Fraunhofer ISST

FRAUNHOFER INSTITUTE FOR SOFTWARE AND SYSTEMS ENGINEERING

ISHARE TRUST FRAMEWORK ADOPTION

Integration into the Dataspace Protocol Stack

open

ADOPTION PLAN

For the Integration of iSHARE Onboarding, Authentication and Authorization Flows

Niklas Schulte

Ronja Quensel

Julia Pampus

Fraunhofer Institute for Software and Systems Engineering
in Dortmund.

Project number: 202584-44425
Date: 31.07.2025

open

Content

1 Introduction

This document outlines the integration of the iSHARE Trust Framework into the Eclipse Dataspace Components[1] (EDC) as an overlay for the Dataspace Protocol (DSP). With the integration of the iSHARE Trust Framework, the DSP flows can be embedded into a legal framework with predefined iSHARE licenses and policies.

Please note that this adoption plan does not include the mapping of iSHARE licenses and policies to Data Catalogue Vocabulary (DCAT) and Open Digital Rights Language (ODRL) and only focusses on the technical integration of the registration, authentication, and authorization flows of the iSHARE Trust Framework into the EDC.

1.1 Integration of Trust Frameworks in Dataspaces

Trust Frameworks play a fundamental role for Dataspaces, in defining how trust is established between participants. In the DSP stack, the DSP sits at the bottom layer of the architecture, defining the Dataspace connector interactions, including the discovery of data offers, their negotiation ending in contract agreements and the initialization of data transfers based on these agreements. On top of this, an identity layer is used. Due to the modular construction of the technology stack, this could either be the Decentralized Claims Protocol (DCP) or realized with a central identity provider, e.g., via the OAuth2 protocol.

Figure 1 shows the architectural layers of the DSP stack. Trust Frameworks are positioned in the top layer of the architecture to embed these processes into a legal framework and to provide an additional governance layer, defining the methods used for authentication and authorization. The iSHARE Trust Framework provides such a legal framework and offers the possibility to delegate access control. This way, technical processes can be established for fine-grained access control patterns which are delegated by a third party.

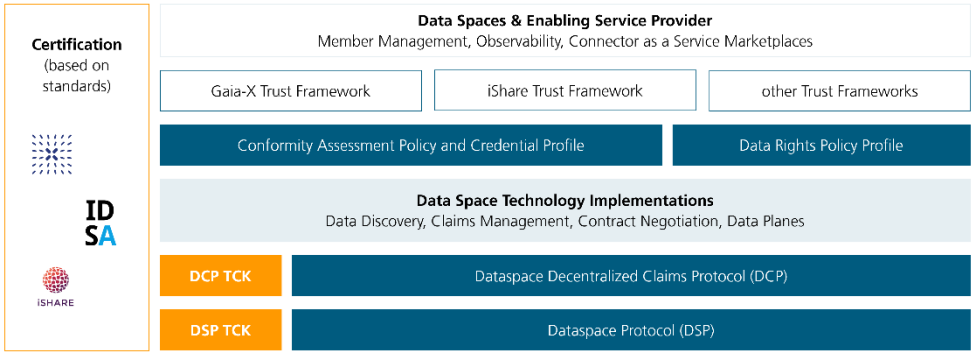

Figure 1: Trust Frameworks in the DSP stack

[1] https://github.com/eclipse-edc

1.1.1 Dataspace Technologies

Dataspaces comprise various technical components that provide capabilities to offer data, discover data offerings, negotiate data usage conditions (i.e. policies), and orchestrate data transfers. With mechanisms like authentication and authorization, observability and the adaptability to domain-specific use cases, e.g., by including custom vocabularies, they build the technical foundation for trust in data ecosystems.

The key component is the dataspace participant agent, namely the *Dataspace Connector* (or Connector), serving as the gateway of services into the dataspace. In recent years, the Connector has become an integral component that is often implemented as one single software component.

Considering data sharing at a higher level of abstraction, the services, whether one or more, are divided into two mental models: Control Plane and Data Plane. Acting as a coordinating layer, services on the *Control Plane* handle messages and oversee the local state of the transfer process. The data itself is transferred on the *Data Plane*, using a wire protocol. In a data sharing scenario, all organizations maintaining the role of dataspace participants operate services that are logically classified as either Control Plane or Data Plane services.

1.1.2 Eclipse Dataspace Components

The dataspace-specific interactions mainly take place on the Control Plane, covering multiple services which interfaces are specified by the DSP. A sample implementation of these components and interactions is provided by the EDC. The EDC is an open-source framework for sovereign, inter-organizational data sharing while implementing relevant dataspace communication and trust principles following the work by IDSA[1] and Gaia-X[2]. The project provides a set of modular components that enable secure data sharing, manage organizational credentials, and automate data usage agreements. EDC serves a variety of use cases, including API access and data processing, while ensuring compliance with established standards.

Within the EDC, the Control Plane services are combined to one *EDC Control Plane* service. The same applies to the Data Plane, resulting in an *EDC Data Plane* service. The Control Plane interactions follow the DSP and DCP specifications, but the interactions on Data Plane layer are implementation-specific and partly undocumented. When using the terms of Control Plane and Data Plane in the following document, we refer to the meaning defined by the EDC project.

1.2 Scope of This Document

This adoption plan is separated into three steps, outlining the connection points between the iSHARE Trust Framework and the DSP. Please note that Authentication and Authorization are two distinct steps in the iSHARE framework and are dependent on information which is stored in two different registries.

1. **iSHARE Adherence:** registration as an iSHARE participant and confirmation of the iSHARE participant status at the iSHARE Participant Registry.

[1] https://internationaldataspaces.org

[2] https://gaia-x.eu

2. **iSHARE Authentication:** authentication based on X.509 certificates or Decentralized Identifiers (DIDs) and Verifiable Credentials (VCs).
3. **iSHARE Authorization:** authorization of participants based on delegation evidence tokens obtained from the iSHARE Authorization Registry.

Implementation Plan (WP1) and Implementation Plan (WP2) detail implementation plans, outlining the implementation steps necessary to connect the EDC to the iSHARE Trust Framework. We only consider the **iSHARE machine-to-machine (M2M) flows** in this document, as the DSP flows are not designed for human interaction.

As we put our focus on establishing trust in dataspaces (cf. Section 1.1), please note that this document only considers the integration of the iSHARE Trust Framework into the Control Plane flows. If required, the iSHARE Trust Framework and token handling could also be used for data/service access on the Data Plane.

1.3 Definition of Dependencies and Work Packages

This adoption plan is split into two separate Work Packages (WPs) called WP1 and WP2, differing in the version of the iSHARE Trust Framework and thereby the identity layer used to ensure backwards compatibility with different versions of the iSHARE Trust Framework.

- **WP1**: an OAuth2-based identity paradigm using X.509 certificates for authentication. WP1 integrates EDC version 0.13.0 into the current version of the iSHARE Trust Framework (v2.2).
- **WP2**: decentralized identities together with VCs, which are transferred via the DCP presentation and issuance flows. WP2 integrates EDC version 0.13.0 into the newest, unreleased version of iSHARE Trust Framework version (v3.0). As iSHARE v3.0 is still unreleased, we base our integration concepts for this WP on the current content of RFC-40[1] regarding the integration of VCs.

1.4 Prerequisites

The following prerequisites are required for the adoption plan in this document.

- **Participant Registry (PR):** there is a least one participant registry, which can be queried to register participants and check for the participant status. For WP1, the PR maintains the list of trusted Certificate Authorities (CAs). For WP2, the PR acts as a trusted issuer for membership credentials.
- **Authorization Registry (AR):** there is at least one authorization registry in place which can be queried to create delegation policies and fetch delegation evidence.
- **Wallet Requirements:** for WP2 each EDC instance needs to operate a DCP-compliant wallet. The EDC Identity Hub may be used, for which we provide further configurations in Section 0.

1.5 Nomenclature

- **EDC**: Eclipse Dataspace Components
- **DSP**: Dataspace Protocol

[1] https://gitlab.com/ishare-foundation/cab/rfc/-/blob/rfc040-verifiable-credentials/RFC Documents/RFC040/README.md?ref_type=heads

- **DCP**: Decentralized Claims Protocol
- **DCAT**: Data Catalog Language
- **ODRL**: Open Digital Rights Language

2 iSHARE Participant Registration

The Participant Registry acts as the trust anchor in the iSHARE Trust Framework. It plays a vital role in the onboarding of participants in the Dataspaces, where it checks for compliance, determining if a party can be admitted to the Dataspace. Typically, this role will be fulfilled by the Dataspace Governance Authority. The Participant Registry can be queried at any time to verify the participant status of a Dataspace member.

To check the adherence to the iSHARE Trust Framework, the participant status of every member must be mutually confirmed on a peer-to-peer basis before any DSP interaction. The conformance of an active participant status is checked during the authentication process, as described in Section 3.

2.1 Onboarding at the Participant Registry (WP1)

The registration as an iSHARE participant is a prerequisite to all subsequent steps. For most of the use cases, we assume that the legal entity on which the EDC acts on behalf of is already registered as iSHARE participant. In this case, no further registration steps need to be performed by the EDC as it can be provided with all necessary information during startup.

Alternatively, the registration process can also be accomplished during onboarding by the EDC itself. Figure 2: Registration Flow for WP1 shows the process of an automated onboarding of the EDC at the Participant Registry. Before this process can be accomplished, the EDC needs to be provided with an X.509 certificate to sign the token request in step 1. After successful registration in step 6, a *parties_token* is issued, which contains all necessary details for further authorization steps. Figure 2 outlines the Registration Flow for WP1.

