eIDAS-Node National IdP and SP Integration Guide

Version 1.0

Document history

Version	Date	Modification reason	Modified by
1.0	16/10/2017	Origination	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.

Table of contents

List of figures

List of tables

1. Introduction

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

eID can be integrated into your national eID infrastructure in a number of ways. The purpose of this document is to provide guidance by recommending one way in which it can be done.

The advantages of adopting this approach are:

- Sustainability
- Greater Security
- Better Scalability
- More Flexibility

1.1. Purpose

The purpose of this document is to describe how Member States can integrate the eIDAS-Node into their national infrastructure systems

1.2. Document aims

The aims of this document are to:

- provide information on customisation of attributes;
- provide details of how to develop and tailor the Specific parts for your country;
- provide information on how to integrate with a SAML infrastructure;
- describe implementation into other web-based infrastructure; and
- 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 – *Developing MS-Specific Parts* describes considerations of the eIDAS-Node architecture to be taken into account when developing your integration strategy.
- Chapter 3 – *Integration possibilities* describes the recommended integration approaches, starting with that requiring the least changes.
- Appendix A – *Diagrams* contains diagrams covering some parts of the software architecture that are mentioned in this document.

1.4. Other technical reference documentation

We recommend that you also familiarise yourself with the following eID technical reference documents which are available on **CEF Digital Home > eID > All eID services > eIDAS Node integration package > eIDAS-Node software releases > Current release**:

- *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.
- *CEF eID 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.

2. Developing MS-Specific Parts

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

2.1. eIDAS-Node Connector and eIDAS-Node Proxy Service in one product

The delivered or custom built `EidasNode.war` web application contains functionalities of both eIDAS-Node Connector and eIDAS-Node Proxy Service. The actual role is 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.

2.2. Required profile and flow of control

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.

For consuming foreign IDs, the eIDAS-Node Connector will require a browser hit (HTTP GET or POST) from an SP located in the national infrastructure. After some basic processing, the request will be forwarded to the MS-Specific code part by Java calls. This includes the propagation of the Java representation of the HTTP request, from where the MS-Specific part needs to construct a Java object (`LightRequest`) to the eIDAS-Node Connector.

The eIDAS-Node Connector then constructs the eIDAS SAML Request from data provided by the Java object, places it into the HTTP Request and then issues a redirect command for the client browser. The redirect target is the eIDAS-Node Proxy Service of the citizen's country.

The eIDAS SAML Request is processed by the eIDAS-Node Proxy Service through the optional consent pages, then converted into a Java object (`LightRequest` again). The MS-Specific part of the eIDAS-Node Proxy Service receives this object along with the HTTP Request/Response context, so it can implement the required specific action.

The above communication pattern is the architectural solution for delivering the Response also. Note that in this case the Java object will be `LightResponse` instead of `LightRequest`.

2.3. Use of the provided MS-Specific sample implementation

There is a sample MS-Specific implementation provided with the eIDAS-Node package. This is provided as a demonstration for the Basic Setup, capable of working together with Demo SP and Demo IdP. The code can be found in the eIDAS-Specific module. It is compiled and built into a JAR file called `eidas-specific.jar`. This JAR file is then placed into the libraries folder of the main `EidasNode.war` application during the build process.

eIDAS by definition is not suitable for national authentication. The provided sample module uses the eIDAS protocol in the whole authentication process which SHOULD NOT be the case in a real production environment.

The eIDAS protocol is intended for the purposes of cross-border authentication. Any use of it in national infrastructure is not advised and would not be supported.

CEF eID software is provided as a sample implementation to meet the goals and strategy of the Technical Specifications for cross border authentication. Any modifications of the code for other purposes would result in software that is not eIDAS compliant. This would cause compatibility problems as the Technical Specifications evolve and further iterations of the CEF eID software are released.

3. Integration possibilities

This section describes the recommended integration approaches, starting with that requiring the least changes.

3.1. Implementing into SAML infrastructure

You should read this section if your national infrastructure is based on SAML, and you wish to implement an eIDAS-Node directly to the existing network. In this case, the provided sample MS-Specific can be reused with minimal changes.

3.1.1. Setting up custom attributes

By default, the EIDAS-SAMLEngine module only supports the attributes included in eIDAS core dataset (see the document *eIDAS SAML Attribute Profile* from the eIDAS Technical Sub-group).

