Distributed Data Framework Architecture

Version 2.13.10. Copyright (c) Codice Foundation

Table of Contents

License

Copyright (c) Codice Foundation.

This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

This document last updated: 2019-06-14 03:38:38:947.

Developers will build or extend the functionality of the applications.

DDF includes several extension points where external developers can add functionality to support individual use cases.

DDF is written in Java and uses many open source libraries. DDF uses OSGi to provide modularity, lifecycle management, and dynamic services. OSGi services can be installed and uninstalled while DDF is running. DDF development typically means developing new OSGi bundles and deploying them to the running DDF. A complete description of OSGi is outside the scope of this documentation. For more information about OSGi, see the OSGi Alliance website ⬀.
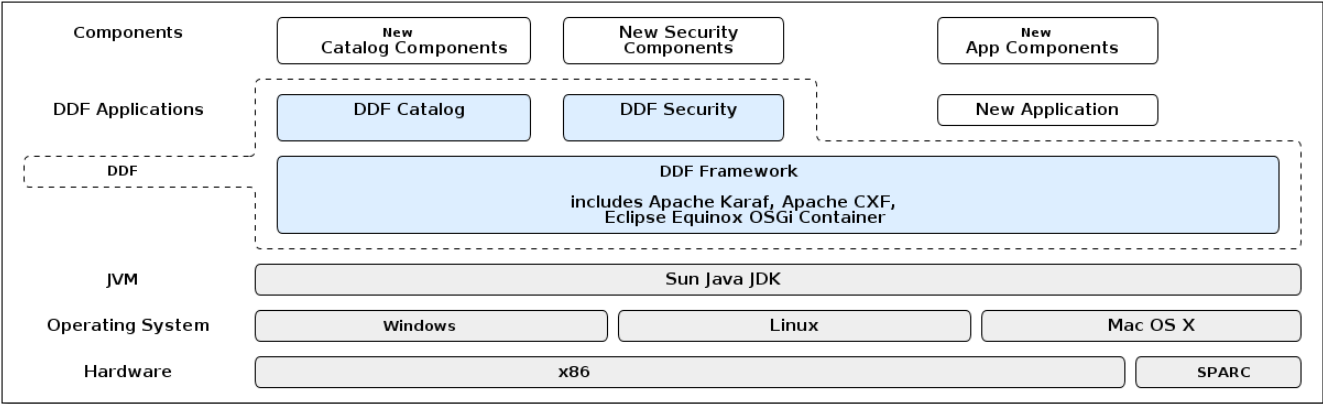
Architecture Diagram

IMPORTANT	If developing for a Highly Available Cluster of DDF, see High Availability Guidance.

1. Catalog Framework API
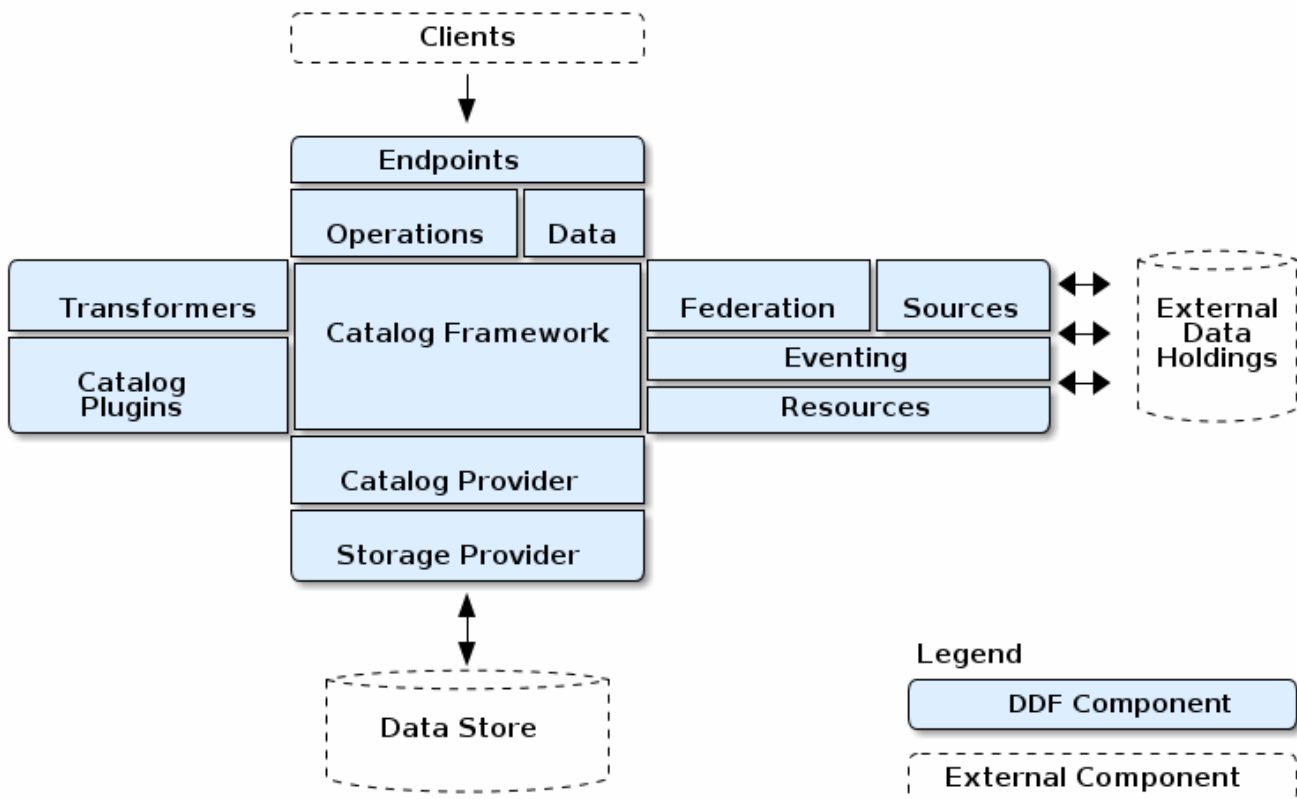
Catalog Architecture
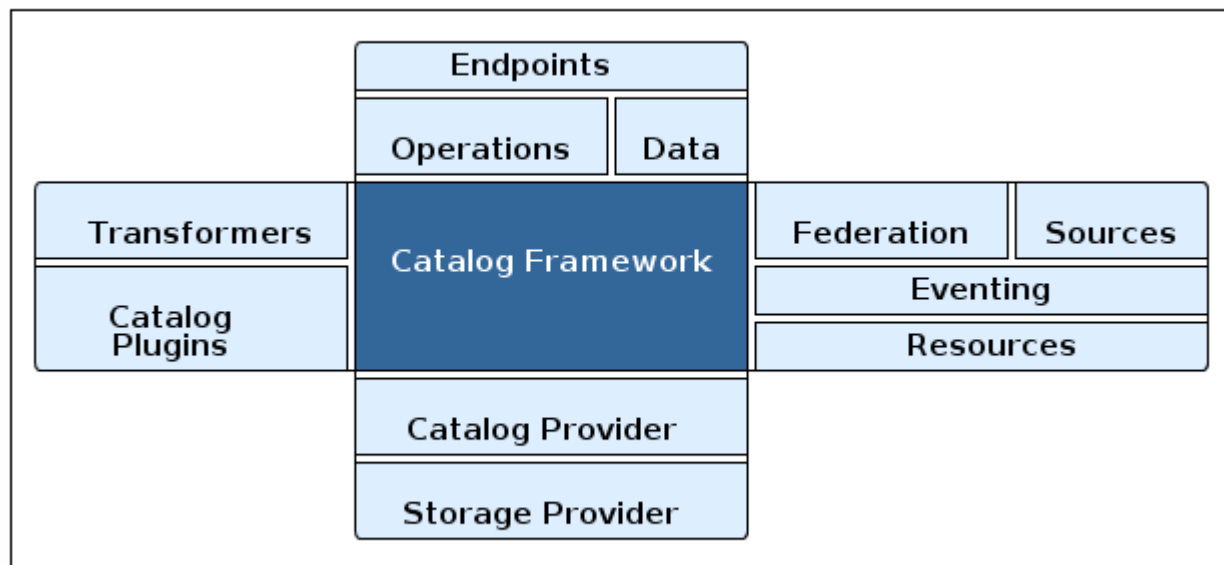
Catalog Framework Architecture

The `CatalogFramework` is the routing mechanism between catalog components that provides integration points for the Catalog Plugins. An endpoint invokes the active Catalog Framework, which calls any

configured Pre-query or Pre-ingest plug-ins. The selected federation strategy calls the active Catalog Provider and any connected or federated sources. Then, any Post-query or Post-ingest plug-ins are invoked. Finally, the appropriate response is returned to the calling endpoint.

The Catalog Framework wires all Catalog components together.

It is responsible for routing Catalog requests and responses to the appropriate target.

Endpoints send Catalog requests to the Catalog Framework. The Catalog Framework then invokes Catalog Plugins, Transformers, and Resource Components as needed before sending requests to the intended destination, such as one or more Sources.

The Catalog Framework decouples clients from service implementations and provides integration points for Catalog Plugins and convenience methods for Endpoint developers.

1.1. Catalog API Design

The Catalog is composed of several components and an API that connects them together. The Catalog API is central to DDF's architectural qualities of extensibility and flexibility. The Catalog API consists of Java interfaces that define Catalog functionality and specify interactions between components. These interfaces provide the ability for components to interact without a dependency on a particular underlying implementation, thus allowing the possibility of alternate implementations that can maintain interoperability and share developed components. As such, new capabilities can be developed independently, in a modular fashion, using the Catalog API interfaces and reused by other DDF installations.

1.1.1. Ensuring Compatibility

The Catalog API will evolve, but great care is taken to retain backwards compatibility with developed components. Compatibility is reflected in version numbers.

1.1.2. Catalog Framework Sequence Diagrams

Because the Catalog Framework plays a central role to Catalog functionality, it interacts with many different Catalog components. To illustrate these relationships, high-level sequence diagrams with notional class names are provided below. These examples are for illustrative purposes only and do not necessarily represent every step in each procedure.
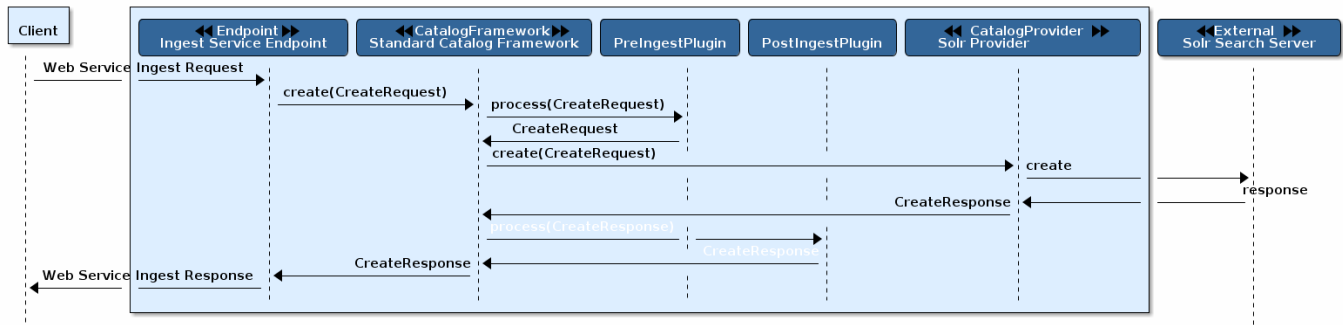
Ingest Request Data Flow

The Ingest Service Endpoint, the Catalog Framework, and the Catalog Provider are key components of the Reference Implementation. The Endpoint bundle implements a Web service that allows clients to create, update, and delete metacards. The Endpoint calls the `CatalogFramework` to execute the operations of its specification. The `CatalogFramework` routes the request through optional `PreIngest` and `PostIngest` Catalog Plugins, which may modify the ingest request/response before/after the Catalog Provider executes the ingest request and provides the response. Note that a `CatalogProvider` must be present for any ingest requests to be successfully processed, otherwise a fault is returned.