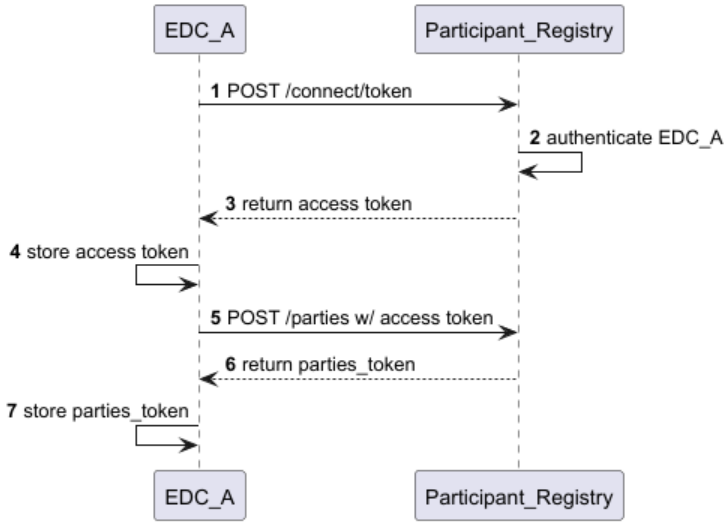

Figure 2: Registration Flow for WP1

2.2 Onboarding at the Participant Registry (WP2)

For WP2, this flow remains mostly the same, with two differences to WP1. Instead of the creation of a *parties_token*, a *Participant Credential* is issued to the participants wallet. For any later authorization flows, this credential can be presented to other parties to confirm its adherence to the iSHARE Trust Framework. Figure 3 shows this

authentication flow at the Participant Registry. During this registration process, the Participant Registry is also going to issue a did:iShare to the EDC participant. Corresponding to the did:ishare documentation, the Participant Registry is queried for the resolution of the DID and is also going to host the DID-document.

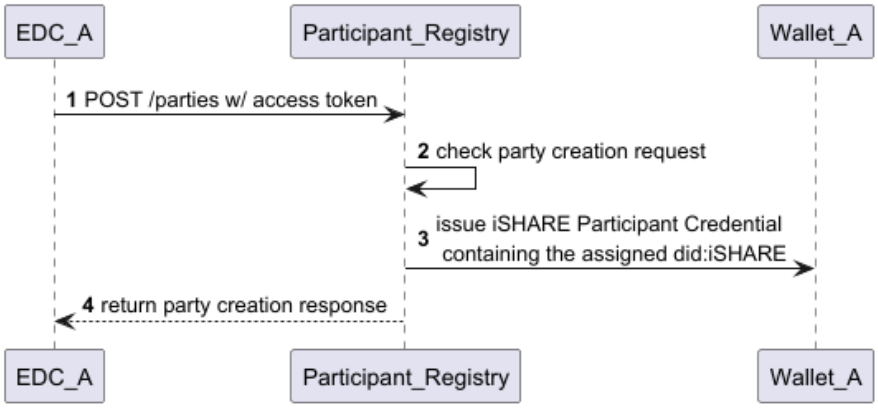

Figure 3: Registration Flow for WP2

3 iSHARE Authentication Flow

Authentication in the iSHARE Trust Framework is handled on a peer-to-peer basis. To connect to another party, the authentication endpoint of the counterparty needs to be queried to receive an access token, as described in the iSHARE Authentication Flow for M2M interactions[1].

3.1 Authentication with X.509 Certificates (WP1)

For WP1, this authentication is based on X.509 certificates with a central list of certificate authorities stored at the Participant Registry. To this end, the certificates issued at the registration process are used to sign authentication requests. Based on these certificates, EDC A needs to authenticate at EDC B before sending any DSP request. Figure 4 shows the process of mutual authentication between two EDCs.

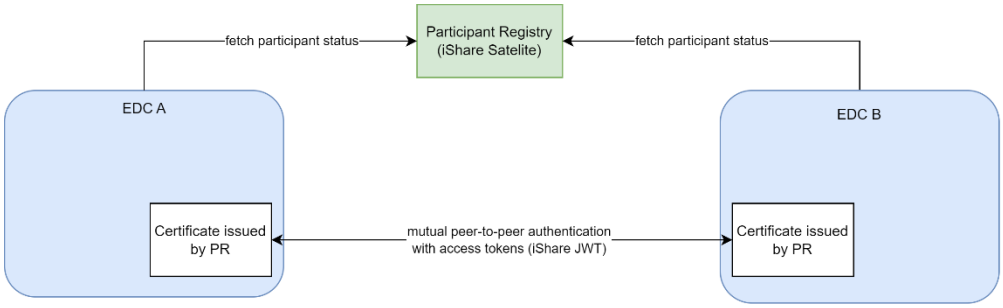

Figure 4: Authentication in iSHARE based on X.509 certificates

Figure 5 details these steps which are necessary for the authentication of an EDC instance "EDC A". The authentication of EDC A takes place in two different steps. At first, an access token is requested at EDC B. The request contains a client assertion signed with the private key associated with its X.509 certificate. After this, the validity of EDC A's certificate is checked by EDC B against the trusted list of CAs, which is fetched from the Participant Registry. Subsequently, its revocation status and expiry date are checked. As the last step before the access token is issued, the participant status of the counterparty is fetched and validated. As described in Section 2, a successful registration as a participant is a prerequisite for any authentication in the iSHARE Trust Framework.

[1] https://dev.iSHARE.eu/reference/authentication

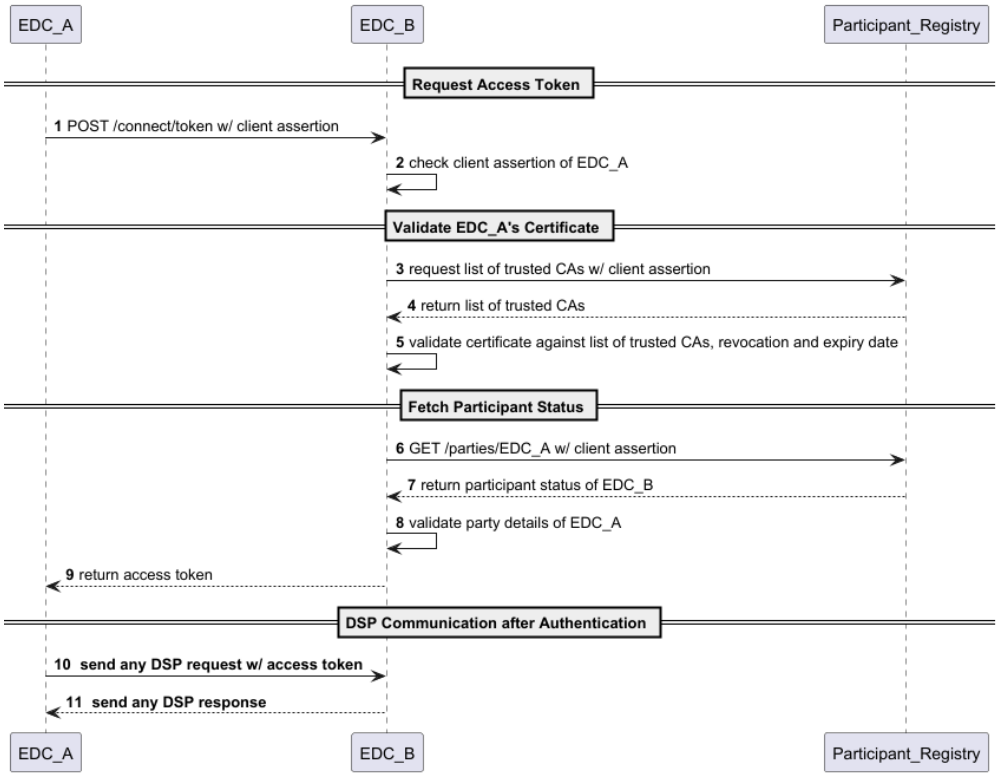

Figure 5: Authentication Flow for WP1

3.2 Authentication with Verifiable Credentials (WP2)

By using VCs, the authentication flow can be simplified for WP2. When DCP is used on the identity layer, each DSP request will be accompanied by the presentation of VCs using the DCP. Which credentials are presented depends on the context of the request, either through matching policy constraints to required VCs or by manually defining which credentials should be presented to the counterparty, if no policy is available in the request context. Additionally, the EDC allows for defining default credentials, that will be presented for every request.

As the *Participant Credentia*l is used for authentication, it needs to be presented for every DSP request and thus should be defined as a default credential in the EDC. Figure 6 shows the authentication flow for WP2.

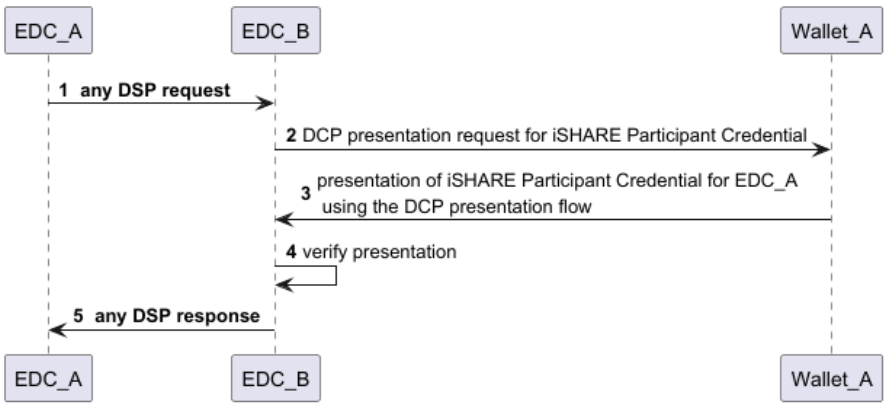

Figure 6: Authentication Flow for WP2

4 iSHARE Authorization Flow

In the iSHARE Trust Framework, authentication and authorization are handled as two distinct processes. While the integration of the authentication flows has been described in the previous section, this section describes the integration of the iSHARE authorization flows to the DSP interactions.

4.1 Authorization with Delegated Access Control

Authorizations in the iSHARE Trust Framework are managed at the Authorization Registry to support delegated access control patterns. Delegated access control allows one party to act on behalf of another party. This way, fine-grained access control patterns can be modeled. Within the AR, delegation policies are registered by the party which delegates the access (called *delegator*), and the Delegation Evidence is obtained for the party which wants to access the data (called *delegate*). There are many possibilities to integrate delegated access control into the different steps of the DSP communication. Different possibilities are elaborated in Section 4.4.