3.1.2. Attribute registry

Attribute registry is responsible for holding and supplying information of types, value format and namespace for creating and validating requests and responses. The registry basically contains Attribute Definition objects built from custom XML files or a hard-coded list of supported core attributes in:

- `LegalPersonSpec`
- `NaturalPersonSpec`
- `RepresentativeLegalPersonSpec`
- `RepresentativeNaturalPersonSpec`.

The hard coded list is necessary to get a reference of attribute definitions to perform business rule-based validations on requests and replies.

The core attributes come from the hardcoded set, but this can be partly overridden by `coreAttributeRegistryFile` set in SAML engine configuration (see how `saml-engine-eidas-attributes.xml` is used in `SPSamlEngine.xml`). This might help in testing and implementing aspects of the MS-Specific part, but should not be used in production for the eIDAS-Node itself.

There is another XML file specified by the optional `additionalAttributeRegistryFile` parameter in the SAML Engine configuration. This file can be used to support additional attributes, sometimes referred as dynamic attributes (the sample filename is `saml-engine-additional-attributes_EidasNode.xml`). Each SAML Engine instance can have different configurations specified in the `SamlEngine.xml` files.

The following is an example of code to introduce a new attribute to the XML configuration:

```
    <entry
key="19.NameUri">http://eidas.europa.eu/attributes/natural/NewSomething</entry>
    <entry key="19.FriendlyName">NEW_SOMETHING</entry>
    <entry key="19.PersonType">NaturalPerson</entry>
    <entry key="19.Required">false</entry>
```

```
    <entry
key="19.XmlType.NamespaceUri">http://eidas.europa.eu/attributes/naturalperson</entr
y>
    <entry key="19.XmlType.LocalPart">NewSomethingType</entry>
    <entry key="19.XmlType.NamespacePrefix">eidas-natural</entry>
    <entry
key="19.AttributeValueMarshaller">eu.eidas.auth.commons.attribute.impl.LiteralStrin
gAttributeValueMarshaller</entry>
```

For the `key` prefix number, take the last one and increment it. For eIDAS protocol the person type (`NaturalPerson`, `LegalPerson`, `RepresentativeNaturalPerson` or `RepresentativeLegalPerson`) must be specified and aligned with namespace.

3.1.3. Attribute registry validation and metadata support

The list of supported attributes is published in the eIDAS-Node Proxy Service Metadata. The list includes all the eIDAS core and additional attributes, specified in the additional attributes XML file mentioned above.

3.1.4. ProtocolProcessor/ProtocolEngine implementation

The `ProtocolProcessor` needs to be implemented to apply special validation and processing rules of the SAML message format, because the `AbstractProtocolEngine` is designed to support a generic SAML2 profile. The base can be copied from `EidasProtocolProcessor`.

To utilise your `ProtocolProcessor`, the configuration file named `SpecificSamlEngine.xml` must be modified: `ProtocolProcessorConf/class` must point to the exact implementing class.

However, if there are extra SAML features used, it is possible that classes of `AbstractProtocolEngine` and/or other parts needs to be amended. In this case it is recommended to copy the modified classes into the MS-Specific part of the code (by package namespace and by physical location to the specific JAR too).

To use your `ProtocolEngine`, the `ProtocolEngineFactory` and `ProtocolEngineConfiguration/ProtocolEngineConfigurationFactory` should be copied or customised also, so in this case the EIDAS-SAMLEngine module should be copied to a new namespace.

These are all referenced from Spring application context `specificApplicationContext.xml`, therefore the references need to be updated.

3.1.5. National scheme of attributes

The `ProtocolEngine` manages the AttributeRegistries. The national set of attributes must be introduced to the MS-Specific `ProtocolEngine`. It can be done via a programmatic AttributeRegistry or via XML files.

If the mapping between eIDAS and your national scheme is straight-forward, and can be carried out by the 'derivation' sample feature you can find in the sample MS-Specific, then adding an XML file should be enough. To do so, define a `saml-engine-<mycountryspecific>-attributes.xml` and reference it

'SpecificSamlEngine.xml' (coreAttributeRegistryFile). Then make your derivation rules accordingly.

