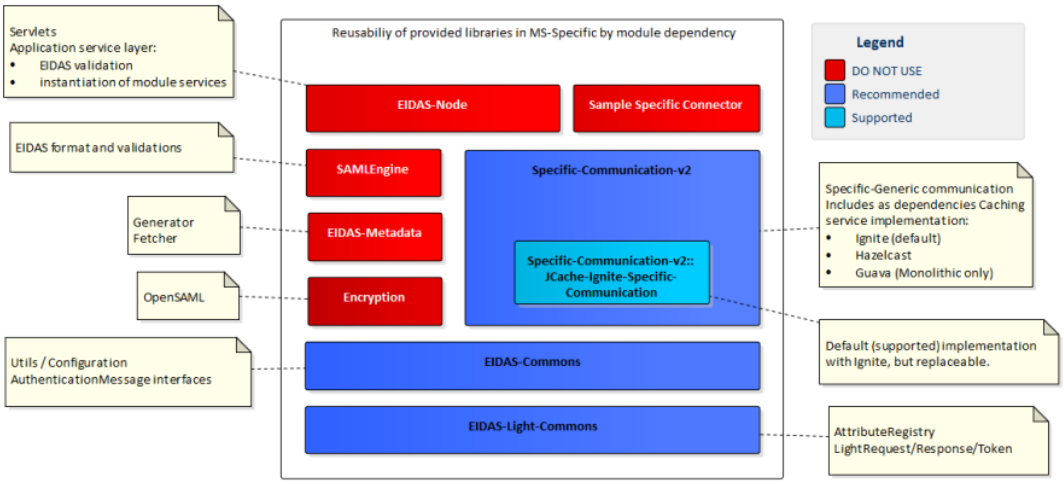

Figure 4: Delivery structure with dependencies

The dependencies can be seen starting from the top; everything that is below the component is a dependency, i.e. the eIDAS-Node is dependent on the full stack, the EIDAS-Metadata is only dependent on Encryption, EIDAS-Commons and EIDAS-Light-Commons.

The recommended stack shown in blue is for Java developers, and is realised mostly in Specific-Communication-v2.0. This module provides all the necessary functions to communicate with an eIDAS-Node. The provided Sample Specifics (Sample Specific Connector and Specific Proxy Service) demonstrate how to use this API. It is also detailed in the following chapters. The API can be redefined, in which case it must also be done in the Generic part.

The Jcache-Ignite-Specific-Communication (light blue) is supported and provided. However it is also customisable, any shared map/cache/distributed database product can be used, that fulfils certain technical requirements set by the Member State, and can be integrated with the eIDAS-Node software. For example enterprise WebLogic users may favour use of Coherence, but Redis or Memcached could equally be alternatives. This option is detailed in section 4.2.6 — Back-end communication with alternative to Ignite.

4.2 Using the provided SpecificCommunication API

The provided EIDAS-*SpecificCommunicationDefinition* module offers tools for developing the control and data flows. To use it, the Specific part must import the modules:

- EIDAS-Light-Commons; and

- EIDAS-Commons.

from the source delivery, or the binary built JAR files. Therefore it must be a Java application built for a supported JVM version (listed in the eIDAS-Node Installation and Configuration Guide).

For a Standard Deployment, a possible component layout is shown below:

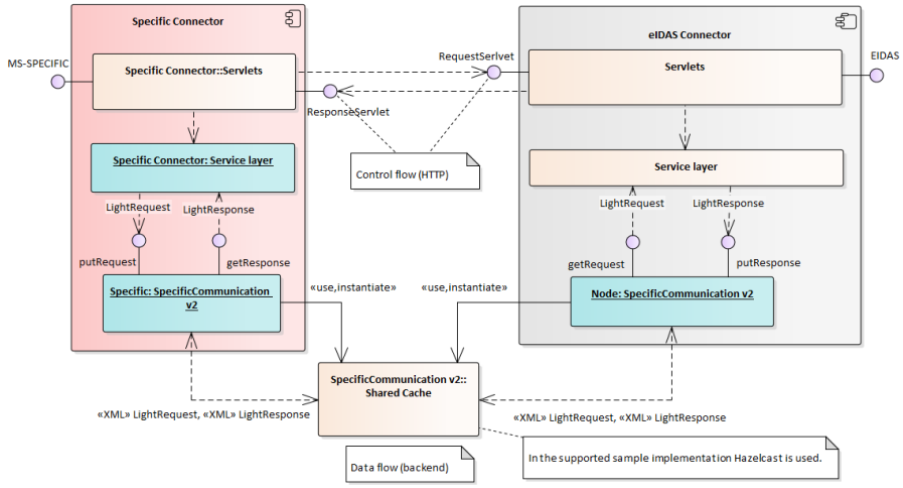

Figure 5: Standard Deployment — possible component layout

The diagram shows how the Specific component in the delivered sample demo chain works. In the Specific Connector there is a service layer that takes care of the necessary transformation of protocols, while a servlet layer helps with the HTTP connections.

The diagram is simplified regarding the control flow; there is no direct connection between the servlet layer, everything here is user-agent based. The actual control flow is visible in the following sequence diagram:

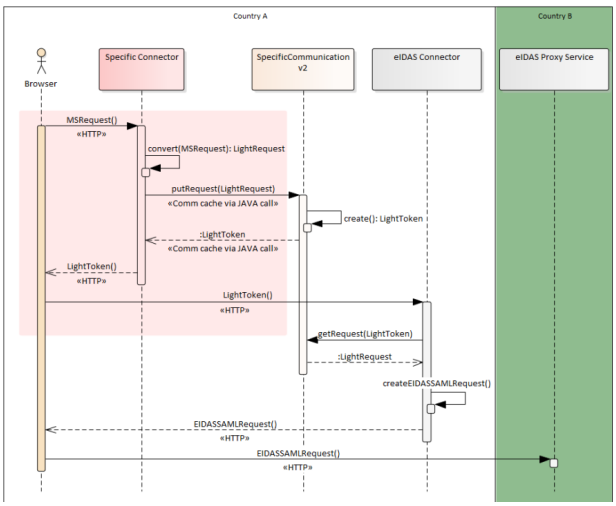

Figure 6: Simplified control flow

The entities shown on the diagram are those referenced as 'reference token' and 'data object' in the previous chapters, which are:

- LightToken is the 'referencing token', There is a unique id inside this token used as the internal key of referenced data object (see section 4.4.1 — *Implementing the LightToken*.
- LightRequest is the 'referenced data object' either a Java POJO or XML, containing all the necessary information to compose an eIDAS SAML Request, or alternatively, all business information that can be

extracted from a standard eIDAS SAML Request (see section 4.4.2 — *Implementing LightRequest / LightResponse*).
- LightResponse is the same as the LightRequest but representing an eIDAS SAML Response (see section 4.4.2 — *Implementing LightRequest / LightResponse*).

The control flow is done by placing a *LightToken* in an HTTP Request and redirecting the user-agent to the eIDAS Node Generic. The URLs for the redirects are shown in the following table.

Table 1: Generic redirect URLs

Action	Relative context URL of Generic
Specific Connector sends Request to the Connector	/SpecificConnectorRequest
Specific Proxy Service sends Response to the Proxy Service	/SpecificProxyServiceResponse

This URL can be changed by updating the web.xml in the EIDAS-Node module. The parameter (form input field name) for the *LightToken* is 'token', and can be customised by renaming in *eu.eidas.auth.commons.EidasParameterKeys.TOKEN.*

When the Generic Node wants to initiate a communication to the Specific, it will do the same, redirect the user-agent to configured Specific URLs, with the *LightToken*:

Table 2: Specific redirect URLs

Action	Relative context URL of Specific
Forward a Request from the eIDAS Network to Specific Proxy Service	/ProxyServiceRequest
Forward a Response from the eIDAS Network to Specific Connector	/ConnectorResponse

These required URLs can be configured differently by changing eu.eidas.node. NodeSpecificViewNames.SPECIFIC_SP_RESPONSE and IDP_REQUEST fields.

4.2.1 Use of LightTokens

The *LightToken* itself is a BASE64 encoded message token. The format is detailed in section 4.4.1 — *Implementing the LightToken*.

LightTokens are automatically created by the API itself, when LightRequest/LightResponse data is supplied to the background communication. The call returns with a BinaryLightObject, that is an HTTP transport ready representation of the LightToken. The BinarylightToken has a digest and is BASE64 encoded.

For the digest, algorithm(s) and password(s) must be supplied to the API as described in section 4.4.1 — Implementing the LightToken.

Since the *LightToken* implementation is internal to the Specific – Generic communication, the format can be changed, but the Generic Node must be compiled and built with the changed API.

4.2.2 Use of LightRequest/LightResponse

The main responsibility of the Specific (or the component responsible for invoking the Generic provided interface) is the composition of LightRequest and LightResponse objects.

The following diagram shows the interfaces and implementing classes in the *EIDAS-LightCommons* module.

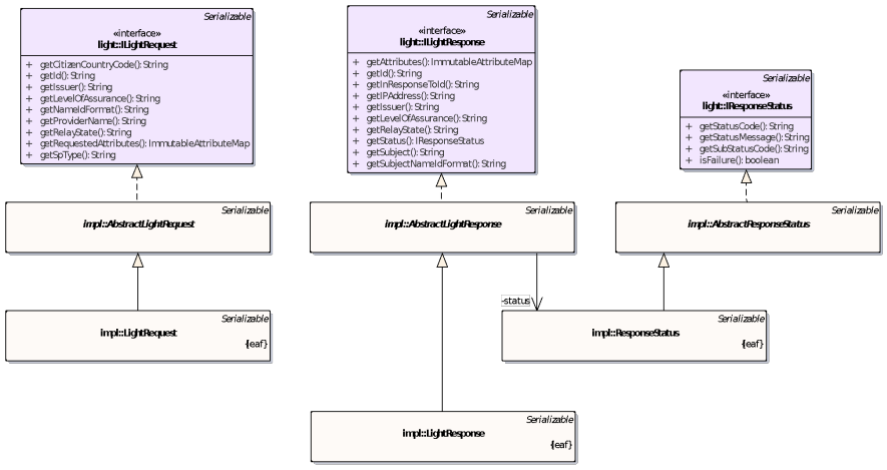

Figure 7: Interfaces and implementing classes in EIDAS-LightCommons

When constructing such an object the Builder pattern needs to be used (for a sample, please see the sample Specific or Generic code).

4.2.2.1 LightRequest

The *LightRequest* can be constructed by creating a *LightRequest.Builder* object using the *LightRequest.builder()* method.

The fields should be populated as shown below.

Table 3: LightRequest properties

Field name	Type	Description
id	String	A unique id that is used internally to correlate with the Response.
issuer	String	Not used in version 2.0. It is the issuer of the previous hop, like the Connector provided for the Specific Proxy. Please do not rely on this information.
citizenCountryCode	String	Country code of the requesting citizen. In version 2.0 and prior, for Specific Proxy it is derived from the last part of domain string of subject of the certificate used for signing. ISO ALPHA-2 format.