4.2 Prerequisites for the Integration of Delegations

The following prerequisites are considered out-of-scope for this adoption plan and are considered as given:

- **Management of delegations:** the management of delegation policies inside the Authorization Registry by a third party is considered outside the scope of this adoption plan. Thus, the EDC should only use the Delegation Evidence and does not register delegation policies itself.
- **Delegation Rights Credential:** for WP2**,** a *Delegation Rights Credential* is issued after the delegator registers a delegation at the AR. The credential is issued by the Authorization Registry to the delegate's wallet. When the delegate requests a service at a later point in time, this credential can be presented and verified by the service provider against the issuing AR.

4.3 Obtaining the Delegation Evidence

In order to use delegations for access control, the EDC needs to first query the Authorization Registry to fetch *a* Delegation Evidence Token and evaluate it afterwards. A Delegation Evidence Token defines the allowed data access for the delegate using *rules* and *actions*[1].

There are two ways of obtaining the Delegation Evidence Token from the AR, which differ in whether the sending or the receiving party fetches the Delegation Evidence Token from the AR. The options are described in more detail in the following sections.

Both options are technically feasible to implement in the EDC. However, for WP1 we recommend the Delegation Evidence is fetched by the receiving EDC as the consumer cannot present credentials natively in an OAuth2 flow. For WP2, the Delegation Evidence should be obtained by the consumer EDC at the Authorization Registry and

[1] https://framework.iSHARE.eu/detailed-descriptions/technical/structure-of-delegation-evidence

then sent over to the provider EDC. This pattern aligns with the presentation of VCs. The following diagrams show the components and sequences for WP1 and WP2.

4.3.1 Provider fetches the Delegation Evidence Token at the Authorization Registry (WP1)

Figure 7 shows where the Delegation Evidence Token is fetched by the provider. After a DSP request is sent by the consumer EDC, the Delegation Evidence is requested at the Authorization Registry by the EDC Provider. Based on the DE, the provider decides whether the DSP request is accepted or not in an EDC policy evaluation.

In this case, the EDC Provider needs find out which Authorization Registry to call, this could be accomplished by querying the participant information for the consumer EDC at the Participant Registry.

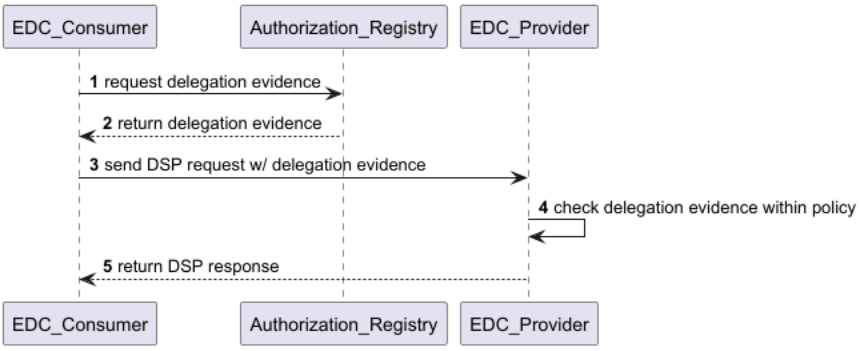

Figure 7: Provider fetches Authorization Registry to obtain Delegation Evidence Token after a DSP request is send, used in WP1.

4.3.2 Consumer presents Delegation Rights Credential (WP2)

The *Delegation Rights Credential* is presented during a service request in the same manner that the *Participant Credential* is presented for the authentication flow shown in Figure 8. Contrary to the *Participant Credential*, the *Delegation Rights Credential* only must be presented if a Delegation Evidence policy is attached to the requested dataset. Therefore, this VC should not be set as a default credential. Instead, the EDC can utilize scope extraction to map the Delegation Evidence policy constraint to the *Delegation Rights Credential*. Thus, the VC will automatically be presented for any DSP negotiation and transfer process message that is linked to a policy containing said constraint.

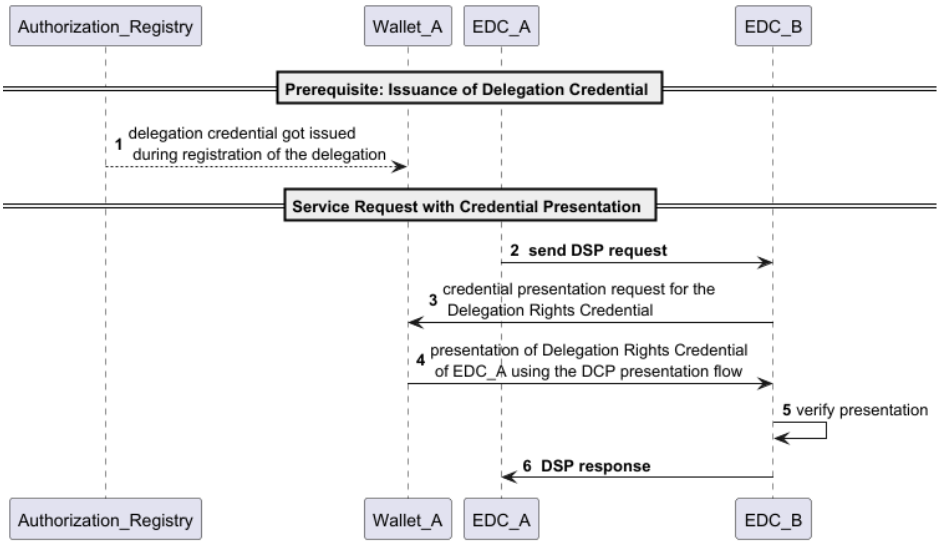

Figure 8: Consumer presents its Delegation Rights Credential, used in WP2.

4.4 Validating the Delegation Evidence for DSP Requests

The check of the obtained Delegation Evidence Token for the respective DSP step should be accomplished through an EDC policy evaluation. Hereby, the Delegation Evidence Token must be checked in different ways, based on the specific DSP requests.

- **Catalog Request:** During catalog requests, the EDC evaluates associated policies to determine whether an offer is visible to the requesting party. This evaluation is based only on information about the requesting participant, not including information about the datasets themselves. While it can be evaluated here whether the requesting party holds any DEs from a specific party, it can thus not be evaluated whether there is a DE for a specific dataset.

- **Contract Negotiation:** For each received contract request, the EDC evaluates the policies in the requested contract. Though again, this evaluation is based only on information about the requesting participant. Thus, the Delegation Evidence cannot be validated here through standard policy evaluation. But the EDC provides different mechanisms to pause contract negotiations and react to events within the negotiation. These can be utilized to ensure that a contract agreement can only be obtained with a valid Delegation Evidence for the given dataset.

- **Transfer Initiation:** When requesting a transfer, the EDC will evaluate policies contained in the contract agreement. All information about the contract agreement, including information about the dataset, is available. The EDC can therefore check whether a Delegation Evidence is available for the requesting party for the requested dataset.

- **On the Data Plane:** Directly before the actual data transfer takes place, the Data Plane could be required to additionally validate the Delegation Evidence Token again. Currently, the Data Plane is not policy-aware, i.e. it does not perform any policy evaluation after it receives the instructions for a transfer from the Control Plane. However, it receives all relevant information to perform such checks, like the contract agreement. An additional validation of the Delegation Evidence could therefore be implemented here, though this would require re-implementing existing Data Plane extensions.

5 Implementation Plan (WP1)

This section provides an implementation guide listing the details, extensions points and areas of change in the EDC, which are necessary to support all the concepts and requirements outlined in the previous sections.

For each class that needs to be implemented, a corresponding template is attached in the Appendix: EDC Java Extensions for WP1 and Appendix: EDC Java Extensions for WP2. The templates contain all relevant interfaces to EDC classes, including imports, super-classes and interfaces as well as usage of existing services, and thus provide guidance on how each class can be integrated into the EDC.

This implementation plan currently only covers the details of WP1. For WP2, this plan will be extended to cover the differences to WP1.

5.1 Prerequisites

For the implementations outlined in the following sections to work in a running EDC instance, the following secrets must be present in the vault used by the EDC instance:

- The **iSHARE certificate** (PEM format)
- The **corresponding private key** (PEM format)
- The **corresponding public key** (PEM format)
- The **certificate chain** (PEM format)

5.2 Registration at the Participant Registry

Should the EDC be required to perform registration at the Participant Registry, this can be encapsulated in a service. The service is responsible for sending a request for participant registration to the Participant Registry. All required information, like the URL of the Participant Registry as well relevant participant details for the "Create New Party" request[1], can be handed to the service as parameters. In the response to the participant registration request, the certificate is obtained. This needs to be stored in the vault used by the EDC, which can be done utilizing the *Vault* class.

Depending on when registration should be performed, e.g., upon start-up of the connector or via a manual trigger, the service can be called in different places. For a manual trigger, the service may be called by a dedicated controller class, allowing starting the registration flow via API request. All relevant parameters can then be defined in the corresponding API call. To have the EDC perform automatic registration upon start-up, the service can be called once within a *ServiceExtension* implementation. All relevant parameters can then be added to and obtained from the EDC's configuration.

More details about controller classes and *ServiceExtension* implementations can be found in the following section.

[1] https://dev.ishare.eu/participant-registry-role/parties-post

5.3 Authentication

For authentication, the following implementation details are outlined for WP1.

5.3.1 Model class for Access Tokens

There is no need for a custom model class, as iSHARE access tokens consist of a JWT and the corresponding expiration time. This combination is represented in the EDC as a *TokenRepresentation*.
IShareTokenDecorator.java

5.3.2 Token Decorator for JWTs

In the EDC, population of header and payload fields of JWTs is handled by implementations of the *TokenDecorator* interface. Implementing an iSHARE-specific *TokenDecorator* provides an easy way to ensure all relevant fields are added to the generated JWTs. Multiple *TokenDecorators* may be implemented for different types of JWTs, if they require different attributes to be present. The structure for a *TokenDecorator* implementation is depicted in IShareTokenDecorator.java.