If you need to perform special actions when translating attributes, like validation, concatenation or similar, then you need to reference them in Java, so you must also add your attributes using the programmatic way. To do this, just replicate the code hierarchy coming with EidasSpec class.

3.1.6. Implementing Metadata

The Metadata generation/consumption for the MS-Specific part may need to be changed, not to consume or produce eIDAS-format metadata.

If your national metadata does not include custom extensions, the MetadataFetcher can be used without modifications. If there are custom elements, the MetadataUtil can be extended to add any extra accessors.

On Metadata provision the EidasNodeMetadataGenerator class is used directly from the Servlet. It provides metadata capabilities according to eIDAS Metadata format, so if there is a need to customise it, a similar class can be implemented. The beans connectorMetadataGeneratorIDP and serviceMetadataGeneratorSP need to be defined in Spring context file of specificApplicationContext.xml.

3.1.7. Implementing specific service layer

If there are specific validations needed for your SAML protocol on service level, the sample EIDAS-Specific code can be modified.

There are several classes that might need to be changed:

- SpecificEidasService and SpecificEidasConnector – these classes contain the details of the Member State's specific implementation.
- SpecificProxyServiceImpl – the class that is responsible for converting the LightRequest coming from the eIDAS Proxy Service into the IdP's specific protocol and the external IdP response into the LightResponse format.
- SpecificConnectorImpl – The class that is responsible for converting the SP's request into the LightRequest and the LightResponse into the SP's response specific protocol.

If your national network contains multiple IdPs to serve the eIDAS-Node Proxy Service, you will need to implement an IdP selector logic to SpecificEidasService and/or SpecificProxyServiceImpl.

3.1.8. Authentication

When you have only one IdP, or there is an HTTP redirection-routing component, the external.authentication must be set to "**true**", and the URL of the IdP (idp.url) needs to be set in eidas_Specific.xml. Otherwise the above mentioned customisation in the service layer implementation is required.

3.2. Implementing into other web-based infrastructure

You should read this section if your national infrastructure is not based on SAML, but is Web Profile-based, and you choose to implement the eIDAS-Node directly to your network.

3.2.1. ProtocolEngine implementation

The supplied `ProtocolProcessor` implementation supports only SAML, however the `ProtocolEngineI` interface is probably generic enough for most Web Profile-based authentication protocols. Implementing your own ProtocolEngine lets you reuse some parts of the provided sample MS-Specific:

- With some modifications, the `ProtocolEngineFactory` and `ProtocolEngineConfiguration/Factory` can be reused, so the design of the application can remain the same;
- Some parts of the service layer can be re-used.

3.2.2. Implementing specific service layer

Please follow the instructions in section 3.1.7 — *Implementing specific service layer*.

3.3. Implementing without re-using sample MS-Specific

There is a way to implement your specific part or an isolated component by realizing interfaces defined for communicating with MS-Specific parts.

In this case you can exclude EIDAS-Specific from your build and implement `ISpecificConnector` and `ISpecificProxyService` interfaces defined in module EIDAS-SpecificCommunicationDefinition. And also the interfaces `IAUConnector` and `IAUService` may have to be implemented also.

The implemented classes corresponding `ISpecificConnector` and `ISpecificProxyService` must be injected to `connectorController` (at `specificConnector)` and `serviceController` (at `specificProxyService)` beans with Spring `applicationContext.xml`.

For the implementing classes of interfaces `IAUConnector` and `IAUService`, these should be injected in `specificConnectorNode` and `specificServiceNode` respectively. These will be used by `springManagedSpecificConnector` and `springManagedSpecificProxyService` beans at Spring `applicationContext.xml`.

The definition of these interface classes contain detailed information on usage in comments. The HTTP Request/Response context is always propagated, and the actual data is exchanged via objects implemented with `ILightRequest` and `ILightResponse` interfaces.

To populate `ILightRequest` and `ILightResponse` with transfer objects, the EIDAS-Commons and EIDAS-LightCommons should be used.

Table 1: Interface objects

Parameters	Description
`ILightRequest lightRequest`	The data transfer object exchanged between the eIDAS protocol and the specific protocol
`HttpServletRequest httpServletRequest`	The HTTP Servlet Request
`HttpServletResponse httpServletResponse`	The HTTP Servlet Response

Appendix A. Diagrams

A Software Architecture Document (SAD) is planned to help the implementation and integration, meanwhile this section contains diagrams covering some parts already mentioned in this document.