Field name	Type	Description
levelsOfAssurance	List	List of eu.eidas.auth.commons.light.ILevelOfAssurance ILevelOfAssurance are represented with a type and a value.Types can only be "notified" or "nonNotified", default type if none is given is "notified". Values defined in enumeration class eu.eidas.auth.commons.protocol.eidas. NotifiedLevelOfAssurance can be used for notified Assurance levels. Non-notified Assurance levels cannot have the same URI prefix as the notified ones.
nameIdFormat	String	Optional string sent to the IdP regarding the identifier format requested (if supported). Maps to NameIDPolicy in eIDAS SAML. Values can be used from eu.eidas.auth.commons.protocol.impl.SamlNameId Format enumeration.
providerName	String	Almost free format text identifier of the Service Provider initiating the request. The text must fit into an XML element value (all XML formatting elements must be escaped).
spType	String	Optional element specifying the sector of the SP or the Connector. Must not be used if the sector of the Connector is set up in the Metadata. Possible values are from eu.eidas.auth.commons.protocol.eidas.SpType enumeration.
spCountryCode	String	Optional element sent to the specific proxy, specifying the country code of the relying party, SP. ISO ALPHA-2 format.
requesterId	String	Optional element specifying the Id of the SP in the Connector MS The Connector validates if it is an URI and has max length of 1024 characters. Note that the requester ID has to be unique within the MS but the Connector does not validate the uniqueness of the requester ID or set/update its value.

Field name	Type	Description
relayState	String	The Specific can use this value to propagate an internal state or similar with the Request, that will be returned with the Response (LightResponse). However it is not recommended to use it for correlation, or to expose internal state information, especially as this field is protected only by the transport layer. This is a feature proper to SAML supported in all the eIDAS specified bindings.
requestedAttributes	ImmutableAttributeMap	The list of requested attributes. It also supports values. When populating, the Attributes from an AttributeRegistry must be used. The AttributeRegistry of the Specific and the Generic must be synchronised. For detailed information on the AttributeRegistry, please see Appendix A.1.

Once the fields are populated, the build() method produces the final object to be passed to the interface.

4.2.2.2 LightResponse

The *LightResponse* is similar.to the *LightRequest* in terms of construction and usage.

The fields should be populated as shown in the following table.

Figure 8: LightResponse properties

Field name	Type	Description
id	String	A unique id.
inResponseToId	String	A unique id that is used internally to correlate with the original LightRequest pair.
issuer	String	The name of the issuer of the Light Response.
consent	String	The consent of the principal of the LightResponse
ipAddress	String	The IP address of the user agent as seen on IdP. If specified, the Proxy Service Node replaces it with an address detected there, and uses it in SubjectConfirmation of the eIDAS SAML Response.
levelOfAssurance	String	Assurance level. Values defined in enumeration class eu.eidas.auth.commons.protocol.eidas.NotifiedLevelOfAssurance can be used.

Field name	Type	Description
subjectNameIdFormat	String	Format of the identifier attribute. Values can be used from eu.eidas.auth.commons.protocol.impl.SamlNameIdFormat enumeration.
subject	String	Subject of the Assertion for the eIDAS SAML Response. It is recommended to use the (natural person) unique identifier attribute value here that is also provided in the attribute set. Theoretically, it can be different, e.g. an email address.
status	Complex	Complex element to provide status information from IdP. Detailed below.
status.failure	boolean	Set this to true if the authentication was a failure. Also, on the receiving side expect this is always true, if the authentication is a failure.
status.statusCode	String	Enforced by the SAML2 specifications. Please refer to the EIDASStatusCode enumeration for values. The value must be in alignment with the "failure" flag.
status.subStatusCode	String	Optional, defined by the SAML2 specifications. Please refer to the EIDASSubStatusCode enumeration for values.
status.statusMessage	String	An optional status message. It is transformed by the Proxy Service Node, a generic error code is appended if it is recognised as an error existing in the *eIDAS-Node Error Codes* document. If not, then a "null – " string is added as prefix.
relayState	String	According to the SAML specification if there was a value to this field in the Request, the Specific Proxy Service must return it, but if it was empty, it can be used to propagate new information to the receiving party.
attributes	ImmutableAttributeMap	The list of attributes in the Assertion. It also supports typed values. When populating, the Attributes from an AttributeRegistry must be used. The AttributeRegistry of the Specific and the Generic must be synchronised. For detailed information on the AttributeRegistry, please see Appendix A.1.

Please note that the *relayState* and *inResponseToId* need to be treated according to the information the Specific Proxy Service received in the *LightRequest*. The *relayState* must be populated from the *relayState* received in the Request (if it was not empty), and the inResponseToId must be the same as the Id field from the *LightRequest* pair. This means an implicit requirement to store the information in the Specific to make the correlation.

4.2.3 Service interface and implemented beans

To use the provided API, simply use the provided methods of interface SpecificCommunicationService.

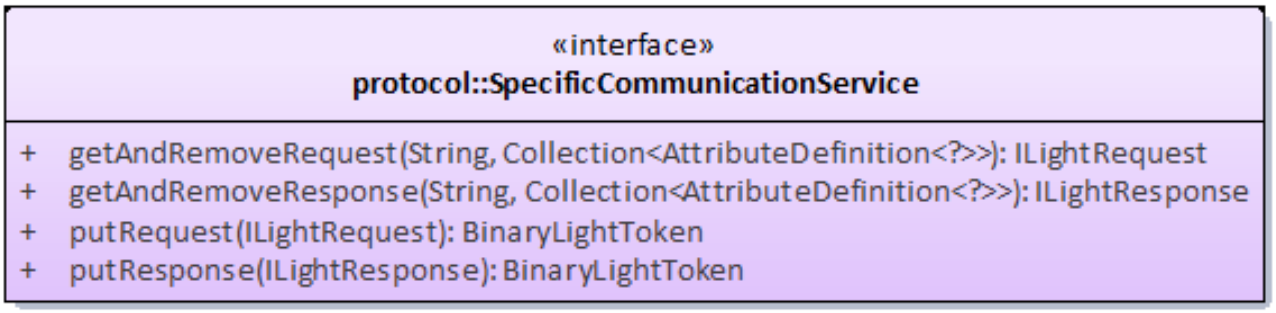

Figure 9: SpecificCommunicationService interface

In the Specific Connector, calls to the *putRequest()* and the *getAndRemoveResponse()* methods are necessary, while the Specific Proxy Service must invoke *getAndRemoveRequest()* and *putResponse().*

Placing a *LightRequest/LightResponse* with the 'put' operation results in a LightToken, that can be used to pass the control flow.

The methods *getAndRemove* expect the *LightToken* from the HTTP Request (BASE64 encoded), and an *AttributeDefinition* list (from *AttributeRegistry*) in order to be able to serialize/deserialize attributes to XML based on the identifier, the nameURI. The result is the Light data object.

The implementation of this interface is based on the generic pattern used in the eIDAS-Node. There are two classes, SpecificConnectorCommunicationServiceImpl and *SpecificProxyserviceCommunicationServiceImpl*, that encapsulate all the *LightToken* encoding operations (see Figure 10). They can be initialised by Spring or by programmatic way, you can find an example of the former in the sample demo Specific implementation.

Upon initialisation the following parameters are set (by the constructor).

Table 4: LightToken operations

Field name	Type	Usage
lightToken[X]IssuerName	String	A custom identifier of the calling service is placed in the LightToken (like EUPROXYSERVICE)
lightToken[X]Secret	String	A secret (passphrase) that is used to create the digest. This must be shared between the communicating parties (i.e. Specific Connector and Generic Connector).

Field name	Type	Usage
lightToken[X]Algorithm	String	A digest algorithm for the LightToken, depending on the capabilities of the underlying Java security MessageDigestSpi implementation. At least SHA256 is recommended.

The [X] can be either Request or Response, so settings can differ for handling the different directions.

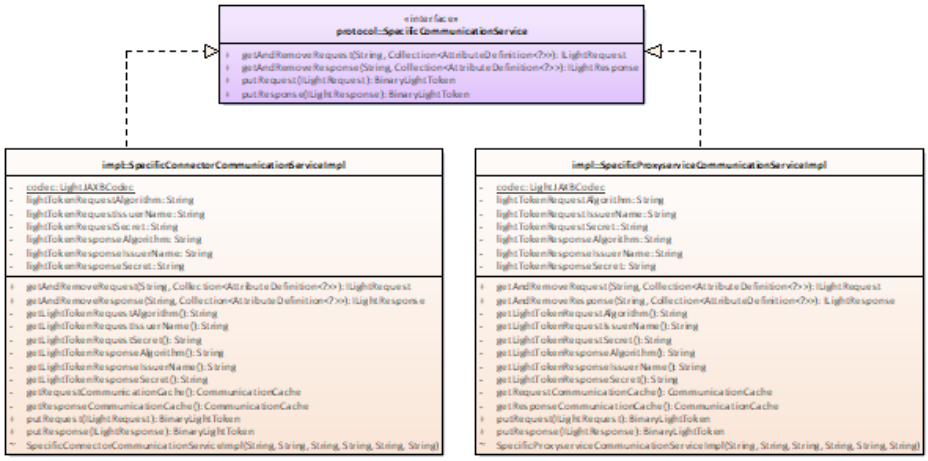

Figure 10: SpecificCommunicationService

The sample demo Specific implementation uses Spring to instantiate these classes as Beans. Configuration is via a configuration file through the application context, so it is reloadable.

4.2.4 Back-end communication with Ignite

The provided API uses Ignite by default to communicate between the Generic and Specific parts, and to replicate eIDAS sessions between Nodes of the same type (Generic-Generic and Specific-Specific) to achieve high availability. With this choice neither HTTP Session nor enterprise bean replication are needed.

The communication is via Shared Caches, with dedicated identifiers. These default files are loaded by the Spring application context. These caches need to be declared in the Ignite configuration. The following shows the default values used by the communication API.

Table 5: Ignite shared caches

Cache name	Role
specificNodeConnectorRequestCache	Stores Requests sent from Specific Connector to Generic Connector
nodeSpecificConnectorResponseCache	Stores Responses sent from Generic Connector to Specific Connector.

Cache name	Role
nodeSpecificProxyserviceRequestCache	Stores Requests sent from Generic Proxy Service to Specific Proxy Service
specificNodeProxyserviceResponseCache	Stores Responses sent from Specific Proxy Service to Generic Proxy Service

Note that the cache names in Ignite configuration corresponding to the above caches are respectively: specificNodeConnectorRequestCache, nodeSpecificProxyserviceRequestCache, specificNodeProxyserviceResponseCache and nodeSpecificConnectorResponseCache.