5.3.3 Service for Handling Access Tokens

There needs to be a service that handles both the generation as well as the validation of iSHARE access tokens as shown in IShareAccessTokenService.java. For the generation of JWTs, the existing *JwtGenerationService* can be utilized. The *JwtGenerationService* requires the ID of the vault entry containing the private key for signing as well as the previously described *TokenDecorator*. The service should also be capable of validating previously generated JWTs to verify whether a counterparty is authenticated, e.g., by verifying the JWT's signature using the corresponding public key and/or caching previously generated tokens and comparing a received token to the cached list.

5.3.4 Controller for the Connect Token Endpoint

The iSHARE framework requires a /connect/token endpoint to be provided by every role, which other parties can utilize to request and obtain an access token for further authentication. In the EDC, typically Jakarta annotations are used to define a controller and its endpoints (incl. HTTP method, request path and parameters, etc.) as shown in IShareAccessTokenController.java. Thus, the endpoint can be defined as specified in the iSHARE documentation[1]. The controller defines the logic behind the endpoint, which in this case is the generation of an access token using the previously defined service for handling access tokens.

5.3.5 Service for Identity Handling

For the integration of iSHARE into the EDC, iSHARE authentication flows need to be followed for all DSP communication. In the EDC, the authentication for DSP communication is handled by implementations of *IdentityService*. A custom implementation of *IdentityService* is therefore required.

[1] https://dev.ishare.eu/all-roles-common-endpoints/access-token-m2m

The *IdentityService* interface provides two methods, one for obtaining a token to use in DSP requests, and one for validating a token received from another party. In these methods, the iSHARE authentication flow is implemented, as IShareIdentityService.java shows.

For obtaining a token, this includes the following steps:
- Generating a JWT for calling the */connect/token* endpoint
- Building the access token request
- Calling the Participant Registry using the counterparty's DID (available as "aud" claim in the token parameters)
- Extracting the counterparty's /capabilities endpoint from the response
- Obtaining the counterparty's /connect/token URL from the /capabilities endpoint response
- Sending the previously built request to the /connect/token endpoint
- Extracting both the token and its expiration time from the response
- Returning this information in a *TokenRepresentation*

For verifying a received token, the following steps are executed:
- Verifying the token by using the service for handling access tokens
- If additional information about the participant should be available for further processing (e.g., policy evaluation), calling the Participant Registry to obtain participant information
- Returning all participant information (incl. participant DID) as part of a *ClaimToken*

5.3.6 Extension Classes

In the EDC, implementations of *ServiceExtension* are required to interact with the runtime context. The extension classes provide access to configuration as well as services already provided by other extensions and enable providing services to the runtime context. At least one such extension class is required to obtain the services needed by the new implementations (like *JwtGenerationService*), as well as register new implementations such as controller and *IdentityService* in the context. This may also be split into multiple extension classes for clearer separation. The following tasks need to be performed by the *ServiceExtension* implementation:

- Registration of the *IdentityService* in the *ServiceExtensionContext*
- Registration of the controller at the *WebService* (as most existing EDC API contexts like the management or DSP APIs utilize JSON-LD expansion and compaction on ingress and egress respectively, a custom API context may be beneficial here, but the default API context can also be used)
- If multiple *ServiceExtension* implementations are created, some of the new implementations may also need to be registered in the *ServiceExtensionContext* to be available for dependent implementations

An example for a *ServiceExtension* for the authentication flow is shown in IShareAuthenticationExtension.java

5.4 Authorization

For authorization, the following implementation details are outlined for WP1.

5.4.1 Service for Validating DE

As described in the previous sections, authorization is handled by verifying whether a valid Delegation Evidence exists for the requesting participant and the requested dataset. As this logic needs to be executed in multiple different places, it should be

encapsulated in a service. The service is responsible for calling the Authorization Registry to obtain available DEs and evaluating the fields of a given Delegation Evidence for correctness. The frame for this validation service is depicted in DelegationEvidenceValidationService.java.

5.4.2 Policy Function

As described in Section 4.4, the Delegation Evidence evaluation should be done as part of policy enforcement. Therefore, a policy function needs to be implemented. As the requirement of a Delegation Evidence for access to a given dataset will likely be modelled as an *ODRL constraint*, the policy function should implement the *AtomicConstraintFunction* interface as shown in DelegationEvidencePolicyFunction.java. Within the function, the service for validating DEs can be utilized for evaluation.

As Delegation Evidence validation on the dataset level cannot be performed during catalog requests as outlined in Section 4.4, the implementation only shows the policy function for evaluation upon transfer initiation (*TransferProcessPolicyContext*). If some checks should still be performed during catalog requests, a second policy function can be created for the *CatalogPolicyContext* (org.eclipse.edc.connector.controlplane.catalog.spi.policy.CatalogPolicyContext) by replacing the *TransferProcessPolicyContext* with the *CatalogPolicyContext*. Please note, that only information about the requesting participant will be available for evaluation.

5.4.3 Delegation Evidence Evaluation During Contract Negotiation

As outlined in Section 4.4, a Delegation Evidence policy cannot be evaluated through standard EDC policy evaluation during contract negotiation, as there is no information about the target dataset available for policy evaluation. To still ensure that a contract agreement can only be obtained when a valid Delegation Evidence exists for the requested dataset, the EDC event system and guard pattern can be utilized.

First, a new *ContractNegotiationEvent* subtype is defined to indicate that a contract negotiation has been flagged as pending (i.e. it will not further be regarded by the state machine). This is shown in DelegationEvidenceNegotiationPendingGuard.java.

Then, a *ContractNegotiationPendingGuard* is implemented as shown in DelegationEvidenceNegotiationPendingGuard.java. This will be invoked whenever negotiations are fetched from the database. Within the guard, only negotiations of type "PROVIDER" and in state "REQUESTED" (i.e. every new incoming contract request) should be regarded. A negotiation is checked for presence of a Delegation Evidence constraint in the associated policy, and, if one is present, an instance of the new *ContractNegotiationEvent* sub-type is created and published via the *EventRouter*. In this case, *true* is returned from the method call to cause the negotiation to be flagged as pending. In all other cases, *false* is returned to ensure no negotiations without Delegation Evidence requirements are flagged as pending.

And last, an *EventSubscriber* is implemented as shown in DelegationEvidenceEventSubscriber.java. It will receive the *ContractNegotiationPendingEvents*, call the Delegation Evidence validation service and, depending on the result of the validation, either transition the negotiation to the state "AGREEING" or "TERMINATED". In both cases, the pending flag is removed.

5.4.4 Extension Classes

Same as for the authentication implementation, at least one implementation of *ServiceExtension* is also required here. The following tasks need to be performed by the *ServiceExtension* implementation:

- Registration of the policy function at the PolicyEngine (scope: transfer.process)
- Creation of rule bindings at the RuleBindingRegistry (one binding for the new Delegation Evidence constraint and one binding for every ODRL action that will be used)
- Registration of the ContractNegotiationPendingGuard implementation in the ServiceExtensionContext
- Registration of the EventSubscriber implementation at the EventRouter

An example for a *ServiceExtension* for the authorization flow is shown in IShareAuthorizationExtension.java

If a policy function for evaluation during catalog requests is created as described in Section 5.4.2, the rule bindings have to additionally be created for the *CATALOG_SCOPE* (org.eclipse.edc.connector.controlplane.catalog.spi.policy.CatalogPolicyContext.CATALOG_SCOPE), and the second policy function needs to also be registered for the *CatalogPolicyContext.class*.

5.4.5 Delegation Evidence Evaluation on the Data Plane

As mentioned in Section 4.4, the EDC Data Pane is currently not policy aware. Thus, no policy evaluation will take place on the Data Plane out-of-the-box. As information about the contract agreement (which includes both the policy and the dataset ID) is passed to the Data Plane by the Control Plane, policy evaluation can be implemented here. This requires a re-implementation of the Data Plane extensions for every transfer technology that should be used. Transfer technologies currently supported by the EDC are HTTP, AWS S3 and Azure Blob Storage. There is a dedicated extension module for each of these transfer technologies. Within these extension modules, the classes implementing *DataSourceFactory,* and potentially also the classes implementing *DataSource*, need to be re-implemented, to add the logic for policy evaluation here. The following steps can be implemented in the factory classes in the *validateRequest()* method:

- Extract the ID of the contract agreement from the DataFlowStartMessage
- Obtain the contract agreement from the database
- Extract policy and dataset information
- Perform policy evaluation utilizing the PolicyEngine
- Return a validation error in case the policy evaluation was not successful

This will cause the Data Plane to terminate the transfer, as validation of the request was unsuccessful. Alternatively, this logic could also be embedded in the *DataSource* implementations, to perform the evaluation right before the actual data transfer would start. In this case, the factory classes would need to pass all relevant information to the *DataSource* classes upon creation.

6 Implementation Plan (WP2)

This section provides an implementation guide listing the details, extensions points and areas of change in the EDC, which are necessary to support all the concepts and requirements outlined in the previous sections.

For each class that needs to be implemented, a corresponding template is attached in the Appendix: EDC Java Extensions for WP1 and Appendix: EDC Java Extensions for WP2. The templates contain all relevant interfaces to EDC classes, including imports, super-classes and interfaces as well as usage of existing services, and thus provide guidance on how each class can be integrated into the EDC.

This implementation plan covers the details for WP2. The plan is extended to cover the differences to the implementation plan for WP1.

6.1 Prerequisites

For the implementations outlined in the following sections the following prerequisites must be fulfilled:

- each participant operates a DCP-compliant wallet instance
- a did:ishare is assigned to each participant during registration at the Participant Registry
- the DID-document for each participant is hosted at the Participant Registry

6.2 Authentication

For authentication, the following implementation details are outlined for WP2.