A.1 Protocol Engine

The Protocol Engine is the heart of all protocol related operations in the eIDAS-Node. `ProtocolEngineI` is a generic interface for realising an eIDAS-enabled protocol over primitives and components. The `ProtocolEngine` class encapsulates four major components responsible for constructing, validating and interpreting authentication messages. These are:

- Protocol Processor: responsible for creating and parsing binary messages (eIDAS SAML), encapsulates supported Attribute Registry;
- Protocol Signer: offers signing and signature validation services for Engine, containing certificate/key used for signing messages (but not for validation);
- Protocol Cipher: composition of services performing message encryption and decryption, contains certificates/keys used for decryption, encryption key/certificate must be supplied; and
- Engine Clock: provides current date/time information for messaging.

The following diagram shows the conceptual hierarchy and associations between classes aggregated in the context of the Protocol Engine:

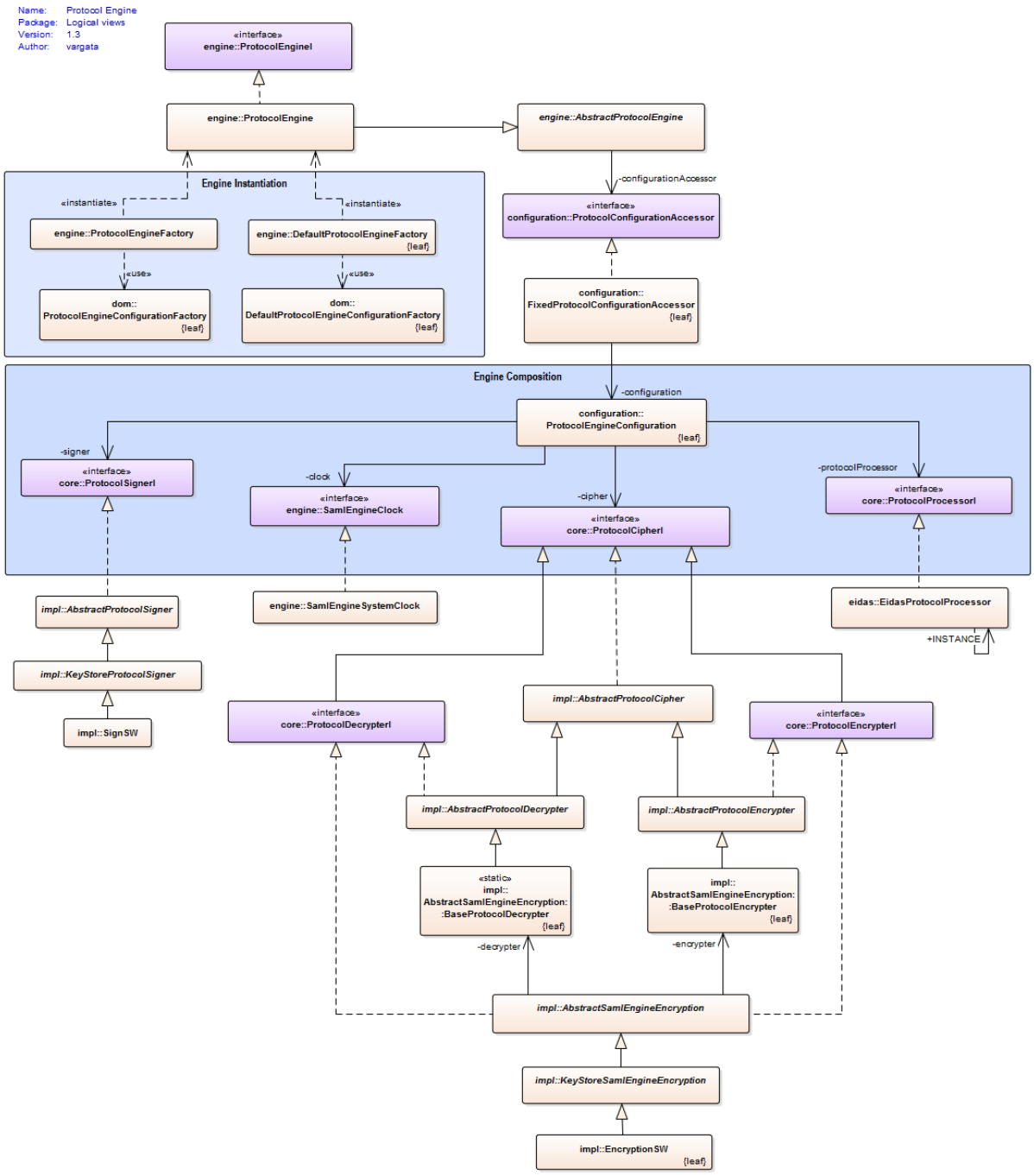

Figure 1: ProtocolEngine — conceptual hierarchy and associations between classes

A.2 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:

1. `NameUri` : [mandatory]: the name URI of the attribute (full name and must be a valid URI)
2. `FriendlyName` : [mandatory]: the friendly name of the attribute (short name)
3. `PersonType` : [mandatory]: either `NaturalPerson`, `LegalPerson`, `RepresentativeNaturalPerson` or `RepresentativeLegalPerson` .
4. `Required` : [optional]: whether the attribute is required by the specification (and is part of the minimal data set which must be requested).
5. `TransliterationMandatory` : [optional]: whether the attribute values must be transliterated if provided in non `LatinScript` variants.
6. `UniqueIdentifier` : [optional]: whether the attribute is a unique identifier of the person (at least one unique identifier attribute must be present in authentication responses).
7. `XmlType.NamespaceUri` : [mandatory]: the XML namespace URI for the attribute values, for example: `http://www.w3.org/2001/XMLSchema` for an XML Schema string
8. `XmlType.LocalPart` : [mandatory]: the name of the XML type for the attributes values, for example: `'string'` for an XML Schema string
9. `XmlType.NamespacePrefix` : [mandatory]: the name of the XML namespace prefix for the attributes values, for example: `'xs'` for an XML Schema string
10. `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`.

A.2.1 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 comes with a `saml-engine-`

`additional-attributes.xml` example file configured for the specific `ProtocolEngine` instance.

A.2.2 Class related attribute registries

The Demo Tools use file/memory based registries instead of hard coded.