All parameters are system dependent, mostly affecting the performance. . Setting expiryPolicyFactory property to limit the duration for the caches using Ignite configuration has an extra security importance. This parameter should be set to a minimum value sufficient enough for the user-agent to be redirected to the next component in the control flow. Below is an example to set it for 5 minutes for the specificNodeConnectorRequestCache:

```
< bean   class = "org.apache.ignite.configuration.CacheConfiguration" >
  < property   name = "name"   value = "specificNodeConnectorRequestCache"
/>
  < property   name = "atomicityMode"   value = "ATOMIC"  />
  < property   name = "backups"   value = "1"  />
  < property   name = "expiryPolicyFactory" >
    < bean   class = "javax.cache.expiry.CreatedExpiryPolicy"   factory-
method = "factoryOf" >
      < constructor -arg>
        < bean   class = "javax.cache.expiry.Duration" >
          < constructor -arg  value = "MINUTES"  />
          < constructor -arg  value = "5"  />
        </ bean >
      </ constructor -arg>
    </ bean >
  </ property >
</ bean >
```

To override the default values the following entries must be defined e.g in **specificCommunicationDefinitionConnector.xml**

```
< entry   key = "specific.node.connector.request.cache.name" >specificNodeCo
nnectorRequestCacheExternal</ entry >
< entry   key = "node.specific.connector.response.cache.name" >nodeSpecificC
onnectorResponseCacheExternal</ entry >
```

and in **specificCommunicationDefinitionProxyService.xml**

```
< entry   key = "node.specific.proxyservice.request.cache.name" >nodeSpecifi
cProxyserviceRequestCacheExternal</ entry >
< entry   key = "specific.node.proxyservice.response.cache.name" >specificNo
deProxyserviceResponseCacheExternal</ entry >
```

It is also necessary to update the cache names in igniteSpecificCommunication.xml so that those match the external ones. For example, to change the cache name related to *specificNodeConnectorRequestCache* one needs to modify the property name as follows:

```
< bean   class = "org.apache.ignite.configuration.CacheConfiguration" >
  < property   name = "name"   value = "specificNodeConnectorRequestCacheExt
ernal" />
  < property   name = "atomicityMode"   value = "ATOMIC" />
  < property   name = "backups"   value = "1" />
  < property   name = "expiryPolicyFactory"   ref = "7_minutes_duration" />
</ bean >
```

Note that the default values are declared in files **specificCommunicationDefinitionConnector.xml** and **specificCommunicationDefinitionProxyService.xml** in EIDAS-SpecificCommunicationDefinition/src/main/resources/default.

Please go to Ignite documentation for more details.

This implementation limits the possibility for a takeover of the flow by a third party (even though other security countermeasures and mechanisms applied to eIDAS authentication will prevent it happening).

Physical communication between the shared maps is via XML. This allows integration with the Generic part using your own implementation rather than the provided API, or even to use technology other than Java. See section 4.4 — *Integrating the provided Generic with custom implementation of communication* for details.

4.2.5 Back-end communication with alternative to Ignite

It is possible to replace Ignite with an alternative, like Oracle Coherence, Redis, etc. without impacting the *SpecificCommunicationService*. In this case the API needs no modification, and instead of using *ConcurrentCacheService* implementation from EIDAS-Commons, another implementation can be packaged. This must implement the *CommunicationCache* interface, and for the internal session correlation, a *ConcurrentCacheService* interface. The hard coded identifier of Maps must be retained.

4.2.6 Overriding default communication cache names

In advanced configurations, it is possible to override the default communication cache names for Ignite caches. The names can be defined in **specificCommunicationDefinitionConnector.xml** and in **specificCommunicationDefinitionProxyService.xml** by using the configuration keys as outlined in below tables.

In **specificCommunicationDefinitionConnector.xml**

Key	Description
specific.node.connector.request.cache.name	Name for cache that stores Requests sent from Specific Connector to Generic Connector
node.specific.connector.response.cache.name	Name for cache that Stores Responses sent from Generic Connector to Specific Connector

In **specificCommunicationDefinitionProxyService.xml**

Key	Description
node.specific.proxyservice.request.cache.name	Name for cache that stores Requests sent from Generic Proxy Service to Specific Proxy Service
specific.node.proxyservice.response.cache.name	Name for cache that stores Responses sent from Specific Proxy Service to Generic Proxy Service

4.2.7 Back-end communication in Monolithic Deployment

By default the Monolithic Deployment does not provide a distributed map, only an in-memory Guava cache, so high availability can be achieved by using advanced load balancing, for example sticky sessions. This is a build-time restriction only, so it is possible to unlock the shared map.

The following diagram shows the basic component layout for the Monolithic Deployment.

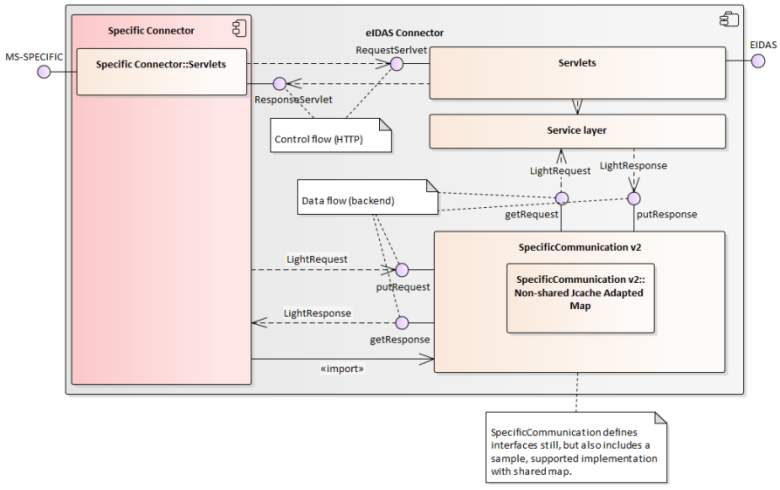

Figure 11: Basic component layout for Monolithic Deployment

4.2.8 Incoming Light Request Validation

The incoming Light Request from MS specific is currently validated for:

Key	Description
incoming.lightRequest.max.number.characters	Maximum size in characters for incoming Light Request The default is 65535 External Configuraion File /specificConnector/specificCommunicationDefinitionConnector.xml

4.2.9 Incoming Light Response Validation

The incoming Light Response from MS specific is currently validated for:

Key	Description
incoming.lightRequest.max.number.characters	Maximum size in characters for incoming Light Request The default is 65535 External Configuraion File /specificConnector/specificCommunicationDefinitionProxyservice.xml

4.3 Re-implementing SpecificCommunication

It is possible to re-implement the EIDAS-SpecificCommunicationDefinition, and use a very different integration approach/pattern. It means rewriting the parts responsible for the data flow and/or the control flow.

The best way to do this is by retaining the *SpecificCommunicationService* interface, so that there is less impact to the Generic part (most likely the application context driven injections need to be adjusted).

4.4 Integrating the provided Generic with custom implementation of communication

Those who prefer not to use the provided API in their Specific solution can integrate with the Generic Node by implementing the control and data flows, fulfilling the same requirements given by the eID API implementation. The pattern, which is the same through the whole communication process, is shown on the following diagram.

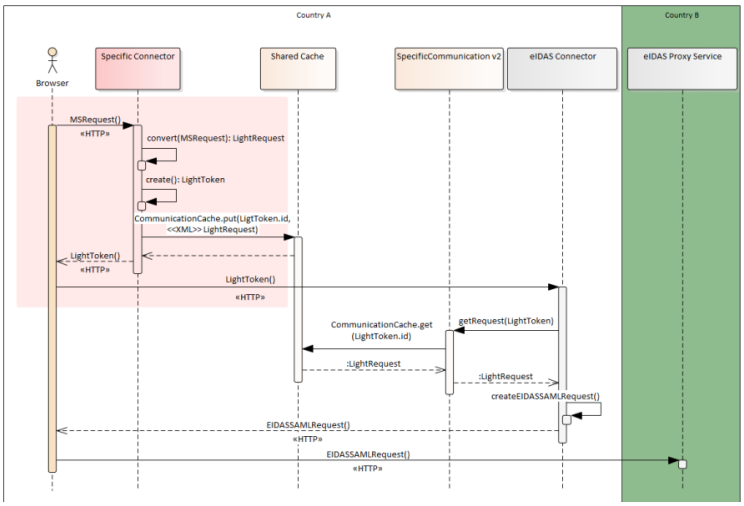

Figure 12: Integrating the provided Generic with custom implementation of communication

4.4.1 Implementing the LightToken

The basic concepts of *LightTokens* are described in section 4.2.1 — *Use of LightTokens*, and the communication is described in section 4.2 — *Using the provided SpecificCommunication API*. The *LightToken* needs to be placed in the 'token' parameter of the HTTP Request.

The physical form is composed of the following fields.

Table 7: LightToken format

Field	Description
Issuer name	A simple text string that helps identify (debug) which component is sending the redirect (e.g. EIDASPROXY)
ID	A unique identifier. This identifier is used to reference the real data object (LightRequest/LightResponse) in the backend communication. It must be unique within the eviction period defined for the data object.
Creation timestamp	A timestamp showing when the LightToken was created. Together with the ID it makes the token unique and prevents reuse, it also provides a first line of check when discarding an expired request. It also helps debugging. The format is: yyyy-MM-dd HH:mm:ss SSS (see https://www.joda.org/joda-time/apidocs/org/joda/time/format/DateTimeFormat.html[4]).
Digest	This is a custom digest encoded in BASE64. It guarantees a minimum level of integrity and a minimum level of assurance that the token has originated from a trusted component. It is there to help the first-line detection and discarding of requests in a denial of service attack.

These four fields must be concatenated with a vertical bar character ('|', ASCII 124) without any trailing spaces, and encoded in BASE64.

Example:

specificCommunicationDefinitionConnectorRequest|852a64c0-8ac1-445f-b0e1-992ada493033|2017-12-11 14:12:05 148|7M8p+uP8CKXuMi2IqSda1tg452WlRvcOSwu0dcisSYE=

In BASE64 encoded:

c3BlY2lmaWNDb21tdW5pY2F0aW9uRGVmaW5pdGlvbkNvbm5lY3RvclJlcXVlc3R8ODUyYTY0YzAtOGFjMS00NDVmLWIwZTEtOTkyYWRhNDkzMDMzfDIwMTctMTItMTIgMTQ6MTI6MDUgMTQ4fDdNOHArdVA4Q0tYdU1pMklxU2RhMXRnNDUyV2xSdmNPU3d1MGRjaXNTWUU9

To create the message digest, the fields must be concatenated together into the following sequence: 2, 1 then 3 (id, issuer, timestamp) with the vertical bar separating. Append with another vertical bar and finally the secret shared between the communicating parties (i.e., Specific Connector and Generic Connector) (see lightToken[X]Secret in Table 4. The digest is calculated from the binary representation of this string.