6.2.1 Policy Validator Rule

As mentioned in Section 3.2, authentication can be accomplished by the presentation of the *Participant Credential.* As this credential needs to be presented for every request, it should be configured as a default credential. Within the EDC, this can be achieved by implementing a *PolicyValidatorRule<RequestPolicyContext>*. In the *apply* method, the pre-configured default scopes are added to the existing set of scopes, so that the scope for the *Participant Credential* is available for every DSP request. iShareDefaultScopeMappingFunction.java shows the implementation for the default scope mapping function. The format of a scope defining access to a type of VC is as follows:

org.eclipse.edc.vc.type:<credential-type>:<read | write | *>

6.2.2 Participant Agent Service Extension

Optionally, an implementation of *ParticipantAgentServiceExtension* can be created. A *ParticipantAgentServiceExtension* is called when the *ParticipantAgent* instance is constructed during the authentication process and can be utilized to add additional attributes to this instance. By default, an instance of *ParticipantAgent* provides access to all the participant's claims, including all VCs, but these need to be accessed through chained method calls, when they are used during any evaluation. In the *ParticipantAgentServiceExtension*, information (e.g. claims and attributes) can be extracted from the VCs and added to the *ParticipantAgent* as attributes, so that they can be accessed more easily by other dependent classes.

iShareParticipantAgentServiceExtension.java shows the implementation of the *ParticipantAgentServiceExtension*.

6.2.3 Policy Evaluation Function

To facilitate the evaluation of the Participant Credential for every incoming DSP request, a new policy evaluation function is additionally created. This function can obtain the list of VCs through the *ParticipantAgent* instance supplied in the policy context. Thus, the *Participant Credential* can be extracted and verified. iShareParticipantPolicyFunction.java shows the implementation for the policy evaluation function.

6.2.4 Extensions Classes

A *ServiceExtension* implementation is created to perform the extraction of the default scopes from configuration as well as registrations of the new implementations at the respective services. The following tasks are performed by the *ServiceExtension*:

- Extraction of the default scopes from configuration
- Registration of the scope mapping function at the *PolicyEngine* as a post validator
- Registration of the policy evaluation function at the *PolicyEngine*
- Optionally, registration of the *ParticipantAgentServiceExtension* instance, if one was created

iShareAuthenticationExtension.java shows the implementation of the *AuthenticationExtension*.

6.3 Authorization

For authorization, the following implementation details are outlined for WP2.

6.3.1 Scope Extractor

The authorization relies on the presentation of the *Delegation Rights Credential*, as described in Section 4.3. To this end, a *ScopeExtractor* to handle Delegation Evidence policies is created to automatically present the *Delegation Rights Credential* if a respective policy constraint is present. Within the *extractScopes* method, it should be checked whether a constraint is a valid Delegation Evidence constraint, and if so, return a scope that allows read access to the *Delegation Rights Credential*. If the constraint is not a valid Delegation Evidence constraint, an empty set should be returned to avoid any automatic mapping to a VC-type. The implementation of the *ScopeExtractor* is shown in DelegationEvidenceScopeExtractor.java.

6.3.2 Delegation Evidence Validation

The *DelegationEvidenceValidationService* is updated from the implementation for WP1. Instead of fetching the DEs from the AR, all DEs are available as credentials in the *vc* claim of the *ParticipantAgent* instance. Therefore, a second *validateDelegationEvidence* method is added to the service, that accepts an instance of *ParticipantAgent* instead of just its ID. The previously used method will also be kept in the service, as the *ParticipantAgent* is not available for the evaluation during contract negotiations, as this is not based on the EDC policy evaluation as outlined for WP1. Thus, to facilitate evaluation of DEs during contract negotiations, the provider will still need to actively fetch Delegation Evidence information from the AR. The implementation for the Delegation Evidence Validation is shown in DelegationEvidenceValidationService.java, ContractNegotiationPendingEvent.java, DelegationEvidenceEventSubscriber.java,

DelegationEvidenceNegotiationPendingGuard.java and DelegationEvidencePolicyFunction.java. For the classes required for Delegation Evidence Validation, there are only two differences compared to the implementation for WP1:

- Validation service: second method added as described above
- Policy function: calls the new method of the validation service

6.3.3 Participant Agent Service Extension

As is described for Authentication, another *ParticipantAgentServiceExtension* can be created for extracting information from the VCs beforehand and adding them as singular attributes to the *ParticipantAgent* for easier access in the validation service. The implementation of a *ParticipantAgentServiceExtension* is shown in iShareParticipantAgentServiceExtension.java.

6.3.4 Service Extension

The *ServiceExtension* for Authorization remains very similar to the one outlined in the implementation plan for WP1 with two additions:

- Registration of the *ScopeExtractor* instance at the *ScopeExtractorRegistry*
- Optionally, registration of the *ParticipantAgentServiceExtension* instance, if one was created

The *ServiceExtension* implementation is shown in iShareAuthorizationExtension.java.

6.4 Configuration of the Identity Hub

If the EDC Identity Hub is used as the DCP-compliant wallet to present credentials, the following configurations need to be performed:

- a key public/private key pairs for participant and superuser
- did:ishare used to configure the EDC participant ID
- trusted issuer configuration for Participant Registry and AR

If the Identity Hub instance is used for only one participant, a *ServiceExtension* implementation can be created to statically create and configure the *ParticipantContext* for this participant, so that this will be automatically performed upon start-up of the Identity Hub instance. If the Identity Hub instance is shared by multiple participants, it may be more feasible to create and remove them dynamically at runtime. Thus, no *ServiceExtension* should be created, and registrations should instead happen by using the Identity Hub's API.

Disregarding of whether the Identity Hub or another DCP compliant wallet is used, some configuration values need to be supplied to the Control Plane, so that it uses the wallet during DSP communication. The following properties need to be supplied:

- edc.participant.id=did:ishare:<id>
- edc.iam.issuer.id= did:ishare:<id>
- edc.iam.sts.privatekey.alias=<key-of-vault-entry-for-private-key>
- edc.iam.sts.publickey.id=did:ishare:<id>#<key-id>
- edc.iam.sts.oauth.client.secret.alias=<key-of-vault-entry-for-sts-client-secret>
- edc.iam.sts.oauth.client.id=<sts-client-id>
- edc.iam.sts.oauth.token.url=<url-of-the-remote-sts>
- edc.iam.trusted-issuer.<issuer-id>.id=<did-of-the-issuer>
- edc.iam.trusted-issuer.<issuer-id>.supportedtypes=["<credential-type>"]

7 Appendix: EDC Java Extensions for WP1

7.1 Authentication

7.1.1 IShareTokenDecorator.java

```java
import org.eclipse.edc.spi.iam.TokenParameters;
import org.eclipse.edc.token.spi.TokenDecorator;

public class IShareTokenDecorator implements TokenDecorator {

    // multiple TokenDecorators may be required for different iSHARE tokens, if they require
    // different header and payload fields

    @Override
    public TokenParameters.Builder decorate(TokenParameters.Builder builder) {
        // add iSHARE JWT headers
        builder.header("key", "value");
        // ...

        // add iSHARE JWT payload fields
        builder.claims("key", "value");
        // ...

        return builder;
    }
}
```

7.1.2 IShareAccessTokenService.java

```java
import org.eclipse.edc.spi.iam.TokenRepresentation;
import org.eclipse.edc.spi.result.Result;
import org.eclipse.edc.spi.security.Vault;
import org.eclipse.edc.token.JwtGenerationService;

public class IShareAccessTokenService {

    private JwtGenerationService jwtGenerationService;
    private IShareTokenDecorator tokenDecorator;
    private Vault vault;

    // ID of the vault entries containing the private & public key
    private String privateKeyId;
    private String publicKeyId;

    // constructor

    public Result<TokenRepresentation> generateAccessToken() {
        var jwtResult = jwtGenerationService.generate(privateKeyId, tokenDecorator);
        var jwt = jwtResult.getContent();

        // get/calculate expiration time

        var tokenRepresentation = TokenRepresentation.Builder.newInstance()
                .token(jwt)
                .expiresIn(expiresIn)
                .build();
        return Result.success(tokenRepresentation);
    }

    public Result<Void> validateAccessToken(String jwt) {
        var publicKey = vault.resolveSecret(publicKeyId);

        // validate signature on JWT

        return Result.success(); // return Result.failure("reason");
    }

    // optionally, issued access tokens can be cached for the participant
    // - e.g. in a Map<String, List<TokenRepresentation>>
    // - additional parameter for the participant ID on both methods
    // - during validation, the received JWT can be checked against the stored JWTs
}
```

7.1.3 IShareAccessTokenController.java

```java
import jakarta.json.Json;
import jakarta.json.JsonObject;
import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MultivaluedMap;

import static jakarta.ws.rs.core.MediaType.APPLICATION_JSON;
import static jakarta.ws.rs.core.MediaType.APPLICATION_FORM_URLENCODED;

public class IShareAccessTokenController {

    private IShareAccessTokenService accessTokenService;

    // constructor

    @Produces(APPLICATION_JSON)
    @Consumes(APPLICATION_FORM_URLENCODED)
    @POST
    @Path("/connect/token")
    public JsonObject getAccessToken(MultivaluedMap<String, String> request) {
        // extract client_id from request
        // verify client assertion & participant status

        var result = accessTokenService.generateAccessToken();
        if (result.failed()) {
            // error response
        }

        var accessToken = result.getContent();

        // build JSON response, e.g. using Jakarta JSON library
        return Json.createObjectBuilder()
                .add("access_token", Json.createArrayBuilder()
                        .add(accessToken.getToken())
                        .build())
                .add("token_type", Json.createArrayBuilder()
                        .add("Bearer")
                        .build())
                .add("expires_in", Json.createArrayBuilder()
                        .add(accessToken.getExpiresIn())
                        .build())
                .build();
    }

}
```