This process is similar for updating catalog entries, with update requests calling the `update(UpdateRequest)` methods on the Endpoint, `CatalogFramework`, and Catalog Provider. Similarly, for deletion of catalog entries, the delete requests call the `delete(DeleteRequest)` methods on the `Endpoint`, `CatalogFramework`, and `CatalogProvider`.

1.1.2.1. Error Handling

Any ingest attempts that fail inside the Catalog Framework (whether the failure comes from the Catalog Framework itself, pre-ingest plugin failures, or issues with the Catalog Provider) will be logged to a separate log file for ease of error handling. The file is located at `data/log/ingest_error.log` and will log the Metacards that fail, their ID and Title name, and the stack trace associated with their failure. By default, successful ingest attempts are not logged. However, that functionality can be achieved by setting the log level of the `ingestLogger` to DEBUG (note that enabling DEBUG can cause a non-trivial performance hit).

TIP	To turn off logging failed ingest attempts into a separate file, execute the following via the command line console
	``` log:set ERROR ingestLogger ```

1.1.2.2. Query
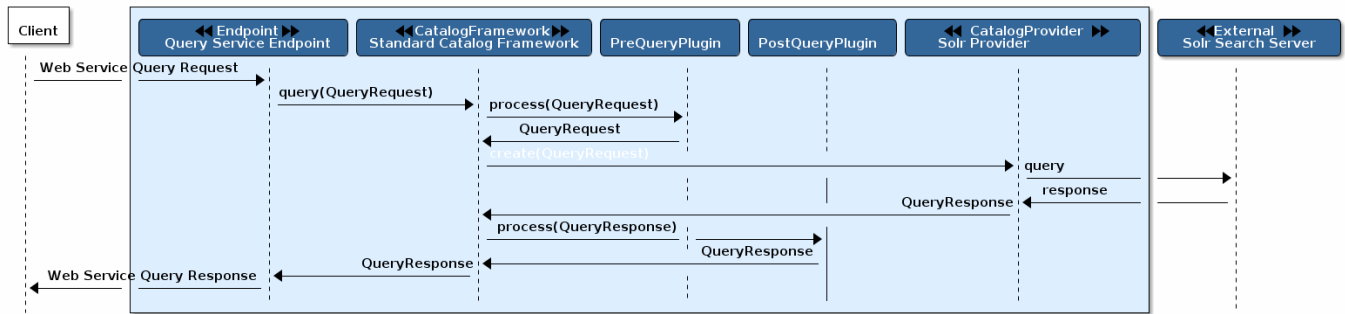
Query Request Data Flow

The Query Service Endpoint, the Catalog Framework, and the `CatalogProvider` are key components for processing a query request as well. The Endpoint bundle contains a Web service that exposes the interface to query for `Metacards`. The Endpoint calls the `CatalogFramework` to execute the operations of its specification. The `CatalogFramework` relies on the `CatalogProvider` to execute the actual query. Optional PreQuery and PostQuery Catalog Plugins may be invoked by the `CatalogFramework` to modify the query request/response prior to the Catalog Provider processing the query request and providing the query response. If a `CatalogProvider` is not configured and no other remote Sources are configured, a fault will be returned. It is possible to have only remote Sources configured and no local `CatalogProvider` configured and be able to execute queries to specific remote Sources by specifying the site name(s) in the query request.

1.1.2.3. Product Retrieval

The Query Service Endpoint, the Catalog Framework, and the `CatalogProvider` are key components for processing a retrieve product request. The Endpoint bundle contains a Web service that exposes the interface to retrieve products, also referred to as Resources. The Endpoint calls the `CatalogFramework` to execute the operations of its specification. The `CatalogFramework` relies on the Sources to execute the actual product retrieval. Optional `PreResource` and `PostResource` Catalog Plugins may be invoked by the `CatalogFramework` to modify the product retrieval request/response prior to the Catalog Provider processing the request and providing the response. It is possible to retrieve products from specific remote Sources by specifying the site name(s) in the request.

1.1.2.4. Product Caching

The Catalog Framework optionally provides caching of products, so future requests to retrieve the same product will be serviced much quicker. If caching is enabled, each time a retrieve product request is received, the Catalog Framework will look in its cache (default location `<DDF_HOME>/data/product-cache`) to see if the product has been cached locally. If it has, the product is retrieved from the local site and returned to the client, providing a much quicker turnaround because remote product retrieval and network traffic was avoided. If the requested product is not in the cache, the product is retrieved from the Source (local or remote) and cached locally while returning the product to the client. The caching to a local file of the product and the streaming of the product to the client are done simultaneously so that the client does not have to wait for the caching to complete before receiving the product. If errors are detected during the caching, caching of the product will be

abandoned, and the product will be returned to the client.

The Catalog Framework attempts to detect any network problems during the product retrieval, e.g., long pauses where no bytes are read implying a network connection was dropped. (The amount of time defined as a "long pause" is configurable, with the default value being five seconds.) The Catalog Framework will attempt to retrieve the product up to a configurable number of times (default = three), waiting for a configurable amount of time (default = 10 seconds) between each attempt, trying to successfully retrieve the product. If the Catalog Framework is unable to retrieve the product, an error message is returned to the client.

If the admin has enabled the **Always Cache When Canceled** option, caching of the product will occur even if the client cancels the product retrieval so that future requests will be serviced quickly. Otherwise, caching is canceled if the user cancels the product download.

1.1.2.5. Product Download Status

As part of the caching of products, the Catalog Framework also posts events to the OSGi notification framework. Information includes when the product download started, whether the download is retrying or failed (after the number of retrieval attempts configured for product caching has been exhausted), and when the download completes. These events are retrieved by the Search UI and presented to the user who initiated the download.

1.1.3. Catalog API

The Catalog API is an OSGi bundle (`catalog-core-api`) that contains the Java interfaces for the Catalog components and implementation classes for the Catalog Framework, Operations, and Data components.

1.1.3.1. Catalog API Search Interfaces

The Catalog API includes two different search interfaces.

Search UI Application Search Interface

The DDF Search UI application provides a graphic interface to return results and locate them on an interactive globe or map.

SSH Search Interface

Additionally, it is possible to use a client script to remotely access DDF via SSH and send console commands to search and ingest data.

1.1.3.2. Catalog Search Result Objects

Data is returned from searches as Catalog Search `Result` objects. This is a subtype of Catalog `Entry` that also contains additional data based on what type of sort policy was applied to the search. Because it is a subtype of Catalog `Entry`, a Catalog Search `Result` has all Catalog `Entry`'s fields such as metadata, effective time, and modified time. It also contains some of the following fields, depending on type of

search, that are populated by DDF when the search occurs:

Distance

Populated when a point-radius spatial search occurs. Numerical value that indicates the result's distance from the center point of the search.

Units

Populated when a point-radius spatial search occurs. Indicates the units (kilometer, mile, etc.) for the distance field.

Relevance

Populated when a contextual search occurs. Numerical value that indicates how relevant the text in the result is to the text originally searched for.

1.1.3.3. Search Programmatic Flow

Searching the catalog involves three basic steps:

1. Define the search criteria (contextual, spatial, or temporal).

    a. Optionally define a sort policy and assign it to the criteria.

    b. For contextual search, optionally set the `fuzzy` flag to `true` or `false` (the default value for the `Metadata Catalog fuzzy` flag is `true`, while the `portal` default value is `false`).

    c. For contextual search, optionally set the `caseSensitive` flag to true (the default is that `caseSensitive` flag is NOT set and queries are not case sensitive). Doing so enables case sensitive matching on the search criteria. For example, if `caseSensitive` is set to true and the phrase is "Baghdad" then only metadata containing "Baghdad" with the same matching case will be returned. Words such as "baghdad", "BAGHDAD", and "baghDad" will not be returned because they do not match the exact case of the search term.

2. Issue a search.

3. Examine the results.

1.1.3.4. Sort Policies

Searches can also be sorted according to various built-in policies. A sort policy is applied to the search criteria after its creation but before the search is issued. The policy specifies to the DDF the order the Catalog search results should be in when they are returned to the requesting client. Only one sort policy may be defined per search.

There are three policies available.

Table 1. Sort Policies

Sort Policy	Sorts By	Default Order	Available for
Temporal	The catalog search result's effective time field	Newest to oldest	All Search Types
Distance	The catalog search result's distance field	Nearest to farthest	Point-Radius Spatial searches
Relevance	The catalog search result's relevance field	Most to least relevant	Contextual

If no sort policy is defined for a particular search, the temporal policy will automatically be applied.

1.1.3.5. Asynchronous Search & Retrieval

Asynchronous Search & Retrieval allows a requestor to execute multiple queries at once, begin multiple product downloads while query results are being returned, cancel queries and downloads, and receive status on the state of incoming query results and product downloads.

Table 2. Important Terms for Asynchronous Search

Capability	Description	Endpoint Integration
Asynchronous Search	Search multiple sources simultaneously	Search UI
Product caching	Allows quick retrieval of already downloaded products	Catalog
Canceling Product Downloads	The ability to cancel a download in progress	Catalog
Activities	Activities * `download` * `retry` * `cancel` * `pause` * `remove` * `resume`	Catalog, CometD endpoint
Notifications	Time-stamped messages of an action	Catalog, Search UI/CometD endpoint
Workspaces	Ability to save and manage queries and save metacards	Platform, Search UI/CometD endpoint
3D Map support	Ability to execute a geospatial search using a 3D map	N/A

1.1.3.6. Product Retrieval

The DDF is used to catalog resources. A Resource is a URI-addressable entity that is represented by a Metacard. Resources may also be known as products or data. Resources may exist either locally or on a remote data store.

Examples of Resources

- NITF image

- MPEG video

- Live video stream

- Audio recording

- Document

Product Retrieval Services

- SOAP Web services

- DDF JSON

- DDF REST

The Query Service Endpoint, the Catalog Framework, and the `CatalogProvider` are key components for processing a retrieve product request. The Endpoint bundle contains a Web service that exposes the interface to retrieve products, also referred to as Resources. The Endpoint calls the `CatalogFramework` to execute the operations of its specification. The `CatalogFramework` relies on the Sources to execute the actual product retrieval. Optional PreResource and PostResource Catalog Plugins may be invoked by the `CatalogFramework` to modify the product retrieval request/response prior to the Catalog Provider processing the request and providing the response. It is possible to retrieve products from specific remote Sources by specifying the site name(s) in the request.

NOTE	*Product Caching*
	Existing DDF clients are able to leverage product caching due to the product cache being implemented in the DDF. Enabling the product cache is an administrator function.
	Product Caching is enabled by default.

To configure product caching:

1. Navigate to the **Admin Console**.

2. Select Catalog.

3. Select **Configuration**.

4. Select **Resource Download Settings**.

See Resource Download Settings configurations for all possible configurations.
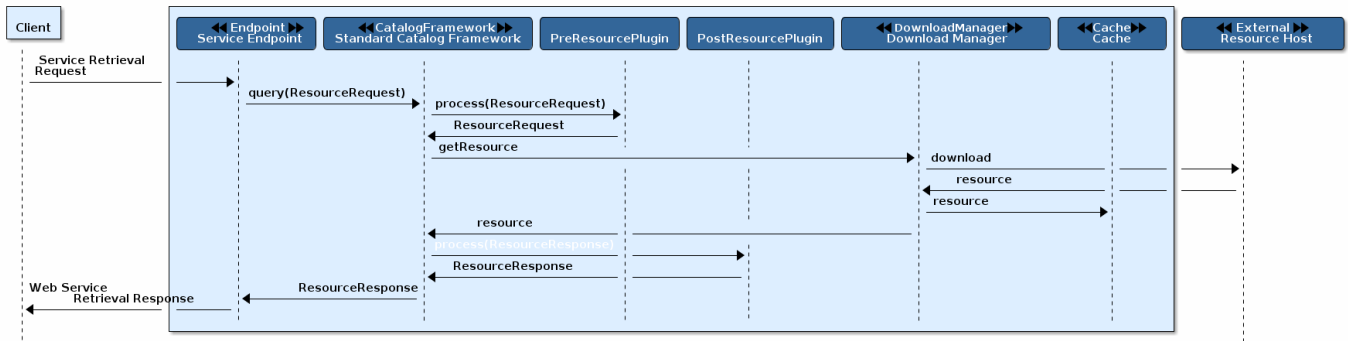
Product Retrieval Request

1.1.3.7. Notifications and Activities

DDF can send/receive notifications of "Activities" occuring in the system.

1.1.3.7.1. Notifications

Currently, the notifications provide information about product retrieval only. For example, in the DDF Search UI, after a user initiates a resource download, they receive notifications when the download completed, failed, canceled, or is being retried.

1.1.3.7.2. Activities

Activity events include the status and progress of actions that are being performed by the user, such as searches and downloads. Activities can be enabled by selecting "Show tasks" in the Standard Search UI configuration. A list of all activities opens in a drop-down menu, from which activities can be read and deleted. If a download activity is being performed, the Activity drop-down menu provides the link to retrieve the product. If caching is enabled, a progress bar is displayed in the Activity (Product Retrieval) drop-down menu until the action being performed is complete.

1.2. Included Catalog Frameworks, Associated Components, and Configurations

These catalog frameworks are available in a standard DDF installation:

Standard Catalog Framework

Reference implementation of a Catalog Framework that implements all requirements of the Catalog API.

Catalog Framework Camel Component

Supports creating, updating, and deleting metacards using the Catalog Framework from a Camel route.

1.2.1. Standard Catalog Framework

The Standard Catalog Framework provides the reference implementation of a Catalog Framework that implements all requirements of the Catalog API. `CatalogFrameworkImpl` is the implementation of the DDF Standard Catalog Framework.

The Standard Catalog Framework is the core class of DDF. It provides the methods for create, update, delete, and resource retrieval (CRUD) operations on the `Sources`. By contrast, the Fanout Catalog Framework only allows for query and resource retrieval operations, no catalog modifications, and all queries are enterprise-wide.

Use this framework if:

- access to a catalog provider is required to create, update, and delete catalog entries.
- queries to specific sites are required.
- queries to only the local provider are required.

It is possible to have only remote Sources configured with no local `CatalogProvider` configured and be able to execute queries to specific remote sources by specifying the site name(s) in the query request.

The Standard Catalog Framework also maintains a list of `ResourceReaders` for resource retrieval operations. A resource reader is matched to the scheme (i.e., protocol, such as `file://`) in the URI of the resource specified in the request to be retrieved.

Site information about the catalog provider and/or any federated source(s) can be retrieved using the Standard Catalog Framework. Site information includes the source's name, version, availability, and the list of unique content types currently stored in the source (e.g., NITF). If no local catalog provider is configured, the site information returned includes site info for the catalog framework with no content types included.

1.2.1.1. Installing the Standard Catalog Framework

The Standard Catalog Framework is bundled as the `catalog-core-standardframework` feature and can be installed and uninstalled using the normal processes described in Configuration.

1.2.1.2. Configuring the Standard Catalog Framework

These are the configurable properties on the Standard Catalog Framework.

See Catalog Standard Framework configurations for all possible configurations.

Table 3. Standard Catalog Framework Exported Services

Registered Interface	Service Property	Value
`ddf.catalog.federation.FederationStrategy`	shortname	`sorted`

Registered Interface	Service Property	Value

`org.osgi.service.event.EventHandler`	event.topics	`ddf/catalog/event/CREATED,`
		`ddf/catalog/event/UPDATED,`
		`ddf/catalog/event/DELETED`

`ddf.catalog.CatalogFramework`		
`ddf.catalog.event.EventProcessor`		
`ddf.catalog.plugin.PostIngestPlugin`		

Table 4. Standard Catalog Framwork Imported Services

Registered Interface	Availability	Multiple
`ddf.catalog.plugin.PostFederatedQueryPlugin`	optional	`true`
`ddf.catalog.plugin.PostIngestPlugin`	optional	`true`
`ddf.catalog.plugin.PostQueryPlugin`	optional	`true`
`ddf.catalog.plugin.PostResourcePlugin`	optional	`true`
`ddf.catalog.plugin.PreDeliveryPlugin`	optional	`true`
`ddf.catalog.plugin.PreFederatedQueryPlugin`	optional	`true`
`ddf.catalog.plugin.PreIngestPlugin`	optional	`true`
`ddf.catalog.plugin.PreQueryPlugin`	optional	`true`
`ddf.catalog.plugin.PreResourcePlugin`	optional	`true`
`ddf.catalog.plugin.PreSubscriptionPlugin`	optional	`true`
`ddf.catalog.plugin.PolicyPlugin`	optional	`true`
`ddf.catalog.plugin.AccessPlugin`	optional	`true`
`ddf.catalog.resource.ResourceReader`	optional	`true`
`ddf.catalog.source.CatalogProvider`	optional	`true`
`ddf.catalog.source.ConnectedSource`	optional	`true`
`ddf.catalog.source.FederatedSource`	optional	`true`
`ddf.cache.CacheManager`		`false`
`org.osgi.service.event.EventAdmin`		`false`

1.2.1.3. Known Issues with Standard Catalog Framework

None.

1.2.2. Catalog Framework Camel Component

The Catalog Framework Camel Component supports creating, updating, and deleting metacards using the Catalog Framework from a Camel route.

URI Format