4 https://www.joda.org/joda-time/apidocs/org/joda/time/format/DateTimeFormat.html+

Example before calculating the digest:

852a64c0-8ac1-445f-b0e1-992ada493033|specificCommunicationDefinitionConnectorRequest|2017-12-11 14:12:05 148|mySecretConnectorRequest

Finally, the secret needs to be replaced with the calculated digest value, in a BASE64 encoded form, so it results in the first example. Then the whole token must be encoded in BASE64 before going with the HTTP request.

By default the token is limited to 1024 bytes, which can be changed in *LightTokenEncoder.MAX_TOKEN_SIZE*.

When receiving a *LightToken*, it must be processed in the reverse way to how it was constructed.

First, there are some validations & checks to be performed:

- Compare the actual size to a reasonable maximum (maximum not defined, at your discretion).
- Decode from BASE64.
- Split to number of parts (with a limit of four possible parts).
- Using the parts, verify the digest.
- Sanitise the parts, validate formats.
- Optionally the check the timestamp for expiration.

At the end of this process the ID to recover the Light data object can be used.

4.4.2 Implementing LightRequest / LightResponse

The provided eIDAS-Node uses XML for background communication. The custom Specific implementation should compose and parse LightRequest and LightResponse objects according to the following schema definitions. The resulting xml must include a (default) namespace in the root node (xmlns="http://cef.eidas.eu/LightRequest" or xmlns="http://cef.eidas.eu/LightResponse").

4.4.2.1 The XML LightRequest

The following figure shows a simplified schema diagram representing the schema for *LightRequest* XMLs.

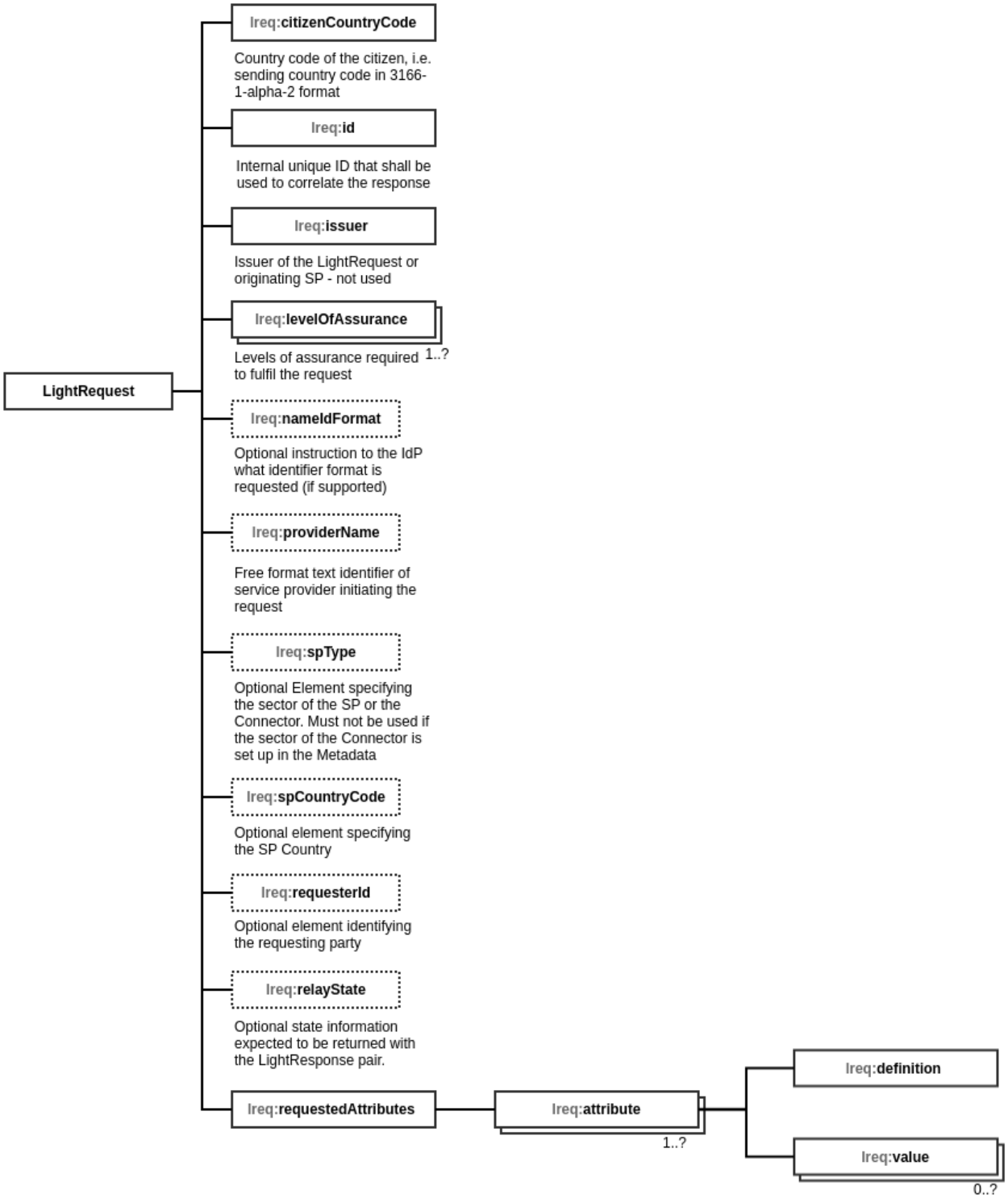

Figure 13: XML LightRequest schema:

The following schema definition can be found in Appendix A.2.1 (there is a copy of this schema files in *EIDAS-SpecificCommunicationDefinition/src/main/resources/xsds* folder of the source code delivery).

Usually the Request does not contain any attribute values, just references for the requested ones. In case it is needed, please read the next section where the use of values is described with the LightResponse.

The XML element *requestedAttributes* is oversimplified in the schema document. The XSD accepts generic xs:string typed values, however – in real life – the marshalling logic expects types according to the XML type definitions defined to the *AttributeRegistry*. The mapping is done through the *AttributeRegistry* with the nameUri identifier, in other words the name of the attribute.

When the AttributeRegistry is referencing a certain type, that type must be supplied in the value elements. Since the current implementation is using the same marshallers/unmarshallers for constructing SAML messages and processing Light XML objects, the value format is the same as in the *eIDAS SAML Attribute Profile v1.1_2* Technical specification.

4.4.2.2 The XML LightResponse

The following schema diagram shows the simplified XML *LightResponse* object:

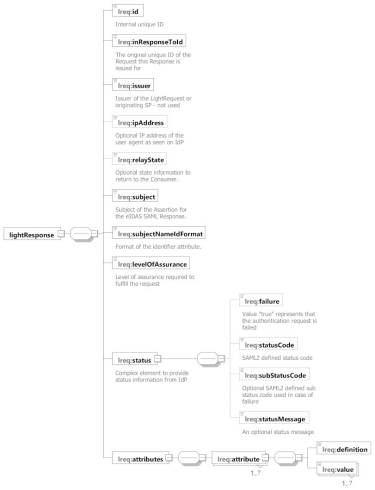

Figure 14: Simplified XML LightResponse schema

The schema file is included in the folder EIDAS-SpecificCommunicationDefinition/src/test/resources of the source delivery, also available in Appendix A.2.2.

The 'attributes' element (also the *requestedAttributes* in the *LightRequest*) is not fully defined in the XSD.

When an attribute needs to be presented to this list in the Light object, there is a sequence which must be followed.

First the 'definition' element should be created. The text value content of this element must exactly match one nameUri from the AttributeRegistry (either from the normal eIDAS Profile or from the "additional attributes"). For example:

```
< definition > http://eidas.europa.eu/attributes/naturalperson/
PersonIdentifier </ definition >
```

The *CommunicationDefinition* uses this element to identify the appropriate AttributeRegistry element, including the marshaller/unmarshaller methods. This also means, that any value appended to the list MUST follow the declared XMLType and MUST meet with the format expected by the serialisation.

The *PersonIdentifier* for example is serialized with the *LiteralStringAttributeValueMarshaller* class, so it can be a free format String, such as:

```
< value >Vivaldi-987654321</ value >
```

So the whole attribute element would look as follows:

```
< attribute >
    < definition > http://eidas.europa.eu/attributes/naturalperson/
CurrentGivenName </ definition >
    < value >Antonio</ value >
    < value >Lucio</ value >
    < value >Vivaldi</ value >
</ attribute >
```

The supported built-in types of EIDAS-Commons are described in the following sections. These can be extended if necessary.

4.4.2.3 DateTimeAttribute

This type of attribute value is in the format:

YYYY + "-" + MM + "-" + DD (as defined for *xsd:date*)

Example:

```
< attribute >
    < definition > http://eidas.europa.eu/attributes/naturalperson/
DateOfBirth </ definition >
    < value >2018-02-28</ value >
</ attribute >
```

4.4.2.4 BooleanAttribute

Boolean values need to follow simple choice of string values:
"true" or "false".

Not used in standard eIDAS Profile.

4.4.2.5 IntegerAttribute

This element supports a signed Integer between *Integer.MIN_VALUE* and *Integer.MAX_VALUE.*

Not used in standard eIDAS Profile.

4.4.2.6 LiteralStringAttribute

The most common generic String type, contains a simple string.

4.4.2.7 GenderAttribute

This attribute can have one of three values:

"Male", "Female" and "Unspecified"

Example:

```
< attribute >
    < definition > http://eidas.europa.eu/attributes/naturalperson/Gender </
definition >
    < value >Male</ value >
</ attribute >
```

4.4.2.8 PostalAddressAttribute

This is the most complex attribute value with several subfields. The structure follows the Core ISA Vocabulary definition, however when it gets serialized, to maintain a flat format it is encoded in BASE64.

In the unencoded format the PostalAddress XML fragment should look like:

```
< AddressId > http://address.example/id/be/eh11aa </ AddressId >
< PoBox >1234</ PoBox >
< LocatorDesignator >28</ LocatorDesignator >
< LocatorName >DIGIT building</ LocatorName >
< CvAddressArea >Etterbeek</ CvAddressArea >
< Thoroughfare >Rue Belliard</ Thoroughfare >
< PostName >ETTERBEEK CHASSE</ PostName >
< AdminUnitFirstLine >BE</ AdminUnitFirstLine >
< AdminUnitSecondLine >ETTERBEEK</ AdminUnitSecondLine >
< PostCode >1040</ PostCode >
< FullCvaddress >Rue Belliard 28\nBE-1040 Etterbeek</ FullCvaddress >
```

For the description of the fields, please consult the Core ISA Vocabulary. It may be sufficient to use only the fullCvaddress.