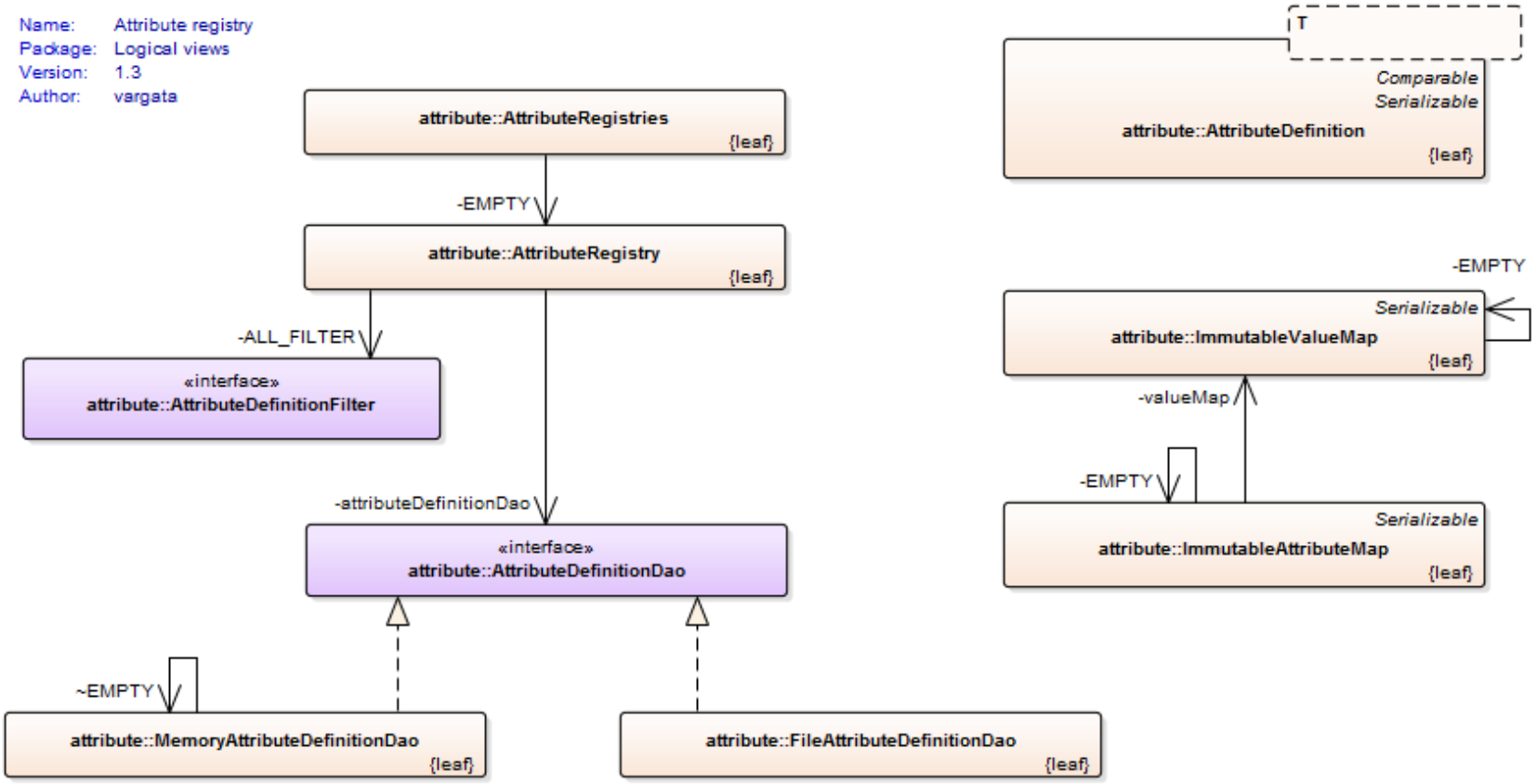

Figure 2: Classes related to basic attribute registry

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, what 