```
catalog:framework
```

1.2.2.1. Message Headers

1.2.2.1.1. Catalog Framework Producer

Header	Description
operation	the operation to perform using the Catalog Framework (possible values are CREATE | UPDATE | DELETE)

1.2.2.2. Sending Messages to Catalog Framework Endpoint

1.2.2.2.1. Catalog Framework Producer

In Producer mode, the component provides the ability to provide different inputs and have the Catalog framework perform different operations based upon the header values.

For the CREATE and UPDATE operation, the message body can contain a list of metacards or a single metacard object.

For the DELETE operation, the message body can contain a list of strings or a single string object. The string objects represent the IDs of metacards to be deleted. The exchange's "in" message will be set with the affected metacards. In the case of a CREATE, it will be updated with the created metacards. In the case of the UPDATE, it will be updated with the updated metacards and with the DELETE it will contain the deleted metacards.

Table 5. Catalog Framework Camel Component Operations

Header	Message Body (Input)	Exchange Modification (Output)
operation = CREATE	List<Metacard> or Metacard	exchange.getIn().getBody() updated with List of Metacards created
operation = UPDATE	List<Metacard> or Metacard	exchange.getIn().getBody() updated with List of Metacards updated
operation = DELETE	List<String> or String (representing metacard IDs)	exchange.getIn().getBody() updated with List of Metacards deleted

NOTE	If there is an exception thrown while the route is being executed, a FrameworkProducerException will be thrown causing the route to fail with a CamelExecutionException.

1.2.2.2.2. Samples

This example demonstrates:

1. Reading in some sample data from the file system.

2. Using a Java bean to convert the data into a metacard.

3. Setting a header value on the Exchange.

4. Sending the Metacard to the Catalog Framework component for ingesting.