Encoded, and in final form, it would look as follows:

```
< attribute >
    < definition > http://eidas.europa.eu/attributes/naturalperson/
CurrentAddress </ definition >
```

```
    < value >PEFkZHJlc3NJZD5odHRwOi8vYWRkcmVzcy5leGFtcGxlL2lkL2JlL2VoMTFhYTw
vQWRkcmVzc0lkPg0KPFBvQm94PjEyMzQ8L1BvQm94ID4NCjxMb2NhdG9yRGVzaWduYXRvcj4yODwv
TG9jYXRvckRlc2lnbmF0b3I+DQo8TG9jYXRvck5hbWU+RElHSVQgYnVpbGRpbmc8L0xvY2F0b3JOY
W1lPg0KPEN2QWRkcmVzc0FyZWE+RXR0ZXJiZWVrPC9DdkFkZHJlc3NBcmVhPg0KPFRob3JvdWdoZm
FyZT5SdWUgQmVsbGlhcmQ8L1Rob3JvdWdoZmFyZT4NCjxQb3N0TmFtZT5FVFRFUkJFRUsgQ0hBU1N
FPC9Qb3N0TmFtZT4NCjxBZG1pblVuaXRGaXJzdExpbmU+QkU8L0FkbWluVW5pdEZpcnN0TGluZT4N
CjxBZG1pblVuaXRTZWNvbmRMaW5lPkVUVEVSQkVFSzwvQWRtaW5Vbml0U2Vjb25kTGluZT4NCjxQb
3N0Q29kZT4xMDQwPC9Qb3N0Q29kZT4NCjxGdWxsQ3ZhZGRyZXNzPlJ1ZSBCZWxsaWFyZCAyOFxuQk
UtMTA0MCBFdHRlcmJlZWs8L0Z1bGxDdmFkZHJlc3M+
    </ value >
</ attribute >
```

5 Appendix A: Diagrams and Schemas

5.1 Attribute Registry

AttributeRegistry is a catalogue of attributes defined for the eIDAS-Node. The attribute registry is implemented in the class *eu.eidas.auth.commons.attribute.AttributeRegistry*. An attribute registry can be instantiated programmatically with the *AttributeRegistry* class or loaded from a file.

The attribute registry file is composed of attribute definitions. They represent the *eu.eidas.auth.commons.attribute.AttributeDefinition* class.

An attribute definition is composed of the following properties:

- **NameUri** : [mandatory]: the name URI of the attribute (full name and must be a valid URI).
- **FriendlyName** : [mandatory]: the friendly name of the attribute (short name);
- **PersonType** : [mandatory]: either NaturalPerson, LegalPerson, RepresentativeNaturalPerson or RepresentativeLegalPerson.
- **Required** : [optional]: whether the attribute is required by the specification (and is part of the minimal data set which must be requested).
- **TransliterationMandatory** : [optional]: whether the attribute values must be transliterated if provided in non LatinScript variants.
- **UniqueIdentifier** : [optional]: whether the attribute is a unique identifier of the person (at least one unique identifier attribute must be present in authentication responses).
- **XmlType.NamespaceUri** : [mandatory]: the XML namespace URI for the attribute values, for example: http://www.w3.org/2001/XMLSchema for an XML Schema string.
- **XmlType.LocalPart** : [mandatory]: the name of the XML type for the attributes values, for example: 'string' for an XML Schema string.
- **XmlType.NamespacePrefix** : [mandatory]: the name of the XML namespace prefix for the attributes values, for example: 'xs' for an XML Schema string.
- **AttributeValueMarshaller** : [mandatory]: the name of a class available in the classpath which implements the *eu.eidas.auth.commons.attribute.AttributeValueMarshaller* interface.

Each attribute definition in the properties file is assigned a unique ID followed by a dot (.) which allows the parser to associate properties to one given attribute definition. The unique ID can be any string not containing a period. A convention can be to use numbers as unique IDs as in the example above.

All properties used by the parser can be found in *eu.eidas.auth.commons.attribute.AttributeSetPropertiesConverter.Suffix*.

5.1.1 Hard coded attributes

In the eIDAS-Node, the eIDAS standard attributes are hard coded in classes *NaturalPersonSpec*, *LegalPersonSpec*, *RepresentativeNaturalPersonSpec* and *RepresentativeLegalPersonSpec*. The reasons for hard coding is that they change only when the Technical Specifications are changed, and strong reference of the attributes are needed to carry out eIDAS-based validations. Beside this hard-wired specification, there are also XML schema definitions *saml_eidas_natural_person.xsd*, *saml_eidas_legal_person.xsd*, *saml_eidas_representative_natural_person.xsd* and *saml_eidas_representative_legal_person.xsd* in SAMLEngine common resources folder, loaded by SAML bootstrap and used only to validate not encrypted response assertions (thus may never be used in production environment).

There is another set of attribute definitions that can be configured by specifying *additionalAttributeRegistryFile* in *SamlEngine.xml* - as a general xml definition for so called 'additional attributes'. By default the package includes a *saml-engine-additional-attributes.xml* example file configured for the specific *ProtocolEngine* instance.

5.1.2 Class related attribute registries

The Demo Specific and Demo SP / IdP use file/memory based registries instead of hard coded.

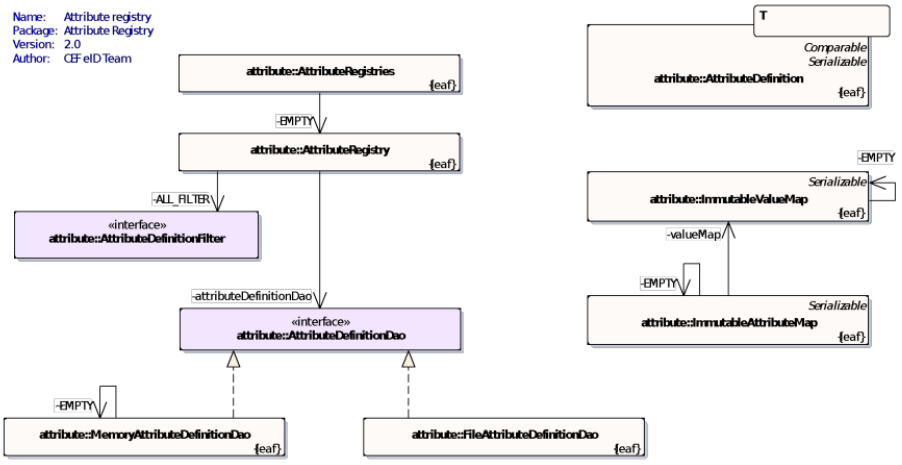

Figure 15: Classes related to basic attribute registry

The diagram above shows the classes related to basic attribute registry. The *AttributeRegistries* class acts like a static factory for creating registries. Depending on which method is called, it provides an *AttributeRegistry* encapsulating a *MemoryAttributeDefinitionDao* (method 'of') or a *FileAttributeDefinitionDao* (fromFile), both extending the *AttributeDefinitionDao* interface. Both are based on *SingletonAccessors* of *ImmutableSortedSets* containing the actual *AttributeDefinitions*.

AttributeRegistry class also provides an interface called AttributeDefinitionFilter, that enables quick filtering of received attributes based on anonymous classes. The example legal MDS filter from *EidasProtocolProcessor*:

```
public static final AttributeRegistry.AttributeDefinitionFilter
MANDATORY_LEGAL_FILTER = new AttributeRegistry.AttributeDefinitionFilter()
{
    @Override
    public boolean accept( @Nonnull AttributeDefinition<?>
attributeDefinition) {
        return attributeDefinition.isRequired() &&
attributeDefinition.getPersonType() == PersonType.LEGAL_PERSON;
    }
};
```

An AttributeRegistry contains definitions only, where values are also needed ImmutableAttributeMap are being used. *ImmutableAttributeMap* is thread safe, serializable and immutable - instantiated by builder pattern - follows the heterogeneous container pattern. When built, internally contains ImmutableValueMap, but basically a set of AttributeDefinitions with associated AttributeValue(s).

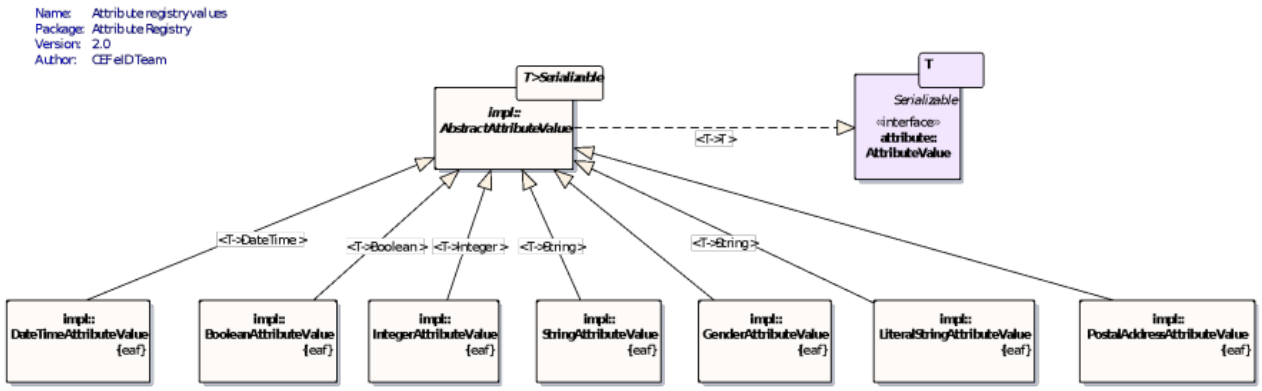

Figure 16: Attribute registry values

The values are typed, therefore can contain complex elements like *PostalAddressAttributeValue. The generic AbstractAttributeValue is responsible to hold any information on SAML attribute level (only transliteration by now).*

Since the values are needed to be converted between user types and XML representation eligible for SAML Assertion, there are *AttributeValueMarshallers* defined for each type:

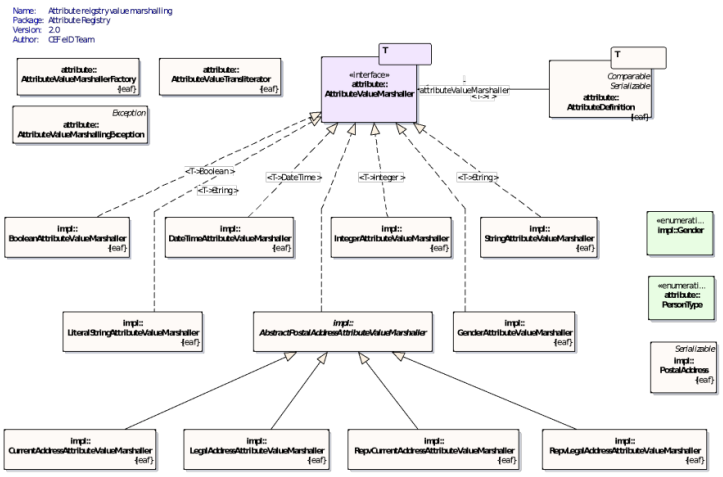

Figure 17: Attribute registry value marshalling

The marshaller interface definition is actually a part of the attribute definition.