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).

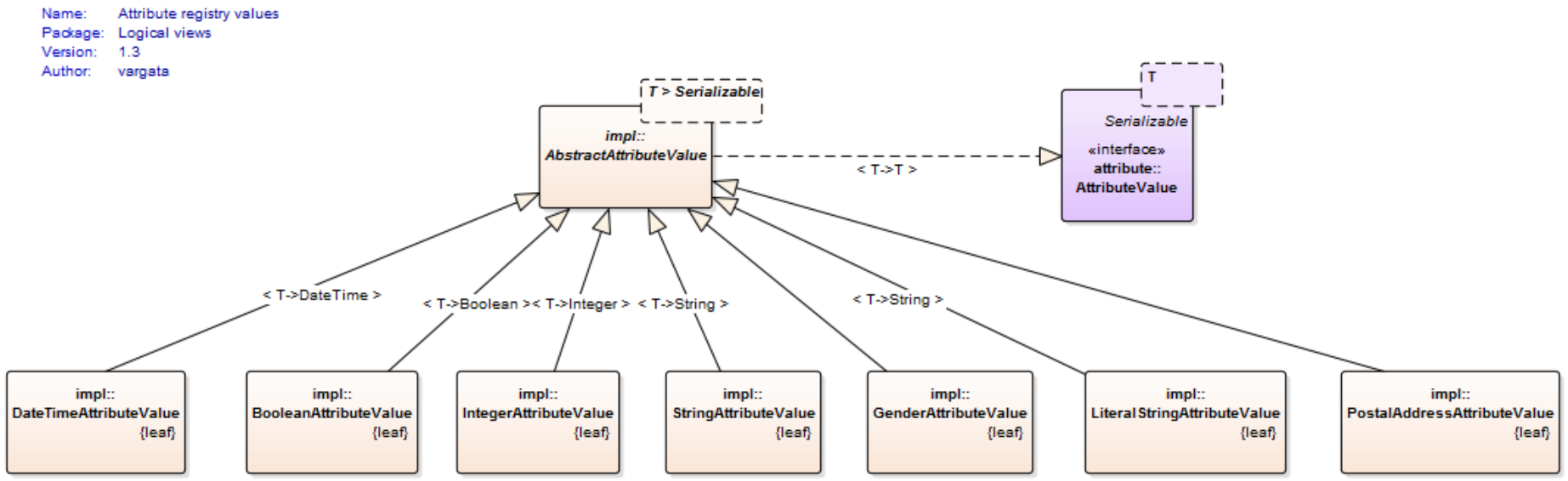

Figure 3: Attribute registry values

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:

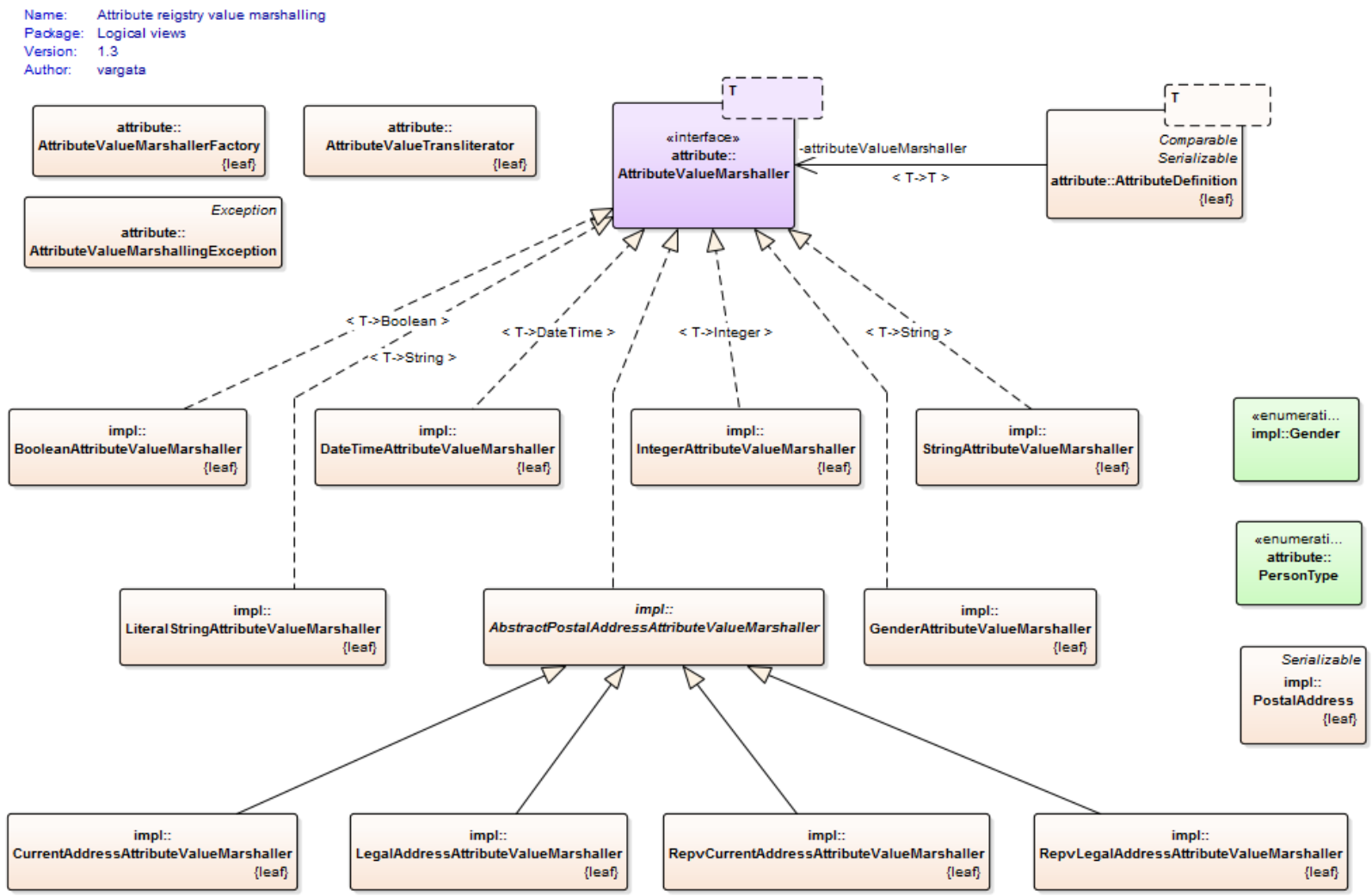

Figure 4: Attribute registry value marshalling

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