```
<route>
 <from uri="file:data/sampleData?noop=true"/>
    <bean ref="sampleDataToMetacardConverter" method="covertToMetacard"/>\
   <setHeader headerName="operation">
  <constant>CREATE</constant>
 </setHeader>
    <to uri="catalog:framework"/>
</route>
```

2. Transformers
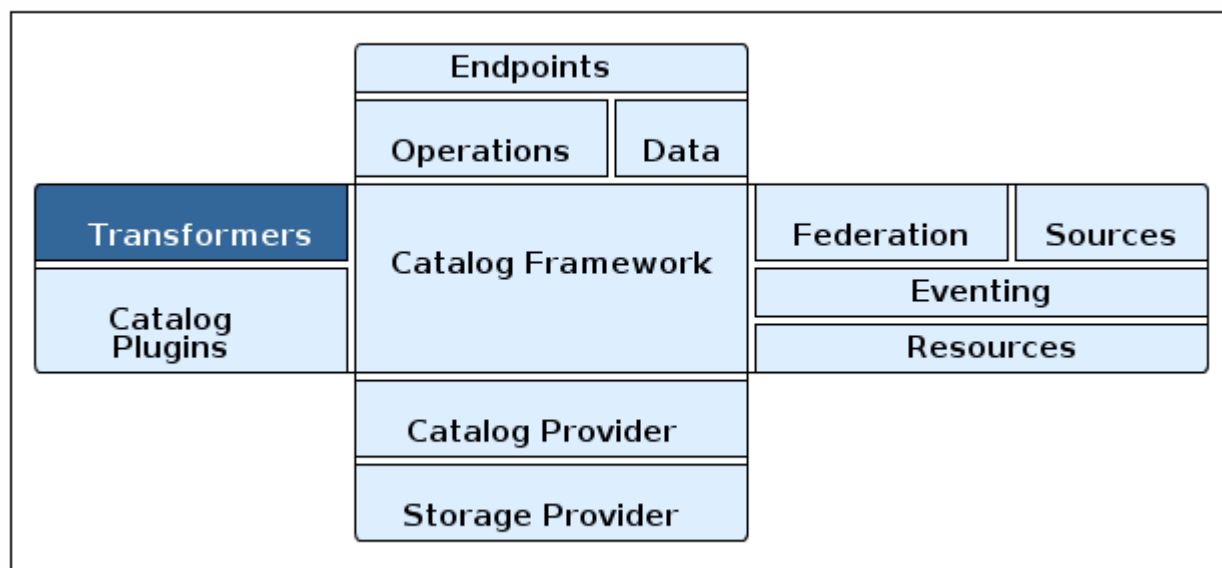
Transformers

Transformers transform data to and from various formats. Transformers are categorized by when they are invoked and used. The existing types are Input transformers, Metacard transformers, and Query Response transformers. Additionally, XSLT transformers are provided to aid in developing custom, lightweight Metacard and Query Response transformers.

Transformers are utility objects used to transform a set of standard DDF components into a desired format, such as into PDF, GeoJSON, XML, or any other format. For instance, a transformer can be used to convert a set of query results into an easy-to-read GeoJSON format (GeoJSON Transformer) or

convert a set of results into a RSS feed that can be easily published to a URL for RSS feed subscription. Transformers can be registered in the OSGi Service Registry so that any other developer can access them based on their standard interface and self-assigned identifier, referred to as its "shortname." Transformers are often used by endpoints for data conversion in a system standard way. Multiple endpoints can use the same transformer, a different transformer, or their own published transformer.

WARNING	The current transformers only work for UTF-8 characters and do not support Non-Western Characters (for example, Hebrew). It is recommend not to use international character sets, as they may not be displayed properly.
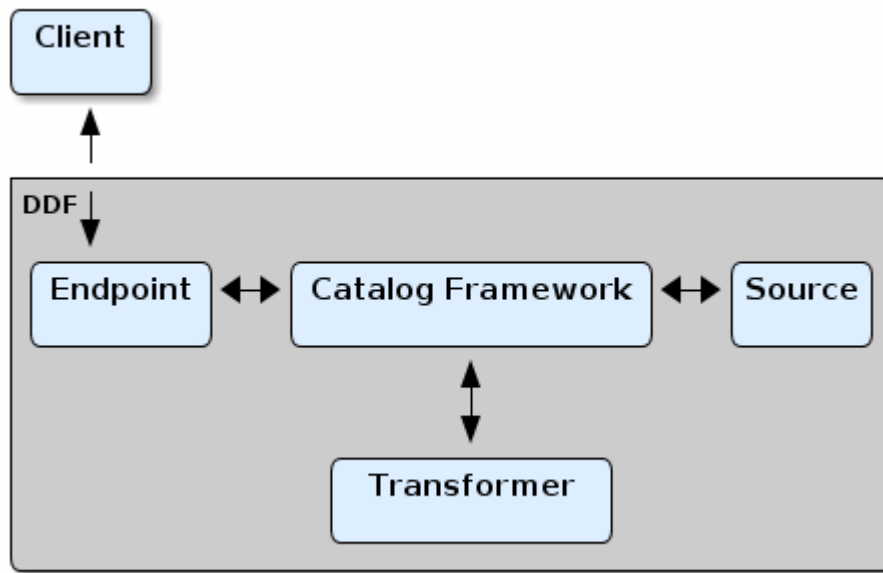
Communication Diagram

Transformers are used to alter the format of a resource or its metadata to or from the catalog's metacard format.

Types of Transformers

Input Transformers

Input Transformers create metacards from input. Once converted to a Metacard, the data can be used in a variety of ways, such as in an `UpdateRequest`, `CreateResponse`, or within Catalog Endpoints or Sources. For instance, an input transformer could be used to receive and translate XML into a Metacard so that it can be placed within a `CreateRequest` to be ingested within the Catalog. Input transformers should be registered within the Service Registry with the interface `ddf.catalog.transform.InputTransformer` to notify Catalog components of any new transformers.

Metacard Transformers

Metacard Transformers translate a metacard from catalog metadata to a specific data format.

Query Response Transformers

Query Response transformers convert query responses into other data formats.

2.1. Available Input Transformers

The following input transformers are available in a standard installation of DDF:

GeoJSON Input Transformer

Translates GeoJSON into a Catalog metacard.

PDF Input Transformer

Translates a PDF document into a Catalog Metacard.

PPTX Input Transformer

Translates Microsoft PowerPoint (OOXML only) documents into Catalog Metacards.

Registry Transformer

Creates Registry metacards from `ebrim` messages and translates a Registry metacard. (used by the Registry application)

Tika Input Transformer

Translates Microsoft Word, Microsoft Excel, Microsoft PowerPoint, OpenOffice Writer, and PDF documents into Catalog records.

Video Input Transformer

Creates Catalog metacards from certain video file types.

XML Input Transformer

Translates an XML document into a Catalog Metacard.

2.2. Available Metacard Transformers

The following metacard transformers are available in a standard installation of DDF:

GeoJSON Metacard Transformer

Translates a metacard into GeoJSON.

HTML Metacard Transformer

Translates a metacard into an HTML-formatted document.

KML Metacard Transformer

Translates a metacard into a KML-formatted document.

KML Style Mapper

Maps a KML Style URL to a metacard based on that metacard's attributes.

Metadata Metacard Transformer

returns the `Metacard.METADATA` attribute when given a metacard.

Registry Transformer

Creates Registry metacards from `ebrim` messages and translates a Registry metacard. (used by the Registry application)

Resource Metacard Transformer

Retrieves the resource bytes of a metacard by returning the product associated with the metacard.

Thumbnail Metacard Transformer

Retrieves the thumbnail bytes of a Metacard by returning the `Metacard.THUMBNAIL` attribute value.

XML Metacard Transformer

Translates a metacard into an XML-formatted document.

2.3. Available Query Response Transformers

The following query response transformers are available in a standard installation of DDF:

Atom Query Response Transformer

Transforms a query response into an Atom 1.0 feed.

CSW Query Response Transformer

Transforms a query response into a CSW-formatted document.

GeoJSON Query Response Transformer

Translates a query response into a GeoJSON-formatted document.

KML Query Response Transformer

Translates a query response into a KML-formatted document.

Query Response Transformer Consumer

Translates a query response into a Catalog Metacard.

XML Query Response Transformer

Translates a query response into an XML-formatted document.

2.4. Transformers Details

Availability and configuration details of available transformers.

2.4.1. Atom Query Response Transformer

The Atom Query Response Transformer transforms a query response into an Atom 1.0 feed. The Atom transformer maps a `QueryResponse` object as described in the Query Result Mapping.

2.4.1.1. Installing the Atom Query Response Transformer

The Atom Query Response Transformer is installed by default with a standard installation.

2.4.1.2. Configuring the Atom Query Response Transformer

The Atom Query Response Transformer has no configurable properties.

2.4.1.3. Using the Atom Query Response Transformer

Use this transformer when Atom is the preferred medium of communicating information, such as for feed readers or federation. An integrator could use this with an endpoint to transform query responses into an Atom feed.

For example, clients can use the OpenSearch Endpoint. The client can query with the format option set to the shortname, `atom`.

Sample OpenSearch Query with Atom Specified as Return Format

```
http://{FQDN}:{PORT}/services/catalog/query?q=ddf?format=atom
```

Developers could use this transformer to programmatically transform `QueryResponse` objects on the fly.

Sample Atom Feed from `QueryResponse` object

```xml
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:os="http://a9.com/-
/spec/opensearch/1.1/">
    <title type="text">Query Response</title>
    <updated>2017-01-31T23:22:37.298Z</updated>
    <id>urn:uuid:a27352c9-f935-45f0-9b8c-5803095164bb</id>
    <link href="#" rel="self" />
    <author>
        <name>Organization Name</name>
    </author>
    <generator version="2.1.0.20130129-1341">ddf123</generator>
    <os:totalResults>1</os:totalResults>
    <os:itemsPerPage>10</os:itemsPerPage>
    <os:startIndex>1</os:startIndex>
    <entry xmlns:relevance="http://a9.com/-/opensearch/extensions/relevance/1.0/"
xmlns:fs="http://a9.com/-/opensearch/extensions/federation/1.0/"
        xmlns:georss="http://www.georss.org/georss">
        <fs:resultSource fs:sourceId="ddf123" />
```

```xml
        <relevance:score>0.19</relevance:score>
        <id>urn:catalog:id:ee7a161e01754b9db1872bfe39d1ea09</id>
        <title type="text">F-15 lands in Libya; Crew Picked Up</title>
        <updated>2013-01-31T23:22:31.648Z</updated>
        <published>2013-01-31T23:22:31.648Z</published>
        <link href=
"http://123.45.67.123:8181/services/catalog/ddf123/ee7a161e01754b9db1872bfe39d1ea09" rel
="alternate" title="View Complete Metacard" />
        <category term="Resource" />
        <georss:where xmlns:gml="http://www.opengis.net/gml">
            <gml:Point>
                <gml:pos>32.8751900768792 13.1874561309814</gml:pos>
            </gml:Point>
        </georss:where>
        <content type="application/xml">
            <ns3:metacard xmlns:ns3="urn:catalog:metacard" xmlns:ns2=
"http://www.w3.org/1999/xlink" xmlns:ns1="http://www.opengis.net/gml"
                xmlns:ns4="http://www.w3.org/2001/SMIL20/" xmlns:ns5=
"http://www.w3.org/2001/SMIL20/Language" ns1:id="4535c53fc8bc4404a1d32a5ce7a29585">
                <ns3:type>ddf.metacard</ns3:type>
                <ns3:source>ddf.distribution</ns3:source>
                <ns3:geometry name="location">
                    <ns3:value>
                        <ns1:Point>
                            <ns1:pos>32.8751900768792 13.1874561309814</ns1:pos>
                        </ns1:Point>
                    </ns3:value>
                </ns3:geometry>
                <ns3:dateTime name="created">
                    <ns3:value>2013-01-31T16:22:31.648-07:00</ns3:value>
                </ns3:dateTime>
                <ns3:dateTime name="modified">
                    <ns3:value>2013-01-31T16:22:31.648-07:00</ns3:value>
                </ns3:dateTime>
                <ns3:stringxml name="metadata">
                    <ns3:value>
                        <ns6:xml xmlns:ns6="urn:sample:namespace" xmlns=
"urn:sample:namespace">Example description.</ns6:xml>
                    </ns3:value>
                </ns3:stringxml>
                <ns3:string name="metadata-content-type-version">
                    <ns3:value>myVersion</ns3:value>
                </ns3:string>
                <ns3:string name="metadata-content-type">
                    <ns3:value>myType</ns3:value>
                </ns3:string>
                <ns3:string name="title">
                    <ns3:value>Example title</ns3:value>
```