5.2 XSD Schemas for Light Objects

5.2.1 LightRequest schema

```
<? xml   version = "1.0"   encoding = "utf-8" ?>
<!--
  ~ Copyright (c) 2020 by European Commission
  ~
```

```
  ~ Licensed under the EUPL, Version 1.2 or - as soon they will be
  ~ approved by the European Commission - subsequent versions of the
  ~ EUPL (the "Licence");
  ~ You may not use this work except in compliance with the Licence.
  ~ You may obtain a copy of the Licence at:
  ~  https://joinup.ec.europa.eu/page/eupl-text-11-12
  ~
  ~ Unless required by applicable law or agreed to in writing, software
  ~ distributed under the Licence is distributed on an "AS IS" basis,
  ~ WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  ~ implied.
  ~ See the Licence for the specific language governing permissions and
  ~ limitations under the Licence.
  -->
< xs :schema  xmlns:xs = "http://www.w3.org/2001/XMLSchema"
      targetNamespace = "http://cef.eidas.eu/LightRequest"  elementFormDefa
ult = "qualified"  version = "1.2" >
 < xs :element  name = "lightRequest" >
  < xs :complexType>
   < xs :sequence>
    < xs :element  name = "citizenCountryCode"  minOccurs = "1"  maxOccurs
= "1"  type = "xs:string" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Country code of the citizen, ie.: sending country code in
       3166-1-alpha-2 format
      </ xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "id"  type = "xs:string"  minOccurs = "1"
maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Internal unique ID what will be used to correlate the response
      </ xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "issuer"  type = "xs:string" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Issuer of the LightRequest or originating SP - not used</ xs :documentation
>
     </ xs :annotation>
    </ xs :element>
```

```
    < xs :element  name = "levelOfAssurance"   minOccurs = "1"   maxOccurs =
"unbounded" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Level of assurance required to fulfill the request</ xs :documentation>
     </ xs :annotation>
     < xs :complexType>
      < xs :simpleContent>
       < xs :extension  base = "xs:anyURI" >
        < xs :attribute  name = "type"   default = "notified" >
         < xs :simpleType>
          < xs :restriction  base = "xs:string" >
           < xs :enumeration  value = "notified" >
            < xs :annotation>
             < xs :documentation  xml:lang = "en" >
              Default value, only one notified level of assurance should be
              given and should have a valid value (regarding specs).
             </ xs :documentation>
            </ xs :annotation>
           </ xs :enumeration>
           < xs :enumeration  value = "nonNotified" >
            < xs :annotation>
             < xs :documentation  xml:lang = "en" >
              Non notified levels of Assurance, the prefix of notified
              level of assurance cannot be used for these levels of assuranc
e.
             </ xs :documentation>
            </ xs :annotation>
           </ xs :enumeration>
          </ xs :restriction>
         </ xs :simpleType>
        </ xs :attribute>
       </ xs :extension>
      </ xs :simpleContent>
     </ xs :complexType>
    </ xs :element>
    < xs :element  name = "nameIdFormat"   minOccurs = "0"   maxOccurs = "1"
>
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Optional instruction to the IdP what identifier format is requested (if sup
ported).</ xs :documentation>
     </ xs :annotation>
     < xs :simpleType>
```

```
      < xs :restriction  base = "xs:string" >
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:persistent" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:transient" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:1.1:nameid-
format:unspecified" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:1.1:nameid-
format:emailAddress" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:1.1:nameid-
format:X509SubjectName" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:1.1:nameid-
format:WindowsDomainQualifiedName" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:kerberos" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:entity" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:encrypted" />
      </ xs :restriction>
     </ xs :simpleType>
    </ xs :element>
    < xs :element  name = "providerName"  type = "xs:string"  minOccurs =
"0"  maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Free format text identifier of service provider initiating the request.</
xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "spType"  minOccurs = "0"  maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Optional element specifying the sector of the SP or the
       Connector. Must not be used if the sector of the Connector is set up
in the Metadata.
      </ xs :documentation>
     </ xs :annotation>
     < xs :simpleType>
      < xs :restriction  base = "xs:string" >
       < xs :enumeration  value = "public" />
       < xs :enumeration  value = "private" />
      </ xs :restriction>
     </ xs :simpleType>
```

```
    </ xs :element>
    < xs :element  name = "spCountryCode"  minOccurs = "0"  maxOccurs =
"1"  type = "xs:string" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Country code of the SP, ie.: sending country code in
       3166-1-alpha-2 format
      </ xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "requesterId"  type = "xs:anyURI"  minOccurs =
"0"  maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Optional element specifying the Id of the SP. Must be unique
       within the Connector MemberState.
      </ xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "relayState"  type = "xs:string"  minOccurs =
"0"  maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Optional state information expected to be returned with the LightResponse p
air.</ xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "requestedAttributes" >
     < xs :complexType>
      < xs :sequence>
       < xs :element  name = "attribute"  maxOccurs = "unbounded" >
        < xs :complexType>
         < xs :sequence>
          < xs :element  name = "definition"  type = "xs:string"
minOccurs = "1"  maxOccurs = "1" />
          < xs :element  name = "value"  type = "xs:string"  minOccurs =
"0"  maxOccurs = "unbounded" />
         </ xs :sequence>
        </ xs :complexType>
       </ xs :element>
      </ xs :sequence>
     </ xs :complexType>
    </ xs :element>
   </ xs :sequence>
```

```
  </ xs :complexType>
 </ xs :element>
</ xs :schema>
```

The previous schema is a copy of the xsd schema which can be found in the *EIDAS-SpecificCommunicationDefinition/src/main/resources/xsd*s folder

5.2.2 Light Response schema

```
<? xml  version = "1.0"  encoding = "utf-8" ?>
<!--
  ~ Copyright (c) 2020 by European Commission
  ~
  ~ Licensed under the EUPL, Version 1.2 or - as soon they will be
  ~ approved by the European Commission - subsequent versions of the
  ~ EUPL (the "Licence");
  ~ You may not use this work except in compliance with the Licence.
  ~ You may obtain a copy of the Licence at:
  ~  https://joinup.ec.europa.eu/page/eupl-text-11-12
  ~
  ~ Unless required by applicable law or agreed to in writing, software
  ~ distributed under the Licence is distributed on an "AS IS" basis,
  ~ WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  ~ implied.
  ~ See the Licence for the specific language governing permissions and
  ~ limitations under the Licence.
  -->
< xs :schema  xmlns:xs = "http://www.w3.org/2001/XMLSchema"  targetNamespac
e = "http://cef.eidas.eu/LightResponse"
      elementFormDefault = "qualified"  version = "1.2" >
 < xs :element  name = "lightResponse" >
  < xs :complexType>
   < xs :sequence>
    < xs :element  name = "id"  type = "xs:string"  minOccurs = "1"
maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en" >Internal unique ID</ xs :documen
tation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "inResponseToId"  type = "xs:string"  minOccurs
= "1"  maxOccurs = "1" >
     < xs :annotation>
```

```
      < xs :documentation  xml:lang = "en"
>The original unique ID of the Request this Response is issued for</ xs :doc
umentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "consent"  minOccurs = "0"  maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en" >The consent of the principal.</
xs :documentation>
     </ xs :annotation>
     < xs :simpleType>
      < xs :restriction  base = "xs:anyURI" >
       < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:consent:unspecified" />
       < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:consent:obtained" />
       < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:consent:prior" />
       < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:consent:current-implicit" />
       < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:consent:current-explicit" />
       < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:consent:unavailable" />
       < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:consent:inapplicable" />
      </ xs :restriction>
     </ xs :simpleType>
    </ xs :element>
    < xs :element  name = "issuer"  type = "xs:string"  minOccurs = "1"
maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Issuer of the LightRequest or originating SP - not used</ xs :documentation
>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "ipAddress"  type = "xs:string"  minOccurs = "0"
  maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Optional IP address of the user agent as seen on IdP</ xs :documentation>
     </ xs :annotation>
    </ xs :element>
```

```
    < xs :element  name = "relayState"   type = "xs:string"   minOccurs =
"0"   maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Optional state information to return to the Consumer.</ xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "subject"   type = "xs:string"   minOccurs = "0"
maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Subject of the Assertion for the eIDAS SAML Response.</ xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "subjectNameIdFormat"   minOccurs = "0"
maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Format of the identifier attribute.</ xs :documentation>
     </ xs :annotation>
     < xs :simpleType>
      < xs :restriction  base = "xs:string" >
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:persistent" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:transient" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:1.1:nameid-
format:unspecified" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:1.1:nameid-
format:emailAddress" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:1.1:nameid-
format:X509SubjectName" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:1.1:nameid-
format:WindowsDomainQualifiedName" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:kerberos" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:entity" />
       < xs :enumeration  value = "urn:oasis:names:tc:SAML:2.0:nameid-
format:encrypted" />
      </ xs :restriction>
     </ xs :simpleType>
    </ xs :element>
```

```
    < xs :element  name = "levelOfAssurance"   type = "xs:anyURI"
minOccurs = "0"   maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en" >
       Level of assurance required to fulfill the request
       Either notified level of assurance matching the following:
       < xs :restriction  base = "xs:string" >
        < xs :enumeration  value = "http://eidas.europa.eu/LoA/low" />
        < xs :enumeration  value = "http://eidas.europa.eu/LoA/substantial"
/>
        < xs :enumeration  value = "http://eidas.europa.eu/LoA/high" />
       </ xs :restriction>
       Or non notified level of assurance being an URI having a different pr
efix than
        http://eidas.europa.eu/LoA
      </ xs :documentation>
     </ xs :annotation>
    </ xs :element>
    < xs :element  name = "status"   minOccurs = "1"   maxOccurs = "1" >
     < xs :annotation>
      < xs :documentation  xml:lang = "en"
>Complex element to provide status information from IdP</ xs :documentation>
     </ xs :annotation>
     < xs :complexType>
      < xs :sequence>
       < xs :element  name = "failure"   type = "xs:boolean"   minOccurs =
"0"   maxOccurs = "1" >
        < xs :annotation>
         < xs
:documentation>Value "true" represents that the authentication request is fa
iled</ xs :documentation>
        </ xs :annotation>
       </ xs :element>
       < xs :element  name = "statusCode"   minOccurs = "0"   maxOccurs =
"1" >
        < xs :annotation>
         < xs :documentation>SAML2 defined status code</ xs :documentation>
        </ xs :annotation>
        < xs :simpleType>
         < xs :restriction  base = "xs:string" >
          < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:status:Success" >
           < xs :annotation>
            < xs :documentation>Authentication success</ xs :documentation>
```

```
            </ xs :annotation>
          </ xs :enumeration>
          < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:status:Requester" >
            < xs :annotation>
             < xs
:documentation>Authentication failure: the requester did something wrong</
xs :documentation>
            </ xs :annotation>
          </ xs :enumeration>
          < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:status:Responder" >
            < xs :annotation>
             < xs
:documentation>Authentication failure: error at the the responder side</ xs :
documentation>
            </ xs :annotation>
          </ xs :enumeration>
          < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:status:VersionMismatch" >
            < xs :annotation>
             < xs
:documentation>Authentication failure: The responder could not process the r
equest because the request message's version was incorrect.</ xs :documentati
on>
            </ xs :annotation>
          </ xs :enumeration>
         </ xs :restriction>
        </ xs :simpleType>
       </ xs :element>
       < xs :element  name = "subStatusCode"  minOccurs = "0"  maxOccurs =
"1" >
        < xs :annotation>
         < xs
:documentation>Optional SAML2 defined sub status code used in case of failur
e</ xs :documentation>
        </ xs :annotation>
        < xs :simpleType>
         < xs :restriction  base = "xs:string" >
          < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:status:AuthnFailed" />
          < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:status:InvalidAttrNameOrValue" />
```

```
          < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:status:InvalidNameIDPolicy" />
          < xs :enumeration  value =
"urn:oasis:names:tc:SAML:2.0:status:RequestDenied" />
         </ xs :restriction>
        </ xs :simpleType>
       </ xs :element>
       < xs :element  name = "statusMessage"  type = "xs:string"
minOccurs = "0"  maxOccurs = "1" >
        < xs :annotation>
         < xs :documentation>An optional status message</ xs :documentation>
        </ xs :annotation>
       </ xs :element>
      </ xs :sequence>
     </ xs :complexType>
    </ xs :element>
    < xs :element  name = "attributes" >
     < xs :complexType>
      < xs :sequence>
       < xs :element  name = "attribute"  minOccurs = "0"  maxOccurs =
"unbounded" >
        < xs :complexType>
         < xs :sequence>
          < xs :element  name = "definition"  type = "xs:string"
minOccurs = "1"  maxOccurs = "1" />
          < xs :element  name = "value"  type = "xs:string"  maxOccurs =
"unbounded" />
         </ xs :sequence>
        </ xs :complexType>
       </ xs :element>
      </ xs :sequence>
     </ xs :complexType>
    </ xs :element>
   </ xs :sequence>
  </ xs :complexType>
 </ xs :element>
</ xs :schema>
```