7.1.4 IShareIdentityService.java

```
import okhttp3.Request;
import org.eclipse.edc.http.spi.EdcHttpClient;
import org.eclipse.edc.spi.iam.ClaimToken;
import org.eclipse.edc.spi.iam.IdentityService;
import org.eclipse.edc.spi.iam.TokenParameters;
import org.eclipse.edc.spi.iam.TokenRepresentation;
import org.eclipse.edc.spi.iam.VerificationContext;
import org.eclipse.edc.spi.result.Result;
import org.eclipse.edc.token.JwtGenerationService;

public class IShareIdentityService implements IdentityService {

    private JwtGenerationService jwtGenerationService;
    private IShareTokenDecorator tokenDecorator;
    private IShareAccessTokenService accessTokenService;
    private EdcHttpClient httpClient;

    // ID of the vault entry containing the private key
    private String privateKeyId;

    // Participant ID of this EDC
    private String participantId;

    // URL of the PR to use for obtaining participant information
    private String participantRegistryUrl;

    // constructor

    @Override
    public Result<TokenRepresentation> obtainClientCredentials(TokenParameters
tokenParameters) {
        var clientAssertion = jwtGenerationService.generate(privateKeyId,
tokenDecorator);
        var accessTokenRequestBuilder = new Request.Builder();
        // build HTTP request for access token request using own participantId
        var accessTokenRequest = accessTokenRequestBuilder.build();

        var counterPartyDid = tokenParameters.getStringClaim("aud");
        // send request to PR requesting information about this counterPartyDid
        // extract URL of /capabilities endpoint from the response
        // make request to /capabilities endpoint
        // extract /connect/token endpoint from the response

        // send accessTokenRequest to /connect/token endpoint
        // extract access token and expiration time from response
```

```
        var tokenRepresentation = TokenRepresentation.Builder.newInstance()
                .token(jwt)
                .expiresIn(expiresIn)
                .build();
        return Result.success(tokenRepresentation);
    }

    @Override
    public Result<ClaimToken> verifyJwtToken(TokenRepresentation
tokenRepresentation, VerificationContext verificationContext) {
        var result =
accessTokenService.validateAccessToken(tokenRepresentation.getToken());
        if (result.failed()) {
            return Result.failure("unauthorized");
        }

        // decode JWT
        // get CounterPartyDid from the decoded token

        // if additional information about the counter-party is required for further
processing:
        // call the PR to obtain additional information using the DID from the token

        // add all relevant information about the participant to the ClaimToken
        var claimToken = ClaimToken.Builder.newInstance()
                .claim("client_id", counterPartyDid)
                .claim("key", "value")
                .build();
        return Result.success(claimToken);
    }
}
```

7.1.5 IShareAuthenticationExtension.java

```java
import org.eclipse.edc.http.spi.EdcHttpClient;
import org.eclipse.edc.runtime.metamodel.annotation.Inject;
import org.eclipse.edc.runtime.metamodel.annotation.Provider;
import org.eclipse.edc.runtime.metamodel.annotation.Setting;
import org.eclipse.edc.spi.iam.IdentityService;
import org.eclipse.edc.spi.security.Vault;
import org.eclipse.edc.spi.system.ServiceExtension;
import org.eclipse.edc.spi.system.ServiceExtensionContext;
import org.eclipse.edc.token.JwtGenerationService;
import org.eclipse.edc.web.spi.WebService;

public class IShareAuthenticationExtension implements ServiceExtension {

    // any value that should be taken from configuration (e.g. participant registry
URL) can be defined as shown below
    // the property can then be set in configuration like this:
edc.ishare.x.y.z=value
    // the value will be injected into the String field following the annotation
    // setting keys can be chosen freely
    @Setting(description = "description of the configuration value", key =
"edc.ishare.x.y.z", defaultValue = "default value, if applicable")
    String configValue;

    // any required EDC service can be injected as shown below
    @Inject
    private JwtGenerationService jwtGenerationService;
    @Inject
    private Vault vault;
    @Inject
    private EdcHttpClient edcHttpClient;
    @Inject
    private WebService webService;

    @Provider
    public IdentityService identityService() {
        return new IShareIdentityService(...);
    }

    @Override
    public void initialize(ServiceExtensionContext context) {
        var accessTokenService = new IShareAccessTokenService(...);
        var participantId = context.getParticipantId();
        webService.registerResource(new
IShareAccessTokenController(accessTokenService, participantId));
    }
}
```

7.2 Authorization

7.2.1 DelegationEvidenceValidationService.java

```
public class DelegationEvidenceValidationService {

    // additional parameters may be required for evaluation depending on the
implementation
    public boolean validateDelegationEvidence(String participantId, String
datasetId) {
        // call AR to fetch DEs for participant
        // check whether DE for given dataset exists
        // validate DE (e.g. not expired, not revoked, etc.)
        // return true or false depending on validation result
    }

}
```

7.2.2 DelegationEvidencePolicyFunction.java

```
import
org.eclipse.edc.connector.controlplane.contract.spi.policy.TransferProcessPolicyCont
ext;
import org.eclipse.edc.policy.engine.spi.AtomicConstraintRuleFunction;
import org.eclipse.edc.policy.model.Operator;
import org.eclipse.edc.policy.model.Permission;
import org.eclipse.edc.spi.result.Result;

// if DE constraint is used in other ODRL rule types than permission, a different
rule type can be chosen here
public class DelegationEvidencePolicyFunction implements
AtomicConstraintRuleFunction<Permission, TransferProcessPolicyContext> {

    private DelegationEvidenceValidationService validationService;

    @Override
    public boolean evaluate(Operator operator, Object o, Permission permission,
TransferProcessPolicyContext policyContext) {
        var participant = policyContext.participantAgent();
        var agreement = policyContext.contractAgreement();
        var datasetId = agreement.getAssetId();

        return
validationService.validateDelegationEvidence(participant.getIdentity(), datasetId);
    }

    @Override
    public Result<Void> validate(Operator operator, Object rightValue, Permission
permission) {
        // perform syntax validation, i.e. whether a given constraint matches the
expected pattern (valid operator & right operand)
    }

    @Override
    public String name() {
        return "DelegationEvidencePolicyFunction";
    }
}
```

7.2.3 ContractNegotiationPendingEvent.java

```
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.databind.annotation.JsonPOJOBuilder;
import
org.eclipse.edc.connector.controlplane.contract.spi.event.contractnegotiation.Contra
ctNegotiationEvent;

public class ContractNegotiationPendingEvent extends ContractNegotiationEvent {
    @Override
    public String name() {
        return "ContractNegotiationPendingEvent";
    }

    @JsonPOJOBuilder(
            withPrefix = ""
    )
    public static class Builder extends
ContractNegotiationEvent.Builder<ContractNegotiationPendingEvent,
ContractNegotiationPendingEvent.Builder> {
        private Builder() {
            super(new ContractNegotiationPendingEvent());
        }

        @JsonCreator
        public static ContractNegotiationPendingEvent.Builder newInstance() {
            return new ContractNegotiationPendingEvent.Builder();
        }

        public ContractNegotiationPendingEvent.Builder self() {
            return this;
        }
    }
}
```

7.2.4 DelegationEvidenceNegotiationPendingGuard.java

```
import
org.eclipse.edc.connector.controlplane.contract.spi.negotiation.ContractNegotiationP
endingGuard;
import
org.eclipse.edc.connector.controlplane.contract.spi.types.negotiation.ContractNegoti
ation;
import org.eclipse.edc.spi.event.EventEnvelope;
import org.eclipse.edc.spi.event.EventRouter;

import java.time.Clock;

public class DelegationEvidenceNegotiationPendingGuard implements
ContractNegotiationPendingGuard {

    private EventRouter eventRouter;
    private Clock clock;

    @Override
    public boolean test(ContractNegotiation contractNegotiation) {
        if (!contractNegotiation.getType().equals(ContractNegotiation.Type.PROVIDER)
||
                !contractNegotiation.stateAsString().equals("REQUESTED")) {
            return false;
        }

        var policy = contractNegotiation.getLastContractOffer().getPolicy();

        // check whether policy contains a DE constraint
        // if no DE policy constraint exists, return false

        var event = ContractNegotiationPendingEvent.Builder.newInstance()
                // add all relevant event fields
                .build();
        var envelope = EventEnvelope.Builder.newInstance()
                .payload(event)
                .at(clock.millis())
                .id() // if no ID is set, a random UUID will be generated
                .build();
        eventRouter.publish(envelope);
        return true;
    }
}
```

7.2.5 DelegationEvidenceEventSubscriber.java

```
import
org.eclipse.edc.connector.controlplane.contract.spi.negotiation.store.ContractNegoti
ationStore;
import org.eclipse.edc.spi.event.Event;
import org.eclipse.edc.spi.event.EventEnvelope;
import org.eclipse.edc.spi.event.EventSubscriber;

public class DelegationEvidenceEventSubscriber implements EventSubscriber {

    private DelegationEvidenceValidationService validationService;
    private ContractNegotiationStore negotiationStore;

    @Override
    public <E extends Event> void on(EventEnvelope<E> eventEnvelope) {
        if (eventEnvelope.getPayload() instanceof ContractNegotiationPendingEvent) {
            var event = (ContractNegotiationPendingEvent)
eventEnvelope.getPayload();
            var negotiationId = event.getContractNegotiationId();
            var negotiation = negotiationStore.findById(negotiationId);

            var participantId = negotiation.getCounterPartyId();
            var datasetId = negotiation.getLastContractOffer().getAssetId();

            if (validationService.validateDelegationEvidence(participantId,
datasetId)) {
                negotiation.transitionAgreeing();
            } else {
                negotiation.transitionTerminating();
            }

            negotiation.setPending(false);
            negotiationStore.save(negotiation);
        }
    }
}
```