```
                </ns3:string>
            </ns3:metacard>
        </content>
    </entry>
</feed>
```

Table 6. Atom Query Response Transformer Result Mapping

XPath to Atom XML	Value
/feed/title	"Query Response"
/feed/updated	ISO 8601 dateTime of when the feed was generated
/feed/id	Generated UUID URN ⧉
/feed/author/name	Platform Global Configuration organization
/feed/generator	Platform Global Configuration site name
/feed/generator/@version	Platform Global Configuration version
/feed/os:totalResults	SourceResponse Number of Hits
/feed/os:itemsPerPage	Request's Page Size
/feed/os:startIndex	Request's Start Index
/feed/entry/fs:resultSource/@fs:sourceId	Source Id from which the Result came. `Metacard.getSourceId()`
/feed/entry/relevance:score	Result's relevance score if applicable. `Result.getRelevanceScore()`
/feed/entry/id	`urn:catalog:id:<Metacard.ID>`
/feed/entry/title	`Metacard.TITLE`
/feed/entry/updated	ISO 8601 dateTime of `Metacard.MODIFIED`
/feed/entry/published	ISO 8601 dateTime of `Metacard.CREATED`
/feed/entry/link[@rel='related']	URL to retrieve underlying resource (if applicable and link is available)

XPath to Atom XML	Value
`/feed/entry/link[@rel='alternate']`	Link to alternate view of the Metacard (if a link is available)
`/feed/entry/category`	`Metacard.CONTENT_TYPE`
`/feed/entry//georss:where`	GeoRSS GML of every Metacard attribute with format `AttributeFormat.GEOMETRY`
`/feed/entry/content`	Metacard XML generated by `DDF.catalog.transform.MetacardTransformer` with `shortname=xml`. If no transformer found, `/feed/entry/content/@type` will be text and `Metacard.ID` is displayed `<content type="text">4e1f38d1913b4e93ac622e6c1b258f89</content>`

2.4.2. CSW Query Response Transformer

The CSW Query Response Transformer transforms a query response into a CSW-formatted document.

2.4.2.1. Installing the CSW Query Response Transformer

The CSW Query Response Transformer is installed by default with a standard installation in the Spatial application.

2.4.2.2. Configuring the CSW Query Response Transformer

The CSW Query Response Transformer has no configurable properties.

2.4.3. GeoJSON Input Transformer

The GeoJSON input transformer is responsible for translating GeoJSON into a Catalog metacard.

Table 7. GeoJSON Input Transformer Usage

Schema	Mime-types
N/A	`application/json`

2.4.3.1. Installing the GeoJSON Input Transformer

The GeoJSON Input Transformer is installed by default with a standard installation.

2.4.3.2. Configuring the GeoJSON Input Transformer

The GeoJSON Input Transformer has no configurable properties.

2.4.3.3. Using the GeoJSON Input Transformer

Using the REST Endpoint, for example, HTTP POST a GeoJSON metacard to the Catalog. Once the REST Endpoint receives the GeoJSON Metacard, it is converted to a Catalog metacard.

Example HTTP POST of a Local `metacard.json` *File Using the Curl Command*

```
curl -X POST -i -H "Content-Type: application/json" -d "@metacard.json"
https://{FQDN}:{PORT}/services/catalog
```

2.4.3.4. Conversion to a Metacard

A GeoJSON object consists of a single JSON object. This can be a geometry, a feature, or a `FeatureCollection`. The GeoJSON input transformer only converts "feature" objects into metacards because feature objects include geometry information and a list of properties. A geometry object alone does not contain enough information to create a metacard. Additionally, the input transformer currently does not handle `FeatureCollection`s.

IMPORTANT

Cannot create Metacard from this limited GeoJSON

```
{ "type": "LineString",
  "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]
  }
```

The following sample *will* create a valid metacard:

Sample Parseable GeoJson (Point)

```json
{
    "properties": {
        "title": "myTitle",
        "thumbnail": "CA==",
        "resource-uri": "http://example.com",
        "created": "2012-09-01T00:09:19.368+0000",
        "metadata-content-type-version": "myVersion",
        "metadata-content-type": "myType",
        "metadata": "<xml></xml>",
        "modified": "2012-09-01T00:09:19.368+0000"
    },
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [
            30.0,
            10.0
        ]
    }
}
```

In the current implementation, `Metacard.LOCATION` is not taken from the properties list as WKT, but instead interpreted from the `geometry` JSON object. The geometry object is formatted according to the GeoJSON standard. Dates are in the ISO 8601 standard. White space is ignored, as in most cases with JSON. Binary data is accepted as Base64. XML must be properly escaped, such as what is proper for normal JSON.

Currently, only **Required Attributes** are recognized in the properties.

2.4.3.4.1. Metacard Extensibility

GeoJSON supports custom, extensible properties on the incoming GeoJSON using DDF's extensible metacard support. To have those customized attributes understood by the system, a corresponding `MetacardType` must be registered with the `MetacardTypeRegistry`. That `MetacardType` must be specified by name in the metacard-type property of the incoming GeoJSON. If a `MetacardType` is specified on the GeoJSON input, the customized properties can be processed, cataloged, and indexed.

Sample GeoJSON input

```
{
    "properties": {
        "title": "myTitle",
        "thumbnail": "CA==",
        "resource-uri": "http://example.com",
        "created": "2012-09-01T00:09:19.368+0000",
        "metadata-content-type-version": "myVersion",
        "metadata-content-type": "myType",
        "metadata": "<xml></xml>",
        "modified": "2012-09-01T00:09:19.368+0000",
        "min-frequency": "10000000",
        "max-frequency": "20000000",
        "metacard-type": "ddf.metacard.custom.type"
    },
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [
            30.0,
            10.0
        ]
    }
}
```

When the GeoJSON Input Transformer gets GeoJSON with the `MetacardType` specified, it will perform a lookup in the `MetacardTypeRegistry` to obtain the specified `MetacardType` in order to understand how to parse the GeoJSON. If no `MetacardType` is specified, the GeoJSON Input Transformer will assume the default `MetacardType`. If an unregistered `MetacardType` is specified, an exception will be returned to the client indicating that the `MetacardType` was not found.

2.4.3.5. Usage Limitations of the GeoJSON Input Transformer

The GeoJSON Input Transformer does not handle multiple geometries.

2.4.4. GeoJSON Metacard Transformer

GeoJSON Metacard Transformer translates a metacard into GeoJSON.

2.4.4.1. Installing the GeoJSON Metacard Transformer

The GeoJSON Metacard Transformer is not installed by default with a standard installation.

To install:

1. Navigate to the **Admin Console**.

2. Select the **System** tab.

3. Select the **Features** tab.

4. Install the `catalog-transformer-json` feature.

2.4.4.2. Configuring the GeoJSON Metacard Transformer

The GeoJSON Metacard Transformer has no configurable properties.

2.4.4.3. Using the GeoJSON Metacard Transformer

The GeoJSON Metacard Transformer can be used programmatically by requesting a `MetacardTransformer` with the id `geojson`. It can also be used within the REST Endpoint by providing the transform option as `geojson`.

Example REST GET Method with the GeoJSON Metacard Transformer

```
https://{FQDN}:{PORT}/services/catalog/0123456789abcdef0123456789abcdef?transform=geojson
```

Example REST GET Output from the GeoJSON Metacard Transformer

```json
{
    "properties":{
        "title":"myTitle",
        "thumbnail":"CA==",
        "resource-uri":"http:\/\/example.com",
        "created":"2012-08-31T23:55:19.518+0000",
        "metadata-content-type-version":"myVersion",
        "metadata-content-type":"myType",
        "metadata":"<xml>text<\/xml>",
        "modified":"2012-08-31T23:55:19.518+0000",
        "metacard-type": "ddf.metacard"
    },
    "type":"Feature",
    "geometry":{
        "type":"LineString",
        "coordinates":[
            [
                30.0,
                10.0
            ],
            [
                10.0,
                30.0
            ],
            [
                40.0,
                40.0
            ]
        ]
    }
}
```

2.4.5. GeoJSON Query Response Transformer

The GeoJSON Query Response Transformer translates a query response into a GeoJSON-formatted document.

2.4.5.1. Installing the GeoJSON Query Response Transformer

The GeoJSON Query Response Transformer is installed by default with a standard installation in the Catalog application.

2.4.5.2. Configuring the GeoJSON Query Response Transformer

The GeoJSON Query Response Transformer has no configurable properties.

2.4.6. HTML Metacard Transformer

The HTML metacard transformer is responsible for translating a metacard into an HTML-formatted document.

2.4.6.1. Installing the HTML Metacard Transformer

The HTML Metacard Transformer is installed by default with a standard installation in the Search UI application.

2.4.6.2. Configuring the HTML Metacard Transformer

The HTML Metacard Transformer has no configurable properties.

2.4.6.3. Using the HTML Metacard Transformer

Using the REST Endpoint for example, request a metacard with the transform option set to the HTML shortname.

```
https://{FQDN}:{PORT}/services/catalog/0123456789abcdef0123456789abcdef?transform=html
```

HTML Metacard Transformer Example Output