The previous schema is a copy of the xsd schema which can be found in the *EIDAS-SpecificCommunicationDefinition/src/main/resources/xsds/* folder

6 Appendix B: Examples

6.1 LightToken QED

In section demonstrates the generation of the LightToken in a Linux environment as an example.

Step 0 - Starting from the BASE64 encoded string in section in section 4.4.1

c3BlY2lmaWNDb21tdW5pY2F0aW9uRGVmaW5pdGlvbkNvbm5lY3RvclJlcXVlc3R8ODUyYTY0YzAtOGFjMS00NDVmLWIwZTEtOTkyYWRhNDkzMDMzfDIwMTctMTItMTEgMTQ6MTI6MDUgMTQ4fDdNOHArdVA4Q0tYdU1pMklxU2RhMXRnNDUyV2xSdmNPU3d1MGRjaXNTWUU9

Step 1 - Decode it in BASE64

```
$ echo

c3BlY2lmaWNDb21tdW5pY2F0aW9uRGVmaW5pdGlvbkNvbm5lY3RvclJlcXVlc3R8ODUyYTY0YzAtOGFjMS00NDVmLWIwZTEtOTkyYWRhNDkzMDMzfDIwMTctMTItMTEgMTQ6MTI6MDUgMTQ4fDdNOHArdVA4Q0tYdU1pMklxU2RhMXRnNDUyV2xSdmNPU3d1MGRjaXNTWUU9 | base64 --decode

specificCommunicationDefinitionConnectorRequest|852a64c0-8ac1-445f-
b0e1-992ada493033|2017-12-11 14:12:05 148|
7M8p+uP8CKXuMi2IqSda1tg452WlRvcOSwu0dcisSYE=
```

Step 2 - Create String to be used for calculating digest from id|issuer|timestamp|secret

852a64c0-8ac1-445f-b0e1-992ada493033|specificCommunicationDefinitionConnectorRequest|2017-12-11 14:12:05 148|mySecretConnectorRequest

Step 3 - Calculate digest and Base64 encode it

```
$ printf %s "852a64c0-8ac1-445f-b0e1-992ada493033|
specificCommunicationDefinitionConnectorRequest|2017-12-11 14:12:05 148|
mySecretConnectorRequest" | openssl dgst -sha256 -binary | base64

result:

7M8p+uP8CKXuMi2IqSda1tg452WlRvcOSwu0dcisSYE=
```

Step 4 - Concatenate issuer|id|timestamp|digest

specificCommunicationDefinitionConnectorRequest|852a64c0-8ac1-445f-b0e1-992ada493033|2017-12-11 14:12:05 148|7M8p+uP8CKXuMi2IqSda1tg452WlRvcOSwu0dcisSYE=

Step 5 - BASE64 encode it

```
$ printf %s "specificCommunicationDefinitionConnectorRequest|
852a64c0-8ac1-445f-b0e1-992ada493033|2017-12-11 14:12:05 148|
7M8p+uP8CKXuMi2IqSda1tg452WlRvcOSwu0dcisSYE=" | base64 -w 1024
```

the result will be the same as the one in Step 0.

6.2 Python's Ignite Thin client Specific Connector POC

This section presents an example of a working POC for implementing a Thin Client that sets a LightRequest in the SpecificConnector to Connector Communication Cache and creates and sends the corresponding BLT to the Connector. The code is written in Python, one of the possibilities for Ignite Thin Clients. There is no special reason for this choice, other possibility could have been used.

The main objective for this POC is to facilitate and show another possibility of integration with the eIDAS Node, through a simple example, than the ones already presented in the Demo Specific Connector.

Note that this is not production ready code and should not be used as is. Although it can be used as a basis for developing production ready code, adequate security measures should be also added.

To allow the use of Ignite Thin Clients, it is necessary to enable this possibility. Therefore, in *igniteSpecificCommunication.xml* file as a child of the following bean

```
< bean id = "igniteSpecificCommunication.cfg" class = "org.apache.ignite
.configuration.IgniteConfiguration" >
```

the following property needs to be added,

```
<!-- Thin client connection configuration. -->
< property name = "clientConnectorConfiguration" >
    < bean class = "org.apache.ignite.configuration.ClientConnectorConfigu
ration" >
        < property name = "host" value = "127.0.0.1" />
        < property name = "port" value = "10900" />
        < property name = "portRange" value = "1" />
    </ bean >
</ property >
```

After this step, the Python code that sets a LightRequest in the communication cache and that creates and sends the corresponding Binary Light Token is as follows:

```
# Copyright (c) 2019 by European Commission
#
# Licensed under the EUPL, Version 1.2 or - as soon they will be
# approved by the European Commission - subsequent versions of the
# EUPL (the "Licence");
```

```
# You may not use this work except in compliance with the Licence.
# You may obtain a copy of the Licence at:
# https://joinup.ec.europa.eu/page/eupl-text-11-12
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the Licence is distributed on an "AS IS" basis,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the Licence for the specific language governing permissions and
# limitations under the Licence
from pyignite import Client
import datetime
import requests
import base64
import hashlib
import uuid
#Example to set a LightRequest in the SpecificConnector to Connector Cache
and
# to create and send the corresponding BLT to the Connector.
id = str (uuid.uuid4())
print ( id )
lightRequest = "" "<?xml version=" 1.0 " encoding=" UTF - 8 " standalone=" y
es"?>
<lightRequest xmlns = "http://cef.eidas.eu/LightRequest" >
<citizenCountryCode>CA< / citizenCountryCode>
< id > """ + id + """ "< / id >
<issuer>pythonSpecificConnectorCA< / issuer>
<levelOfAssurance>http: / / eidas.europa.eu / LoA / low< / levelOfAssurance>
<nameIdFormat>urn:oasis:names:tc:SAML: 1.1 :nameid - format :unspecified< /
nameIdFormat>
<providerName>DEMO - SP - CA< / providerName>
<spType>public< / spType>
<requestedAttributes>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson / D - 2012 - 1
7 - EUIdentifier< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson / EORI< /
definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson / LEI< /
definition>
< / attribute>
```

```
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson /
LegalName< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson /
LegalPersonAddress< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson /
LegalPersonIdentifier< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson / SEED< /
definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson / SIC< /
definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson /
TaxReference< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson /
VATRegistrationNumber< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / naturalperson /
BirthName< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / naturalperson /
CurrentAddress< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / naturalperson /
CurrentFamilyName< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / naturalperson /
CurrentGivenName< / definition>
< / attribute>
<attribute>
```

```
<definition>http: / / eidas.europa.eu / attributes / naturalperson /
DateOfBirth< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / naturalperson / Gender<
/ definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / naturalperson /
PersonIdentifier< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / naturalperson /
PlaceOfBirth< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / legalperson /
LegalAdditionalAttribute< / definition>
< / attribute>
<attribute>
<definition>http: / / eidas.europa.eu / attributes / naturalperson /
AdditionalAttribute< / definition>
< / attribute>
< / requestedAttributes>
< / lightRequest>"""
#Connect to cache
client = Client()
client.connect( '127.0.0.1' , 10900 )
specificNodeConnectorRequestCache = client.get_cache( 'specificNodeConnect
orRequestCache' )
#Put Light Request in cache
specificNodeConnectorRequestCache.put( id , lightRequest)
issuer = "specificCommunicationDefinitionConnectorRequest"
now = datetime.datetime.now().strftime( "%Y-%m-%d %H:%M:%S %f" )[: - 3 ]
print (now)
timestamp = str (now)
mySecretConnectorRequest = "mySecretConnectorRequest"
#calculation of digest id|issuer|timestamp|secret
bltForDigest = id + "|" + issuer + "|" + timestamp + "|"
  + mySecretConnectorRequest
print (bltForDigest)
digest = hashlib.sha256(bltForDigest.encode())
print (digest.digest())
digestBase64 = base64.b64encode(digest.digest())
print (digestBase64)
```

```
# BLT to be sent: issuer|id|timestamp|digest
blt = issuer + "|" + id + "|" + timestamp + "|" +
 digestBase64.decode( "utf-8" )
print (blt)
#BLT in Base64
bltBase64 = base64.b64encode(blt.encode())
print (bltBase64)
#send of post with BL
connectorEndpoint = 'http://localhost:8080/EidasNode/
SpecificConnectorRequest'
payload = { 'token' : bltBase64}
response = requests.post(connectorEndpoint, payload)
print (response.text)
```

More information can be found at the following URLs:

- https://apacheignite.readme.io/docs/binary-client-protocol
- https://apacheignite.readme.io/docs/python-thin-client

6.3 Ignite's Rest API

As mentioned in https://apacheignite.readme.io/docs/rest-api,
" Ignite provides an HTTP REST client that gives you the ability to communicate with the grid over HTTP and HTTPS protocols using the REST approach. REST APIs can be used to perform different operations like read/write from/to cache, execute tasks, get various metrics and more."