7.2.6 IShareAuthorizationExtension.java

```
import
org.eclipse.edc.connector.controlplane.contract.spi.negotiation.ContractNegotiationP
endingGuard;
import
org.eclipse.edc.connector.controlplane.contract.spi.negotiation.store.ContractNegoti
ationStore;
import
org.eclipse.edc.connector.controlplane.contract.spi.policy.TransferProcessPolicyCont
ext;
import org.eclipse.edc.policy.engine.spi.PolicyEngine;
import org.eclipse.edc.policy.engine.spi.RuleBindingRegistry;
import org.eclipse.edc.policy.model.Permission;
import org.eclipse.edc.runtime.metamodel.annotation.Inject;
import org.eclipse.edc.runtime.metamodel.annotation.Provider;
import org.eclipse.edc.runtime.metamodel.annotation.Setting;
import org.eclipse.edc.spi.event.EventRouter;
import org.eclipse.edc.spi.system.ServiceExtension;
import org.eclipse.edc.spi.system.ServiceExtensionContext;

import java.time.Clock;

import static
org.eclipse.edc.connector.controlplane.contract.spi.policy.TransferProcessPolicyCont
ext.TRANSFER_SCOPE;
import static
org.eclipse.edc.jsonld.spi.PropertyAndTypeNames.ODRL_USE_ACTION_ATTRIBUTE;

public class IShareAuthorizationExtension implements ServiceExtension {

    // any value that should be taken from configuration (e.g. participant registry
URL) can be defined as shown below
    // the property can then be set in configuration like this:
edc.ishare.x.y.z=value
    // the value will be injected into the String field following the annotation
    // setting keys can be chosen freely
    @Setting(description = "description of the configuration value", key =
"edc.ishare.x.y.z", defaultValue = "default value, if applicable")
    String configValue;

    @Inject
    private PolicyEngine policyEngine;
    @Inject
    private RuleBindingRegistry ruleBindingRegistry;
    @Inject
    private EventRouter eventRouter;
    @Inject
```

```
    private Clock clock;
    @Inject
    private ContractNegotiationStore negotiationStore;

    @Provider
    public ContractNegotiationPendingGuard contractNegotiationPendingGuard() {
        return new DelegationEvidenceNegotiationPendingGuard(eventRouter, clock);
    }

    @Override
    public void initialize(ServiceExtensionContext context) {
        var deValidationService = new DelegationEvidenceValidationService();

        // create a binding of every ODRL action that the DE constraint will be used
with
        ruleBindingRegistry.bind(ODRL_USE_ACTION_ATTRIBUTE, TRANSFER_SCOPE);

        // first parameter needs to be the left operand of the ODRL DE constraint
        ruleBindingRegistry.bind("DelegationEvidence", TRANSFER_SCOPE);

        // String parameter needs to be the left operand of the ODRL DE constraint,
same as used for rule binding
        policyEngine.registerFunction(TransferProcessPolicyContext.class,
Permission.class,
                "DelegationEvidence", new
DelegationEvidencePolicyFunction(deValidationService));

        eventRouter.register(ContractNegotiationPendingEvent.class,
                new DelegationEvidenceEventSubscriber(deValidationService,
negotiationStore));
    }

}
```

8 Appendix: EDC Java Extensions for WP2

8.1 Authentication

8.1.1 iShareDefaultScopeMappingFunction.java

```
import org.eclipse.edc.policy.context.request.spi.RequestPolicyContext;
import org.eclipse.edc.policy.engine.spi.PolicyValidatorRule;
import org.eclipse.edc.policy.model.Policy;

import java.util.HashSet;
import java.util.Set;

public class IShareDefaultScopeMappingFunction implements
PolicyValidatorRule<RequestPolicyContext> {

    private final Set<String> defaultScopes;

    // the default scopes should at least contain
"org.eclipse.edc.vc.type:ParticipantCredential:read", more may be added
    public IShareDefaultScopeMappingFunction(Set<String> defaultScopes) {
        this.defaultScopes = defaultScopes;
    }

    @Override
    public Boolean apply(Policy policy, RequestPolicyContext requestPolicyContext) {
        var builder = requestPolicyContext.requestScopeBuilder();
        var requestScope = builder.build();
        var scopes = requestScope.getScopes();

        var newScopes = new HashSet<>(defaultScopes);
        newScopes.addAll(scopes);

        builder.scopes(newScopes);
        return true;
    }
}
```

8.1.2 iShareParticipantAgentServiceExtension.java

```java
import org.eclipse.edc.iam.verifiablecredentials.spi.model.VerifiableCredential;
import org.eclipse.edc.participant.spi.ParticipantAgentServiceExtension;
import org.eclipse.edc.spi.iam.ClaimToken;
import org.jetbrains.annotations.NotNull;

import java.util.HashMap;
import java.util.Map;

import static java.util.Collections.emptyList;
import static java.util.Optional.ofNullable;

public class IShareParticipantAgentServiceExtension implements
ParticipantAgentServiceExtension {
    @Override
    public @NotNull Map<String, String> attributesFor(ClaimToken claimToken) {
        var credentials =
ofNullable(claimToken.getListClaim("vc")).orElse(emptyList())
                .stream()
                .filter(o -> o instanceof VerifiableCredential)
                .map(o -> (VerifiableCredential) o)
                .toList();

        var attributes = new HashMap<String, String>();

        // extract all required information from the VCs and add them to the
attributes map

        return attributes;
    }
}
```

8.1.3 iShareParticipantPolicyFunction.java

```java
import org.eclipse.edc.iam.verifiablecredentials.spi.model.VerifiableCredential;
import org.eclipse.edc.participant.spi.ParticipantAgentPolicyContext;
import org.eclipse.edc.policy.engine.spi.AtomicConstraintRuleFunction;
import org.eclipse.edc.policy.model.Operator;
import org.eclipse.edc.policy.model.Permission;

import java.util.Collections;
import java.util.List;
import java.util.Optional;

public class IShareParticipantPolicyFunction<C extends
ParticipantAgentPolicyContext> implements AtomicConstraintRuleFunction<Permission,
C> {
    @Override
    public boolean evaluate(Operator operator, Object rightValue, Permission
permission, C policyContext) {
        // if an instance of ParticipantAgentServiceExtension is created, the
extracted information
        // can instead be accessed through
policyContext.participantAgent().getAttributes()

        var participantCredential = (VerifiableCredential)
Optional.of(policyContext.participantAgent().getClaims())
                .map(it -> it.get("vc"))
                .map(List.class::cast)
                .orElse(Collections.emptyList())
                .stream()
                .filter(o -> o instanceof VerifiableCredential)
                .filter(vc -> ((VerifiableCredential)
vc).getType().contains("ParticipantCredential"))
                .findAny()
                .orElse(null);

        if (participantCredential == null) {
            return false;
        }

        // perform additional validations on participant credential if required

        return true;
    }
}
```

8.1.4 iShareAuthenticationExtension.java

```
import
org.eclipse.edc.connector.controlplane.catalog.spi.policy.CatalogPolicyContext;
import
org.eclipse.edc.connector.controlplane.contract.spi.policy.ContractNegotiationPolicy
Context;
import
org.eclipse.edc.connector.controlplane.contract.spi.policy.TransferProcessPolicyCont
ext;
import org.eclipse.edc.participant.spi.ParticipantAgentService;
import org.eclipse.edc.policy.context.request.spi.RequestCatalogPolicyContext;
import
org.eclipse.edc.policy.context.request.spi.RequestContractNegotiationPolicyContext;
import
org.eclipse.edc.policy.context.request.spi.RequestTransferProcessPolicyContext;
import org.eclipse.edc.policy.engine.spi.PolicyEngine;
import org.eclipse.edc.policy.engine.spi.RuleBindingRegistry;
import org.eclipse.edc.policy.model.Permission;
import org.eclipse.edc.runtime.metamodel.annotation.Inject;
import org.eclipse.edc.runtime.metamodel.annotation.Setting;
import org.eclipse.edc.spi.system.ServiceExtension;
import org.eclipse.edc.spi.system.ServiceExtensionContext;

import java.util.Set;

import static
org.eclipse.edc.jsonld.spi.PropertyAndTypeNames.ODRL_USE_ACTION_ATTRIBUTE;
import static org.eclipse.edc.policy.engine.spi.PolicyEngine.ALL_SCOPES;

public class IShareAuthenticationExtension implements ServiceExtension {

    // any value that should be taken from configuration (e.g. participant registry
URL) can be defined as shown below
    // the property can then be set in configuration like this:
edc.ishare.x.y.z=value
    // the value will be injected into the String field following the annotation
    // setting keys can be chosen freely
    @Setting(description = "description of the configuration value", key =
"edc.ishare.x.y.z", defaultValue = "default value, if applicable")
    String configValue;

    @Setting(key = "...", description = "Set of default DCP scopes, comma-
separated")
    private String scopes;

    // any required EDC service can be injected as shown below
    @Inject
```

```
    private PolicyEngine policyEngine;
    @Inject
    private ParticipantAgentService participantAgentService;
    @Inject
    private RuleBindingRegistry ruleBindingRegistry;

    @Override
    public void initialize(ServiceExtensionContext context) {
        // parse scopes from comma-separated config value
        var defaultScopes = Set.of("...");

        // create a binding for every ODRL action that the DE constraint will be
used with
        ruleBindingRegistry.bind(ODRL_USE_ACTION_ATTRIBUTE, ALL_SCOPES);

        // first parameter needs to be the left operand of the ODRL DE constraint
        ruleBindingRegistry.bind("<participant-constraint-left-operand>",
ALL_SCOPES);

        var function = new IShareDefaultScopeMappingFunction(defaultScopes);
        policyEngine.registerPostValidator(RequestCatalogPolicyContext.class,
function::apply);
        policyEngine.registerPostValidator(RequestContractNegotiationPolicyContext.c
lass, function::apply);
        policyEngine.registerPostValidator(RequestTransferProcessPolicyContext.class
, function::apply);

        var participantCredentialFunction = new IShareParticipantPolicyFunction<>();
        policyEngine.registerFunction(CatalogPolicyContext.class, Permission.class,
"<participant-constraint-left-operand>", participantCredentialFunction);
        policyEngine.registerFunction(ContractNegotiationPolicyContext.class,
Permission.class, "<participant-constraint-left-operand>",
participantCredentialFunction);
        policyEngine.registerFunction(TransferProcessPolicyContext.class,
Permission.class, "<participant-constraint-left-operand>",
participantCredentialFunction);

        // if an implementation of ParticipantAgentServiceExtension is created, this
can be registered as follows
        participantAgentService.register(new
IShareParticipantAgentServiceExtension());
    }
}
```

8.2 Authorization

8.2.1 DelegationEvidenceScopeExtractor.java

```
import org.eclipse.edc.iam.identitytrust.spi.scope.ScopeExtractor;
import org.eclipse.edc.policy.context.request.spi.RequestPolicyContext;
import org.eclipse.edc.policy.model.Operator;

import java.util.Set;

public class DelegationEvidenceScopeExtractor implements ScopeExtractor {
    @Override
    public Set<String> extractScopes(Object leftValue, Operator operator, Object
rightValue, RequestPolicyContext requestPolicyContext) {
        // check whether policy constraint is a DE constraint
        if (leftValue instanceof String leftValueString &&
leftValueString.equals(...)) {

            // perform additional checks/validation if necessary

            return
Set.of("org.eclipse.edc.vc.type:DelegationRightsCredential:read");
        }

        return Set.of();
    }
}
```