```
html metacard.png
```

2.4.7. KML Metacard Transformer

The KML Metacard Transformer is responsible for translating a metacard into a KML-formatted document. The KML will contain an HTML description that will display in the pop-up bubble in Google Earth. The HTML contains links to the full metadata view as well as the product.

2.4.7.1. Installing the KML Metacard Transformer

The KML Metacard Transformer is installed by default with a standard installation in the Spatial Application.

2.4.7.2. Configuring the KML Metacard Transformer

The KML Metacard Transformer has no configurable properties.

2.4.7.3. Using the KML Metacard Transformer

Using the REST Endpoint for example, request a metacard with the transform option set to the KML shortname.

KML Metacard Transformer Example Output

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<kml xmlns:ns2="http://www.google.com/kml/ext/2.2" xmlns="http://www.opengis.net/kml/2.2"
xmlns:ns4="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0" xmlns:ns3=
"http://www.w3.org/2005/Atom">
  <Placemark id="Placemark-0103c77e66d9428d8f48fab939da528e">
    <name>MultiPolygon</name>
    <description>&lt;!DOCTYPE html&gt;
    &lt;html&gt;
      &lt;head&gt;
        &lt;meta content="text/html; charset=windows-1252" http-equiv="content-type"&gt;
        &lt;style media="screen" type="text/css"&gt;
          .label {
            font-weight: bold
          }
          .linkTable {
width: 100% }
          .thumbnailDiv {
            text-align: center
          }
          img {
            max-width: 100px;
            max-height: 100px;
            border-style:none
          }
    &lt;/style&gt;
  &lt;/head&gt;
  &lt;body&gt;
        &lt;div class="thumbnailDiv"&gt;&lt;a
 href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f48
fab939da528e?transform=resource"&gt;&lt;img alt="Thumnail"
src="data:image/jpeg;charset=utf-8;base64, CA=="&gt;&lt;/a&gt;&lt;/div&gt;
    &lt;table&gt;
      &lt;tr&gt;
        &lt;td class="label"&gt;Source:&lt;/td&gt;
        &lt;td&gt;ddf.distribution&lt;/td&gt;
      &lt;/tr&gt;
      &lt;tr&gt;
```

```
        &lt;td class="label"&gt;Created:&lt;/td&gt;
          &lt;td&gt;Wed Oct 30 09:46:29 MDT 2013&lt;/td&gt;
      &lt;/tr&gt;
      &lt;tr&gt;
      &lt;td class="label"&gt;Effective:&lt;/td&gt;
          &lt;td&gt;2014-01-07T14:58:16-0700&lt;/td&gt;
      &lt;/tr&gt;
    &lt;/table&gt;
    &lt;table class="linkTable"&gt;
      &lt;tr&gt;
        &lt;td&gt;&lt;a
href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f48
fab939da528e?transform=html"&gt;View Details...&lt;/a&gt;&lt;/td&gt;
        &lt;td&gt;&lt;a
href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f48
fab939da528e?transform=resource"&gt;Download...&lt;/a&gt;&lt;/td&gt;
      &lt;/tr&gt;
    &lt;/table&gt;
  &lt;/body&gt;
&lt;/html&gt;
</description>
    <TimeSpan>
      <begin>2014-01-07T21:58:16</begin>
    </TimeSpan>
    <Style id="bluenormal">
      <LabelStyle>
        <scale>0.0</scale>
      </LabelStyle>
      <LineStyle>
        <color>33ff0000</color>
        <width>3.0</width>
      </LineStyle>
      <PolyStyle>
        <color>33ff0000</color>
        <fill xsi:type="xs:boolean" xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">true</fill>
      </PolyStyle>
      <BalloonStyle>
<text>&lt;h3&gt;&lt;b&gt;$[name]&lt;/b&gt;&lt;/h3&gt;&lt;table&gt;&lt;tr&gt;&lt;td
width="400"&gt;$[description]&lt;/td&gt;&lt;/tr&gt;&lt;/table&gt;</text>
      </BalloonStyle>
    </Style>
    <Style id="bluehighlight">
      <LabelStyle>
        <scale>1.0</scale>
      </LabelStyle>
      <LineStyle>
        <color>99ff0000</color>
```

```xml
          <width>6.0</width>
        </LineStyle>
        <PolyStyle>
          <color>99ff0000</color>
          <fill xsi:type="xs:boolean" xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">true</fill>
        </PolyStyle>
        <BalloonStyle>
          <text>&lt;h3&gt;&lt;b&gt;$[name]&lt;/b&gt;&lt;/h3&gt;&lt;table&gt;&lt;tr&gt;
&lt;td width="400"&gt;$[description]&lt;/td&gt;&lt;/tr&gt;&lt;/table&gt;</text>
        </BalloonStyle>
      </Style>
      <StyleMap id="default">
        <Pair>
          <key>normal</key>
          <styleUrl>#bluenormal</styleUrl>
        </Pair>
        <Pair>
          <key>highlight</key>
          <styleUrl>#bluehighlight</styleUrl>
        </Pair>
      </StyleMap>
      <MultiGeometry>
        <Point>
          <coordinates>102.0,2.0</coordinates>
        </Point>
        <MultiGeometry>
          <Polygon>
            <outerBoundaryIs>
              <LinearRing>
                <coordinates>102.0,2.0 103.0,2.0 103.0,3.0 102.0,3.0 102.0,2.0</coordinates>
              </LinearRing>
            </outerBoundaryIs>
          </Polygon>
          <Polygon>
100.8,0.2
    <outerBoundaryIs>
    <LinearRing>
      <coordinates>100.0,0.0 101.0,0.0 101.0,1.0 100.0,1.0 100.0,0.0 100.2,0.2 100.8
,0.8 100.2,0.8 100.2,0.2</coordinates>
    </LinearRing>
    </outerBoundaryIs>
  </Polygon>
</MultiGeometry>
</Placemark>
</kml>
```

2.4.8. KML Query Response Transformer

The KML Query Response Transformer translates a query response into a KML-formatted document. The KML will contain an HTML description for each metacard that will display in the pop-up bubble in Google Earth. The HTML contains links to the full metadata view as well as the product.

2.4.8.1. Installing the KML Query Response Transformer

The `spatial-kml-transformer` feature is installed by default in the Spatial Application.

2.4.8.2. Configuring the KML Query Response Transformer

The KML Query Response Transformer has no configurable properties.

2.4.8.3. Using the KML Query Response Transformer

Using the OpenSearch Endpoint, for example, query with the format option set to the KML shortname: `kml`.

KML Query Response Transformer URL

```
http://{FQDN}:{PORT}/services/catalog/query?q=schematypesearch&format=kml
```

KML Query Response Transformer Example Output

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<kml xmlns:ns2="http://www.google.com/kml/ext/2.2" xmlns="http://www.opengis.net/kml/2.2"
xmlns:ns4="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0" xmlns:ns3=
"http://www.w3.org/2005/Atom">
  <Document id="f0884d8c-cf9b-44a1-bb5a-d3c6fb9a96b6">
    <name>Results (1)</name>
    <open xsi:type="xs:boolean" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">false</open>
    <Style id="bluenormal">
      <LabelStyle>
        <scale>0.0</scale>
      </LabelStyle>
      <LineStyle>
        <color>33ff0000</color>
        <width>3.0</width>
      </LineStyle>
      <PolyStyle>
        <color>33ff0000</color>
        <fill xsi:type="xs:boolean" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">true</fill>
      </PolyStyle>
      <BalloonStyle>
```

```xml
      <text>&lt;h3&gt;&lt;b&gt;$[name]&lt;/b&gt;&lt;/h3&gt;&lt;table&gt;&lt;tr&gt;
&lt;td width="400"&gt;$[description]&lt;/td&gt;&lt;/tr&gt;&lt;/table&gt;</text>
    </BalloonStyle>
  </Style>
  <Style id="bluehighlight">
    <LabelStyle>
      <scale>1.0</scale>
    </LabelStyle>
    <LineStyle>
      <color>99ff0000</color>
      <width>6.0</width>
    </LineStyle>
    <PolyStyle>
      <color>99ff0000</color>
      <fill xsi:type="xs:boolean" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">true</fill>
    </PolyStyle>
    <BalloonStyle>
      <text>&lt;h3&gt;&lt;b&gt;$[name]&lt;/b&gt;&lt;/h3&gt;&lt;table&gt;&lt;tr&gt;
&lt;td width="400"&gt;$[description]&lt;/td&gt;&lt;/tr&gt;&lt;/table&gt;</text>
    </BalloonStyle>
  </Style>
  <StyleMap id="default">
    <Pair>
      <key>normal</key>
      <styleUrl>#bluenormal</styleUrl>
    </Pair>
    <Pair>
      <key>highlight</key>
      <styleUrl>#bluehighlight</styleUrl>
    </Pair>
  </StyleMap>
  <Placemark id="Placemark-0103c77e66d9428d8f48fab939da528e">
    <name>MultiPolygon</name>
    <description>&lt;!DOCTYPE html&gt;
&lt;html&gt;
  &lt;head&gt;
    &lt;meta content="text/html; charset=windows-1252" http-equiv="content-type"&gt;
    &lt;style media="screen" type="text/css"&gt;
      .label {
        font-weight: bold
      }
      .linkTable {
width: 100% }
      .thumbnailDiv {
        text-align: center
} img {
        max-width: 100px;
```

```
                    max-height: 100px;
                    border-style:none
                }
        &lt;/style&gt;
    &lt;/head&gt;
    &lt;body&gt;
            &lt;div class="thumbnailDiv"&gt;&lt;a
href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f
48fab939da528e?transform=resource"&gt;&lt;img alt="Thumnail"
src="data:image/jpeg;charset=utf-8;base64, CA=="&gt;&lt;/a&gt;&lt;/div&gt;
    &lt;table&gt;
        &lt;tr&gt;
            &lt;td class="label"&gt;Source:&lt;/td&gt;
            &lt;td&gt;ddf.distribution&lt;/td&gt;
        &lt;/tr&gt;
        &lt;tr&gt;
            &lt;td class="label"&gt;Created:&lt;/td&gt;
            &lt;td&gt;Wed Oct 30 09:46:29 MDT 2013&lt;/td&gt;
        &lt;/tr&gt;
        &lt;tr&gt;
            &lt;td class="label"&gt;Effective:&lt;/td&gt;
            &lt;td&gt;2014-01-07T14:48:47-0700&lt;/td&gt;
        &lt;/tr&gt;
    &lt;/table&gt;
    &lt;table class="linkTable"&gt;
        &lt;tr&gt;
            &lt;td&gt;&lt;a
href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f
48fab939da528e?transform=html"&gt;View Details...&lt;/a&gt;&lt;/td&gt;
            &lt;td&gt;&lt;a
href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f
48fab939da528e?transform=resource"&gt;Download...&lt;/a&gt;&lt;/td&gt;
        &lt;/tr&gt;
    &lt;/table&gt;
    &lt;/body&gt;
&lt;/html&gt;
</description>
        <TimeSpan>
            <begin>2014-01-07T21:48:47</begin>
        </TimeSpan>
        <styleUrl>#default</styleUrl>
        <MultiGeometry>
            <Point>
                <coordinates>102.0,2.0</coordinates>
            </Point>
            <MultiGeometry>
                <Polygon>
                    <outerBoundaryIs>
```

```
                <LinearRing>
                    <coordinates>102.0,2.0 103.0,2.0 103.0,3.0 102.0,3.0
102.0,2.0</coordinates>
                </LinearRing>
100.8,0.2
  </outerBoundaryIs>
</Polygon>
<Polygon>
  <outerBoundaryIs>
    <LinearRing>
      <coordinates>100.0,0.0 101.0,0.0 101.0,1.0 100.0,1.0 100.0,0.0 100.2,0.2
        100.8,0.8 100.2,0.8 100.2,0.2</coordinates>
            </LinearRing>
          </outerBoundaryIs>
        </Polygon>
      </MultiGeometry>
    </MultiGeometry>
  </Placemark>
  </Document>
</kml>
```

2.4.9. KML Style Mapper

The KML Style Mapper provides the ability for the `KmlTransformer` to map a KML Style URL to a metacard based on that metacard's attributes. For example, if a user wanted all JPEGs to be blue, the KML Style Mapper provides the ability to do so. This would also allow an administrator to configure metacards from each source to be different colors.

The configured style URLs are expected to be HTTP URLs. For more information on style URL's, refer to the KML Reference ☑.

The KML Style Mapper supports all basic and extended metacard attributes. When a style mapping is configured, the resulting transformed KML contain a `<styleUrl>` tag pointing to that style, rather than the default KML style supplied by the `KmlTransformer`.

2.4.9.1. Installing the KML Style Mapper

The KML Style Mapper is installed by default with a standard installation in the Spatial Application in the `spatial-kml-transformer` feature.

2.4.9.2. Configuring the KML Style Mapper

The properties below describe how to configure a style mapping. The configuration name is `Spatial KML Style Map Entry`.

See KML Style Mapper configurations for all possible configurations.

KML Style Mapper Example Values

```
<xmlns="http://www.opengis.net/kml/2.2"
  xmlns:ns4="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0"
xmlns:ns3="http://www.w3.org/2005/Atom">
  <Placemark id="Placemark-0103c77e66d9428d8f48fab939da528e">
    <name>MultiPolygon</name>
    <description>&lt;!DOCTYPE html&gt;
&lt;html&gt;
  &lt;head&gt;
    &lt;meta content="text/html; charset=windows-1252" http-equiv="content-type"&gt;
    &lt;style media="screen" type="text/css"&gt;
      .label {
        font-weight: bold
      }
      .linkTable {
width: 100% }
      .thumbnailDiv {
        text-align: center
} img {
        max-width: 100px;
        max-height: 100px;
        border-style:none
      }
    &lt;/style&gt;
  &lt;/head&gt;
  &lt;body&gt;
      &lt;div class="thumbnailDiv"&gt;&lt;a
href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f48
fab939da528e?transform=resource"&gt;&lt;img alt="Thumnail"
src="data:image/jpeg;charset=utf-8;base64, CA=="&gt;&lt;/a&gt;&lt;/div&gt;
    &lt;table&gt;
      &lt;tr&gt;
        &lt;td class="label"&gt;Source:&lt;/td&gt;
        &lt;td&gt;ddf.distribution&lt;/td&gt;
      &lt;/tr&gt;
      &lt;tr&gt;
        &lt;td class="label"&gt;Created:&lt;/td&gt;
        &lt;td&gt;Wed Oct 30 09:46:29 MDT 2013&lt;/td&gt;
      &lt;/tr&gt;
      &lt;tr&gt;
        &lt;td class="label"&gt;Effective:&lt;/td&gt;
        &lt;td&gt;2014-01-07T14:58:16-0700&lt;/td&gt;
      &lt;/tr&gt;
    &lt;/table&gt;
    &lt;table class="linkTable"&gt;
```