Therefore, it is possible to enable it adding for example a maven profile in eIDAS to e.g. the pom.xml of EIDAS-JCache-Ignite as follows:

```
< profiles >
    < profile >
        < id >enableIgniteRest</ id >
        < activation >
            < activeByDefault >false</ activeByDefault >
        </ activation >
        < dependencies >
            < dependency >
                < groupId >org.apache.ignite</ groupId >
                < artifactId >ignite-rest-http</ artifactId >
                < version >${ignite.version}</ version >
            </ dependency >
        </ dependencies >
    </ profile >
</ profiles >
```

Note however, that if enabling this profile, adds several dependencies which can be affected by CVEs, which were not analysed for impact in eIDAS Node's code. Another solution is to use the possibility described at B.2 Python's Ignite Thin client Specific Connector POC, which does not need more dependencies to be included.

7 Appendix C: Ignite advanced configurations

7.1 SSL/TLS

As mention in https://apacheignite.readme.io/docs/ssltls:

"Ignite allows you to use SSL socket communication to provide a secure connection among all Ignite nodes. To use it, set the Factory<SSLContext> and configure the SSL section in the Ignite configuration. Ignite provides a default SSL context factory, *org.apache.ignite.ssl.SslContextFactory*, which uses a configurable keystore to initialize the SSL context. "

In eidas configuration files files *server.p12*, *trust.p12* need to be included at external configuration folders, more specifically at:

EidasConfig/server/ignite/KeyStore

to demo the enabling of SSL/TLS in Ignite

However, you still need to add in the files

- igniteNode.xml
- igniteSpecificCommunication.xml

the below property sslContextFactory under bean *igniteNode.cfg* as depicted below:

```
< beans   xmlns = "http://www.springframework.org/schema/beans"   xmlns:xsi =
http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation = "http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd" >
  < bean   id = "igniteNode.cfg"   class = "org.apache.ignite.configuration.
IgniteConfiguration" >
    ...
    <!-- Ssl/Tls context. -->
    <!--IMPORTANT: THIS IS JUST A DEMO CONFIGURATION AND DEMO KEYSTORES,
NEEDS TO BE CHANGED FOR PRODUCTION ALSO THE KEYS IN THE KEYSTORES NEED TO BE
CREATED AS WELL-->
    < property   name = "sslContextFactory" >
      < bean   class = "org.apache.ignite.ssl.SslContextFactory" >
        <!--uncomment the below keyAlgorithm when IBM JRE is used e.g.
websphere 8.5.5-->
        <!--< property   name = "keyAlgorithm"   value = "IBMX509"  />-->
        < property   name = "keyStoreFilePath"   value = "$
{EIDAS_CONFIG_REPOSITORY}/ignite/KeyStore/server.p12"  />
        < property   name = "keyStorePassword"   value = "123456"  />
        < property   name = "trustStoreFilePath"   value = "$
{EIDAS_CONFIG_REPOSITORY}/ignite/KeyStore/trust.p12"  />
        < property   name = "trustStorePassword"   value = "123456"  />
        < property   name = "protocol"   value = "TLSv1.2"  />
      </ bean >
```

```
    </property>
  </bean>
</beans>
```

Note that this configuration is a demo configuration done using the information on [https://apacheignite.readme.io/docs/ssltls|https://apacheignite.readme.io/docs/ssltls]. Therefore, this should not be used as is in production and all the necessary measures to ensure the needed levels of security should be done,e.g. create new PrivateKeyEntries, trusted entries in the keystores, etc.

8 Appendix D: Ignite Proposed Configuration

As an example follows the demo file for *igniteSpecificCommunication.xml*:

```
< beans   xmlns = "http://www.springframework.org/schema/beans"
        xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd ">

    < bean   id = "igniteSpecificCommunication.cfg"   class = "org.apache.ig
nite.configuration.IgniteConfiguration" >

        < property   name = "igniteInstanceName"   value = "igniteSpecificCo
mmunication" />

        < property   name = "cacheConfiguration" >
            < list >

                <!--Specific Communication Caches-->
                <!-- Partitioned cache example configuration (Atomic mode).
-->
                < bean   class = "org.apache.ignite.configuration.CacheConfi
guration" >
                    < property   name = "name"   value = "specificNodeConnec
torRequestCache" />
                    < property   name = "atomicityMode"   value = "ATOMIC" /
>
                    < property   name = "backups"   value = "1" />
                    < property   name = "expiryPolicyFactory"   ref = "7_min
utes_duration" />
                </ bean >
                <!-- Partitioned cache example configuration (Atomic mode).
-->
                < bean   class = "org.apache.ignite.configuration.CacheConfi
guration" >
                    < property   name = "name"   value = "nodeSpecificProxys
erviceRequestCache" />
                    < property   name = "atomicityMode"   value = "ATOMIC" /
>
                    < property   name = "backups"   value = "1" />
                    < property   name = "expiryPolicyFactory"   ref = "7_min
utes_duration" />
                </ bean >
```

```
                <!-- Partitioned cache example configuration (Atomic mode).
-->
                < bean   class = "org.apache.ignite.configuration.CacheConfi
guration" >
                    < property   name = "name"   value = "specificNodeProxys
erviceResponseCache" />
                    < property   name = "atomicityMode"   value = "ATOMIC" /
>
                    < property   name = "backups"   value = "1" />
                    < property   name = "expiryPolicyFactory"   ref = "7_min
utes_duration" />
                </ bean >
                <!-- Partitioned cache example configuration (Atomic mode).
-->
                < bean   class = "org.apache.ignite.configuration.CacheConfi
guration" >
                    < property   name = "name"   value = "nodeSpecificConnec
torResponseCache" />
                    < property   name = "atomicityMode"   value = "ATOMIC" /
>
                    < property   name = "backups"   value = "1" />
                    < property   name = "expiryPolicyFactory"   ref = "7_min
utes_duration" />
                </ bean >

            </ list >
        </ property >

        <!--Explicitly configure TCP discovery SPI to provide list of
initial nodes from the second cluster.-->
        < property   name = "discoverySpi" >
            < bean   class = "org.apache.ignite.spi.discovery.tcp.TcpDiscove
rySpi" >
                <!-- Initial local port to listen to. -->
                < property   name = "localPort"   value = "49500" />

                <!-- Changing local port range. This is an optional action.
-->
                < property   name = "localPortRange"   value = "3" />

                <!-- Setting up IP finder for this cluster -->
                < property   name = "ipFinder" >
                    < bean   class = "org.apache.ignite.spi.discovery.tcp.ip
finder.vm.TcpDiscoveryVmIpFinder" >
                        < property   name = "addresses" >
```

```
                                < list >
                                    <!--
                                    Addresses and port range of the nodes from
the second cluster.
                                    127.0.0.1 can be replaced with actual IP
addresses or host names. Port range is optional.
                                    -->
                                    < value >127.0.0.1:49500..49502</ value >
                                </ list >
                            </ property >
                        </ bean >
                    </ property >
                </ bean >
            </ property >

            <!--
            Explicitly configure TCP communication SPI changing local port
number
            for the nodes from the second cluster.
            -->
            < property   name = "communicationSpi" >
                < bean   class = "org.apache.ignite.spi.communication.tcp.TcpCom
municationSpi" >
                    < property   name = "localPort"   value = "49100" />
                </ bean >
            </ property >

            <!-- Ssl/Tls context. -->
            <!--IMPORTANT: THIS IS A DEMO CONFIGURATION AND DEMO KEYSTORES, IT
NEEDS TO BE CHANGED FOR PRODUCTION ALSO THE KEYS IN THE KEYSTORES NEED TO BE
CREATED AS WELL-->
            < property   name = "sslContextFactory" >
                < bean   class = "org.apache.ignite.ssl.SslContextFactory" >
                    <!--uncomment the below keyAlgorithm when IBM JRE is used
e.g. websphere 8.5.5-->
                    <!--< property   name = "keyAlgorithm"   value = "IBMX509"
/>-->
                    < property   name = "keyStoreFilePath"   value = "$
{EIDAS_CONFIG_REPOSITORY}/ignite/KeyStore/server.p12" />
                    < property   name = "keyStorePassword"   value = "123456" />
                    < property   name = "trustStoreFilePath"   value = "$
{EIDAS_CONFIG_REPOSITORY}/ignite/KeyStore/trust.p12" />
                    < property   name = "trustStorePassword"   value = "123456"
/>
```

```
                < property  name = "protocol"  value = "TLSv1.2" />
            </ bean >
        </ property >

        <!-- how frequently Ignite will output basic node metrics into the
log-->
        < property  name = "metricsLogFrequency"  value = "#{60 * 10 *
1000}" />

    </ bean >

    <!--
        Initialize property configurer so we can reference environment
variables.
    -->
    < bean  id = "propertyConfigurer"  class = "org.springframework.beans.
factory.config.PropertyPlaceholderConfigurer" >
        < property  name = "systemPropertiesModeName"  value = "SYSTEM_PRO
PERTIES_MODE_FALLBACK" />
        < property  name = "searchSystemEnvironment"  value = "true" />
    </ bean >

    <!--
        Defines expiry policy based on moment of creation for ignite cache.
    -->
    < bean  id = "7_minutes_duration"  class = "javax.cache.expiry.Created
ExpiryPolicy"  factory-method = "factoryOf"  scope = "prototype" >
        < constructor -arg>
            < bean  class = "javax.cache.expiry.Duration" >
                < constructor -arg  value = "MINUTES" />
                < constructor -arg  value = "7" />
            </ bean >
        </ constructor -arg>
    </ bean >
</ beans >
```

9 Appendix E: Message Logging Features

To allow obtaining the NodeId for incoming Light Request and Light Responses, the following parameters were added to *server\specificConnector\specificCommunicationDefinitionConnector.xml*:

Key	Description
lightToken.connector.request.node.id	Id that identifies the sender of the Light Request to the Connector
lightToken.connector.response.node.id	Id that identifies the receiver of the Light Response to the Connector

and to *server\specificProxyService\specificCommunicationDefinitionProxyservice.xml*

Key	Description
lightToken.proxyservice.request.node.id	Id that identifies the receiver of the Light Request to the Connector
lightToken.proxyservice.response.node.id	Id that identifies the sender of the Light Response to the Connector

These values are loaded into instances of SpecificConnectorCommunicationServiceExtensionImpl and SpecificProxyserviceCommunicationServiceExtensionImpl which can then be used to populate the NodeId when performing of message logging.