8.2.2 DelegationEvidenceValidationService.java

```java
import org.eclipse.edc.iam.verifiablecredentials.spi.model.VerifiableCredential;
import org.eclipse.edc.participant.spi.ParticipantAgent;

import java.util.Collections;
import java.util.List;
import java.util.Optional;

public class DelegationEvidenceValidationService {

    // additional parameters may be required for evaluation depending on the
implementation

    public boolean validateDelegationEvidence(String participantId, String
datasetId) {
        // call AR to fetch DEs for participant
        // check whether DE for given dataset exists
        // validate DE (e.g. not expired, not revoked, etc.)
        // return true or false depending on validation result
    }

    public boolean validateDelegationEvidence(ParticipantAgent participantAgent,
String datasetId) {
        var credentials = Optional.of(participantAgent.getClaims())
                .map(it -> it.get("vc"))
                .map(List.class::cast)
                .orElse(Collections.emptyList())
                .stream()
                .filter(o -> o instanceof VerifiableCredential)
                .map(o -> (VerifiableCredential) o)
                .toList();

        // check whether DE credential for given dataset exists
        // validate DE (e.g. not expired, not revoked, etc.)
        // return true or false depending on validation result
    }

}
```

8.2.3 ContractNegotiationPendingEvent.java

```
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.databind.annotation.JsonPOJOBuilder;
import
org.eclipse.edc.connector.controlplane.contract.spi.event.contractnegotiation.Contra
ctNegotiationEvent;

public class ContractNegotiationPendingEvent extends ContractNegotiationEvent {
    @Override
    public String name() {
        return "ContractNegotiationPendingEvent";
    }

    @JsonPOJOBuilder(
            withPrefix = ""
    )
    public static class Builder extends
ContractNegotiationEvent.Builder<ContractNegotiationPendingEvent, Builder> {
        private Builder() {
            super(new ContractNegotiationPendingEvent());
        }

        @JsonCreator
        public static Builder newInstance() {
            return new Builder();
        }

        public Builder self() {
            return this;
        }
    }
}
```

8.2.4 DelegationEvidenceEventSubscriber.java

```
import
org.eclipse.edc.connector.controlplane.contract.spi.negotiation.store.ContractNegoti
ationStore;
import org.eclipse.edc.spi.event.Event;
import org.eclipse.edc.spi.event.EventEnvelope;
import org.eclipse.edc.spi.event.EventSubscriber;

public class DelegationEvidenceEventSubscriber implements EventSubscriber {

    private DelegationEvidenceValidationService validationService;
    private ContractNegotiationStore negotiationStore;

    @Override
    public <E extends Event> void on(EventEnvelope<E> eventEnvelope) {
        if (eventEnvelope.getPayload() instanceof ContractNegotiationPendingEvent) {
            var event = (ContractNegotiationPendingEvent)
eventEnvelope.getPayload();
            var negotiationId = event.getContractNegotiationId();
            var negotiation = negotiationStore.findById(negotiationId);

            var participantId = negotiation.getCounterPartyId();
            var datasetId = negotiation.getLastContractOffer().getAssetId();

            if (validationService.validateDelegationEvidence(participantId,
datasetId)) {
                negotiation.transitionAgreeing();
            } else {
                negotiation.transitionTerminating();
            }

            negotiation.setPending(false);
            negotiationStore.save(negotiation);
        }
    }
}
```

8.2.5 DelegationEvidenceNegotiationPendingGuard.java

```
import
org.eclipse.edc.connector.controlplane.contract.spi.negotiation.ContractNegotiationP
endingGuard;
import
org.eclipse.edc.connector.controlplane.contract.spi.types.negotiation.ContractNegoti
ation;
import org.eclipse.edc.spi.event.EventEnvelope;
import org.eclipse.edc.spi.event.EventRouter;

import java.time.Clock;

public class DelegationEvidenceNegotiationPendingGuard implements
ContractNegotiationPendingGuard {

    private EventRouter eventRouter;
    private Clock clock;

    @Override
    public boolean test(ContractNegotiation contractNegotiation) {
        if (!contractNegotiation.getType().equals(ContractNegotiation.Type.PROVIDER)
||
                !contractNegotiation.stateAsString().equals("REQUESTED")) {
            return false;
        }

        var policy = contractNegotiation.getLastContractOffer().getPolicy();

        // check whether policy contains a DE constraint
        // if no DE policy constraint exists, return false

        var event = ContractNegotiationPendingEvent.Builder.newInstance()
                // add all relevant event fields
                .build();
        var envelope = EventEnvelope.Builder.newInstance()
                .payload(event)
                .at(clock.millis())
                .id() // if no ID is set, a random UUID will be generated
                .build();
        eventRouter.publish(envelope);
        return true;
    }
}
```

8.2.6 DelegationEvidencePolicyFunction.java

```
import
org.eclipse.edc.connector.controlplane.contract.spi.policy.TransferProcessPolicyCont
ext;
import org.eclipse.edc.policy.engine.spi.AtomicConstraintRuleFunction;
import org.eclipse.edc.policy.model.Operator;
import org.eclipse.edc.policy.model.Permission;
import org.eclipse.edc.spi.result.Result;

// if DE constraint is used in other ODRL rule types than permission, a different
rule type can be chosen here
public class DelegationEvidencePolicyFunction implements
AtomicConstraintRuleFunction<Permission, TransferProcessPolicyContext> {

    private DelegationEvidenceValidationService validationService;

    @Override
    public boolean evaluate(Operator operator, Object o, Permission permission,
TransferProcessPolicyContext policyContext) {
        var participant = policyContext.participantAgent();
        var agreement = policyContext.contractAgreement();
        var datasetId = agreement.getAssetId();

        return validationService.validateDelegationEvidence(participant, datasetId);
    }

    @Override
    public Result<Void> validate(Operator operator, Object rightValue, Permission
permission) {
        // perform syntax validation, i.e. whether a given constraint matches the
expected pattern (valid operator & right operand)
    }

    @Override
    public String name() {
        return "DelegationEvidencePolicyFunction";
    }
}
```

8.2.7 iShareAuthorizationExtension.java

```
import
org.eclipse.edc.connector.controlplane.contract.spi.negotiation.ContractNegotiationP
endingGuard;
import
org.eclipse.edc.connector.controlplane.contract.spi.negotiation.store.ContractNegoti
ationStore;
import
org.eclipse.edc.connector.controlplane.contract.spi.policy.TransferProcessPolicyCont
ext;
import org.eclipse.edc.iam.identitytrust.spi.scope.ScopeExtractorRegistry;
import org.eclipse.edc.participant.spi.ParticipantAgentService;
import org.eclipse.edc.policy.engine.spi.PolicyEngine;
import org.eclipse.edc.policy.engine.spi.RuleBindingRegistry;
import org.eclipse.edc.policy.model.Permission;
import org.eclipse.edc.runtime.metamodel.annotation.Inject;
import org.eclipse.edc.runtime.metamodel.annotation.Provider;
import org.eclipse.edc.runtime.metamodel.annotation.Setting;
import org.eclipse.edc.spi.event.EventRouter;
import org.eclipse.edc.spi.system.ServiceExtension;
import org.eclipse.edc.spi.system.ServiceExtensionContext;

import java.time.Clock;

import static
org.eclipse.edc.connector.controlplane.contract.spi.policy.TransferProcessPolicyCont
ext.TRANSFER_SCOPE;
import static
org.eclipse.edc.jsonld.spi.PropertyAndTypeNames.ODRL_USE_ACTION_ATTRIBUTE;

public class IShareAuthorizationExtension implements ServiceExtension {

    // any value that should be taken from configuration (e.g. participant registry
URL) can be defined as shown below
    // the property can then be set in configuration like this:
edc.ishare.x.y.z=value
    // the value will be injected into the String field following the annotation
    // setting keys can be chosen freely
    @Setting(description = "description of the configuration value", key =
"edc.ishare.x.y.z", defaultValue = "default value, if applicable")
    String configValue;

    @Inject
    private ScopeExtractorRegistry scopeExtractorRegistry;
    @Inject
    private PolicyEngine policyEngine;
    @Inject
```

```
    private RuleBindingRegistry ruleBindingRegistry;
    @Inject
    private EventRouter eventRouter;
    @Inject
    private Clock clock;
    @Inject
    private ContractNegotiationStore negotiationStore;
    @Inject
    private ParticipantAgentService participantAgentService;

    @Provider
    private ContractNegotiationPendingGuard contractNegotiationPendingGuard() {
        return new DelegationEvidenceNegotiationPendingGuard(eventRouter, clock);
    }

    @Override
    public void initialize(ServiceExtensionContext context) {
        // register the scope extractor for mapping VCs to policy constraints
        scopeExtractorRegistry.registerScopeExtractor(new
DelegationEvidenceScopeExtractor());

        var deValidationService = new DelegationEvidenceValidationService();

        // create a binding of every ODRL action that the DE constraint will be used
with
        ruleBindingRegistry.bind(ODRL_USE_ACTION_ATTRIBUTE, TRANSFER_SCOPE);

        // first parameter needs to be the left operand of the ODRL DE constraint
        ruleBindingRegistry.bind("DelegationEvidence", TRANSFER_SCOPE);

        // String parameter needs to be the left operand of the ODRL DE constraint,
same as used for rule binding
        policyEngine.registerFunction(TransferProcessPolicyContext.class,
Permission.class,
                "DelegationEvidence", new
DelegationEvidencePolicyFunction(deValidationService));

        eventRouter.register(ContractNegotiationPendingEvent.class,
                new DelegationEvidenceEventSubscriber(deValidationService,
negotiationStore));

        // if an implementation of ParticipantAgentServiceExtension is created, this
can be registered as follows
        participantAgentService.register(...);
    }

}
```