```xml
        &lt;tr&gt;
          &lt;td&gt;&lt;a
href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f48
fab939da528e?transform=html"&gt;View Details...&lt;/a&gt;&lt;/td&gt;
          &lt;td&gt;&lt;a
href="http://{FQDN}:{PORT}/services/catalog/sources/ddf.distribution/0103c77e66d9428d8f48
fab939da528e?transform=resource"&gt;Download...&lt;/a&gt;&lt;/td&gt;
        &lt;/tr&gt;
      &lt;/table&gt;
    &lt;/body&gt;
&lt;/html&gt;
</description>
    <TimeSpan>
      <begin>2014-01-07T21:58:16</begin>
    </TimeSpan>
 <styleUrl>http://example.com/kml/style#sampleStyle</styleUrl>
    <MultiGeometry>
      <Point>
        <coordinates>102.0,2.0</coordinates>
      </Point>
      <MultiGeometry>
        <Polygon>
          <outerBoundaryIs>
            <LinearRing>
              <coordinates>102.0,2.0 103.0,2.0 103.0,3.0 102.0,3.0
102.0,2.0</coordinates>
            </LinearRing>
          </outerBoundaryIs>
        </Polygon>
        <Polygon>
100.8,0.2
<outerBoundaryIs>
  <LinearRing>
    <coordinates>100.0,0.0 101.0,0.0 101.0,1.0 100.0,1.0 100.0,0.0 100.2,0.2
      100.8,0.8 100.2,0.8 100.2,0.2</coordinates>
  </LinearRing>
    </outerBoundaryIs>
  </Polygon>
</MultiGeometry>
</MultiGeometry>
</Placemark>
</kml>
```

2.4.10. Metadata Metacard Transformer

The Metadata Metacard Transformer returns the `Metacard.METADATA` attribute when given a metacard. The MIME Type returned is `text/xml`.

2.4.10.1. Installing the Metadata Metacard Transformer

The Metadata Metacard Transformer is installed by default in a standard installation with the Catalog application.

2.4.10.2. Configuring the Metadata Metacard Transformer

The Metadata Metacard Transformer has no configurable properties.

2.4.10.3. Using the Metadata Metacard Transformer

The Metadata Metacard Transformer can be used programmatically by requesting a metacard transformer with the id `metadata`. It can also be used within the REST Endpoint by providing the transform option as `metadata`.

Example REST GET method with the Metadata Metacard Transformer

```
http://{FQDN}:{PORT}/services/catalog/0123456789abcdef0123456789abcdef?transform=metadata
```

2.4.11. PDF Input Transformer

The PDF Input Transformer is responsible for translating a PDF document into a Catalog Metacard.

Table 8. PDF Input Transformer Usage

Schema	Mime-types
N/A	`application/pdf`

2.4.11.1. Installing the PDF Input Transformer

The PDF Transformer is installed by default with a standard installation in the Catalog application.

2.4.11.2. Configuring the PDF Input Transformer

To configure the PDF Input Transformer:

1. Navigate to the **Catalog** application.
2. Select the **Configuration** tab.
3. Select the **PDF Input Transformer**.

These configurations are available for the PDF Input Transformer:

See PDF Input Transformer configurations for all possible configurations.

2.4.12. PPTX Input Transformer

The PPTX Input Transformer translates Microsoft PowerPoint (OOXML only) documents into Catalog Metacards, using Apache Tika for basic metadata and Apache POI for thumbnail creation. The PPTX Input Transformer ingests PPTX documents into the DDF Content Repository and the Metadata Catalog, and adds a thumbnail of the first page in the PPTX document.

The PPTX Input Transformer will take precedence over the Tika Input Transformer for PPTX documents.

Table 9. PPTX Input Transformer Usage

Schema	Mime-types
N/A	`application/vnd.openxmlformats-officedocument.presentationml.presentation`

2.4.12.1. Installing the PPTX Input Transformer

This transformer is installed by default with a standard installation in the Catalog application.

2.4.12.2. Configuring the PPTX Input Transformer

The PPTX Input Transformer has no configurable properties. '"

2.4.13. Query Response Transformer Consumer

The Query Response Transformer Consumer is responsible for translating a query response into a Catalog Metacard.

2.4.13.1. Installing the Query Response Transformer Consumer

The Query Response Transformer Consumer is installed by default with a standard installation in the Catalog application.

2.4.13.2. Configuring the Query Response Transformer Consumer

The Query Response Transformer Consumer has no configurable properties.

2.4.14. Registry Transformer

The Registry Transformer creates Registry metacards from `ebrim` messages. It also returns the `ebrim` message from the metacard metadata.

2.4.14.1. Installing the Registry Transformer

The Registry Transformer is installed with the Registry application.

1. Install Registry application.

2.4.14.2. Configuring the Registry Transformer

The Registry Transformer has no configurable properties.

2.4.15. Resource Metacard Transformer

The Resource Metacard Transformer retrieves a resource associated with a metacard.

2.4.15.1. Installing the Resource Metacard Transformer

The Resource Metacard Transformer is installed by default in a standard installation with the Catalog application as the feature `catalog-transformer-resource`.

2.4.15.2. Configuring the Resource Metacard Transformer

The Resource Metacard Transformer has no configurable properties.

2.4.15.3. Using the Resource Metacard Transformer

Endpoints or other components can retrieve an instance of the Resource Metacard Transformer using its `id` resource.

Sample Resource Metacard Transformer Blueprint Reference Snippet

```
<reference id="metacardTransformer" interface="ddf.catalog.transform.MetacardTransformer"
filter="(id=resource)"/>
```

2.4.16. Thumbnail Metacard Transformer

The Thumbnail Metacard Transformer retrieves the thumbnail bytes of a Metacard by returning the `Metacard.THUMBNAIL` attribute value.

2.4.16.1. Installing the Thumbnail Metacard Transformer

This transformer is installed by default with a standard installation in the Catalog application.

2.4.16.2. Configuring the Thumbnail Metacard Transformer

The Thumbnail Metacard Transformer has no configurable properties.

2.4.16.3. Using the Thumbnail Metacard Transformer

Endpoints or other components can retrieve an instance of the Thumbnail Metacard Transformer using its id `thumbnail`.

Sample Blueprint Reference Snippet

```
<reference id="metacardTransformer" interface="ddf.catalog.transform.MetacardTransformer"
filter="(id=thumbnail)"/>
```

The Thumbnail Metacard Transformer returns a `BinaryContent` object of the `Metacard.THUMBNAIL` bytes and a MIME Type of `image/jpeg`.

2.4.17. Tika Input Transformer

The Tika Input Transformer is the default input transformer responsible for translating Microsoft Word, Microsoft Excel, Microsoft PowerPoint, OpenOffice Writer, and PDF documents into Catalog records. This input transformer utilizes Apache Tika to provide basic support for these mime types. The metadata common to all these document types, e.g., creation date, author, last modified date, etc., is extracted and used to create the catalog record. The Tika Input Transformer's main purpose is to ingest these types of content into the Metadata Catalog.

The Tika input transformer is most basic input transformer and the last to be invoked. This allows any registered input transformers that are more specific to a document type to be invoked instead of this rudimentary input transformer.

Table 10. Tika Input Transformer Usage

Schema	Mime-types
N/A	This basic transformer can ingest many file types. See All Formats Supported.

2.4.17.1. Installing the Tika Input Transformer

This transformer is installed by default with a standard installation in the Catalog.

2.4.17.2. Configuring the Tika Input Transformer

The properties below describe how to configure the Tika input transformer.

See Tika Input Transformer configurations for all possible configurations.

2.4.18. Video Input Transformer

The video input transformer Creates Catalog metacards from certain video file types. Currently, it is

handles MPEG-2 transport streams as well as MPEG-4, AVI, MOV, and WMV videos. This input transformer uses Apache Tika to extract basic metadata from the video files and applies more sophisticated methods to extract more meaningful metadata from these types of video.

Table 11. Video Input Transformer Usage

Schema	Mime-types
N/A	- video/avi - video/msvideo - video/vnd.avi - video/x-msvideo - video/mp4 - video/MP2T - video/mpeg - video/quicktime - video/wmv - video/x-ms-wmv

2.4.18.1. Installing the Video Input Transformer

This transformer is installed by default with a standard installation in the Catalog application.

2.4.18.1.1. Configuring the Video Input Transformer

The Video Input Transformer has no configurable properties.

2.4.19. XML Input Transformer

The XML Input Transformer is responsible for translating an XML document into a Catalog Metacard.

Table 12. XML Input Transformer Usage

Schema	Mime-types
urn:catalog:metacard	`text/xml`

2.4.19.1. Installing the XML Input Transformer

The XML Input Transformer is installed by default with a standard installation in the Catalog application.

2.4.19.2. Configuring the XML Input Transformer

The XML Input Transformer has no configurable properties.

2.4.20. XML Metacard Transformer

The XML metacard transformer is responsible for translating a metacard into an XML-formatted document. The metacard element that is generated is an extension of `gml:AbstractFeatureType`, which makes the output of this transformer GML 3.1.1 compatible.

2.4.20.1. Installing the XML Metacard Transformer

This transformer comes installed by default with a standard installation in the Catalog application.

To install or uninstall manually, use the `catalog-transformer-xml` feature.

2.4.20.2. Configuring the XML Metacard Transformer

The XML Metacard Transformer has no configurable properties.

2.4.20.3. Using the XML Metacard Transformer

Using the REST Endpoint for example, request a metacard with the transform option set to the XML shortname.

XML Metacard Transformer URL

```
https://{FQDN}:{PORT}/services/catalog/ac0c6917d5ee45bfb3c2bf8cd2ebaa67?transform=xml
```

Table 13. Metacard to XML Mappings

Metacard Variables	XML Element
id	metacard/@gml:id
metacardType	metacard/type
sourceId	metacard/source
all other attributes	metacard/<AttributeType>[name='<AttributeName>']/value For instance, the value for the metacard attribute named "title" would be found at metacard/string[@name='title']/value

XML Adapted Attributes (AttributeTypes)

- boolean
- base64Binary
- dateTime
- double
- float
- geometry
- int
- long

- `object`
- `short`
- `string`
- `stringxml`

2.4.21. XML Query Response Transformer

The XML Query Response Transformer is responsible for translating a query response into an XML-formatted document. The metacard element generated is an extension of `gml:AbstractFeatureCollectionType`, which makes the output of this transformer GML 3.1.1 compatible.

2.4.21.1. Installing the XML Query Response Transformer

This transformer is installed by default with a standard installation in the Catalog application. To uninstall, uninstall the `catalog-transformer-xml` feature.

2.4.21.2. Configuring the XML Query Response Transformer

To configure the XML Query Response Transformer:

1. Navigate to the **Admin Console**.
2. Select the **Catalog** application.
3. Select the **Configuration** tab.
4. Select the XML Query Response Transformer.

See XML Query Response Transformer configurations for all possible configurations.

2.4.21.3. Using the XML Query Response Transformer

Using the OpenSearch Endpoint, for example, query with the format option set to the XML shortname `xml`.

XML Query Response Transformer Query Example

```
http://{FQDN}:{PORT}/services/catalog/query?q=input?format=xml
```

XML Query Response Transformer Example Output

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns3:metacards xmlns:ns1="http://www.opengis.net/gml" xmlns:ns2=
"http://www.w3.org/1999/xlink" xmlns:ns3="urn:catalog:metacard" xmlns:ns4=
"http://www.w3.org/2001/SMIL20/" xmlns:ns5="http://www.w3.org/2001/SMIL20/Language">
    <ns3:metacard ns1:id="000ba4dd7d974e258845a84966d766eb">
        <ns3:type>ddf.metacard</ns3:type>
        <ns3:source>southwestCatalog1</ns3:source>
        <ns3:dateTime name="created">
          <ns3:value>2013-04-10T15:30:05.702-07:00</ns3:value>
        </ns3:dateTime>
        <ns3:string name="title">
            <ns3:value>Input 1</ns3:value>
        </ns3:string>
    </ns3:metacard>
    <ns3:metacard ns1:id="00c0eb4ba9b74f8b988ef7060e18a6a7">
        <ns3:type>ddf.metacard</ns3:type>
        <ns3:source>southwestCatalog1</ns3:source>
        <ns3:dateTime name="created">
          <ns3:value>2013-04-10T15:30:05.702-07:00</ns3:value>
        </ns3:dateTime>
        <ns3:string name="title">
            <ns3:value>Input 2</ns3:value>
        </ns3:string>
    </ns3:metacard>
</ns3:metacards>
```

2.5. Mime Type Mapper

The MimeTypeMapper is the entry point in DDF for resolving file extensions to mime types, and vice versa.

`MimeTypeMappers` are used by the `ResourceReader` to determine the file extension for a given mime type in aid of retrieving a product. `MimeTypeMappers` are also used by the `FileSystemProvider` in the Catalog Framework to read a file from the content file repository.

The `MimeTypeMapper` maintains a list of all of the `MimeTypeResolvers` in DDF.

The `MimeTypeMapper` accesses each `MimeTypeResolver` according to its priority until the provided file extension is successfully mapped to its corresponding mime type. If no mapping is found for the file extension, `null` is returned for the mime type. Similarly, the `MimeTypeMapper` accesses each `MimeTypeResolver` according to its priority until the provided mime type is successfully mapped to its corresponding file extension. If no mapping is found for the mime type, `null` is returned for the file

extension.

For files with no file extension, the MimeTypeMapper will attempt to determine the mime type from the contents of the file. If it is unsuccessful, the file will be ingested as a binary file.

DDF Mime Type Mapper

Core implementation of the DDF Mime API.

2.5.1. DDF Mime Type Mapper

The DDF Mime Type Mapper is the core implementation of the DDF Mime API. It provides access to all `MimeTypeResolvers` within DDF, which provide mapping of mime types to file extensions and file extensions to mime types.

2.5.1.1. Installing the DDF Mime Type Mapper

The DDF Mime Type Mapper is installed by default with a standard installation in the Platform application.

2.5.1.2. Configuring DDF Mime Type Mapper

The DDF Mime Type Mapper has no configurable properties.

2.6. Mime Type Resolver

A `MimeTypeResolver` is a DDF service that can map a file extension to its corresponding mime type and, conversely, can map a mime type to its file extension.

`MimeTypeResolvers` are assigned a priority (0-100, with the higher the number indicating the higher priority). This priority is used to sort all of the `MimeTypeResolvers` in the order they should be checked to map a file extension to a mime type (or vice versa). This priority also allows custom `MimeTypeResolvers` to be invoked before default `MimeTypeResolvers` by setting custom resolver's priority higher than the default.

`MimeTypeResolvers` are not typically invoked directly. Rather, the `MimeTypeMapper` maintains a list of `MimeTypeResolvers` (sorted by their priority) that it invokes to resolve a mime type to its file extension (or to resolve a file extension to its mime type).

Custom Mime Type Resolver

The Custom Mime Type Resolver is a `MimeTypeResolver` that defines the custom mime types that DDF will support.

Tika Mime Type Resolver

Provides support for resolving over 1300 mime types.

2.6.1. Custom Mime Type Resolver

These are mime types not supported by the default `TikaMimeTypeResolver`.

Table 14. Custom Mime Type Resolver Default Supported Mime Types

File Extension	Mime Type
`nitf`	`image/nitf`
`ntf`	`image/nitf`
`json`	`json=application/json;id=geojson`

As a `MimeTypeResolver`, the Custom Mime Type Resolver will provide methods to map the file extension to the corresponding mime type, and vice versa.

2.6.1.1. Installing the Custom Mime Type Resolver

One Custom Mime Type Resolver is configured and installed for the `image/nitf` mime type. This custom resolver is bundled in the `mime-core-app` application and is part of the `mime-core` feature.

Additional Custom Mime Type Resolvers can be added for other custom mime types.

2.6.1.1.1. Configuring the Custom Mime Type Resolver

The configurable properties for the Custom Mime Type Resolver are accessed from the **MIME Custom Types** configuration in the Admin Console.

- Navigate to the Admin Console.
- Select the **Platform** application.
- Select **Configuration**.
- Select **MIME Custom Types**.

Managed Service Factory PID

- `Ddf_Custom_Mime_Type_Resolver`

See Custom Mime Type Resolver configurations for all possible configurations.

2.6.2. Tika Mime Type Resolver

The `TikaMimeTypeResolver` is a `MimeTypeResolver` that is implemented using the Apache Tika open source product.

Using the Apache Tika content analysis toolkit, the `TikaMimeTypeResolver` provides support for resolving over 1300 mime types, but not all mime types yield the same quality metadata.

The `TikaMimeTypeResolver` is assigned a default priority of `-1` to insure that it is always invoked last by the `MimeTypeMapper`. This insures that any custom `MimeTypeResolvers` that may be installed will be invoked before the `TikaMimeTypeResolver`.

The `TikaMimeTypeResolver` provides the bulk of the default mime type support for DDF.

2.6.2.1. Installing the Tika Mime Type Resolver

The `TikaMimeTypeResolver` is bundled as the `mime-tika-resolver` feature in the `mime-tika-app` application.

This feature is installed by default.

2.6.2.1.1. Configuring the Tika Mime Type Resolver

The Tika Mime Type Resolver has no configurable properties.

3. Catalog Plugins
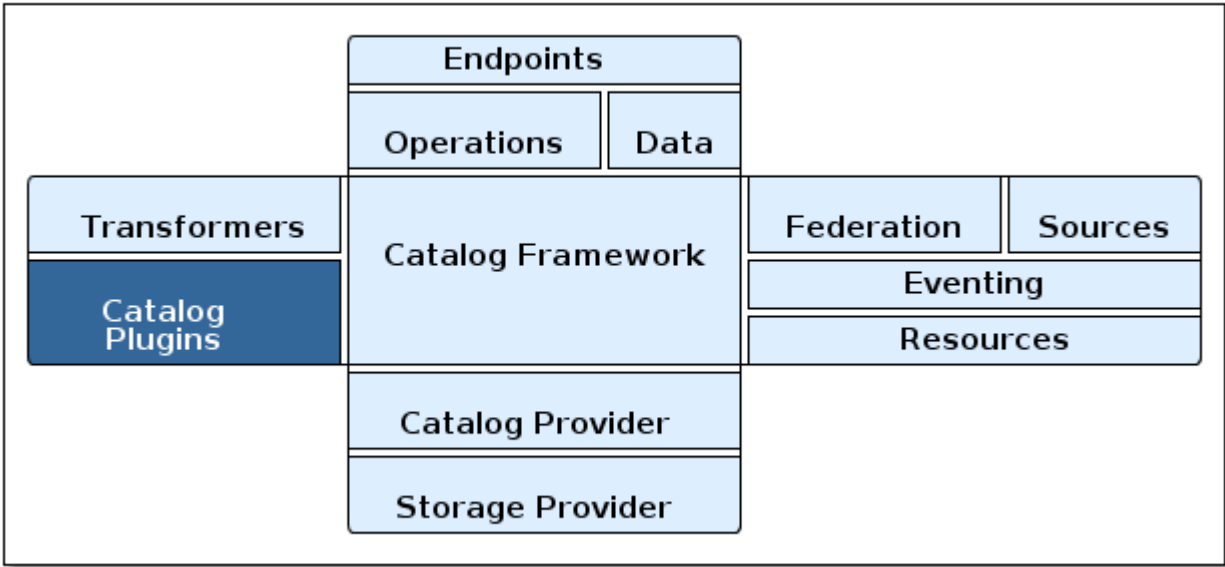
Catalog Architecture: Catalog Plugins

Plugins are additional tools to use to add additional business logic at certain points, depending on the type of plugin.

The Catalog Framework calls Catalog Plugins to process requests and responses as they enter and leave the Framework.

3.1. Types of Plugins

Plugins can be designed to run before or after certain processes. They are often used for validation, optimization, or logging. Many plugins are designed to be called at more than one time. See Catalog Plugin Compatibility.

Pre-Authorization Plugins

Perform any changes needed before security rules are applied.

Policy Plugins

Allows or denies access to the Catalog operation or response.

Access Plugins

Used to build policy information for requests.

Pre-Ingest Plugins

Perform any changes to a metacard prior to ingest.

Post-Ingest Plugins

Perform actions after ingest is completed.

Post-Process Plugins

Performs additional processing after ingest.

Pre-Query Plugins

Perform any changes to a query before execution.

Pre-Federated-Query Plugins

Perform any changes to a federated query before execution.

Post-Query Plugins

Perform any changes to a response after query completes.

Post-Federated-Query Plugins

Perform any changes to a response after federated query completes.

Pre-Resource Plugins

Perform any changes to a request associated with a metacard prior to download.

Post-Resource Plugins

Perform any changes to a resource after download.

Pre-Create Storage Plugins

Perform any changes before creating a resource.

Post-Create Storage Plugins

Perform any changes after creating a resource.

Pre-Update Storage Plugins

Perform any changes before updating a resource.

Post-Update Storage Plugins

Perform any changes after updating a resource.

Pre-Subscription Plugins

Perform any changes before creating a subscription.

Pre-Delivery Plugins

Perform any changes before delivering a subscribed event.

Plugins are called in a specific order during different operations. Custom Plugins can be added to the chain for special use cases.

Query Request Plugin Call Order
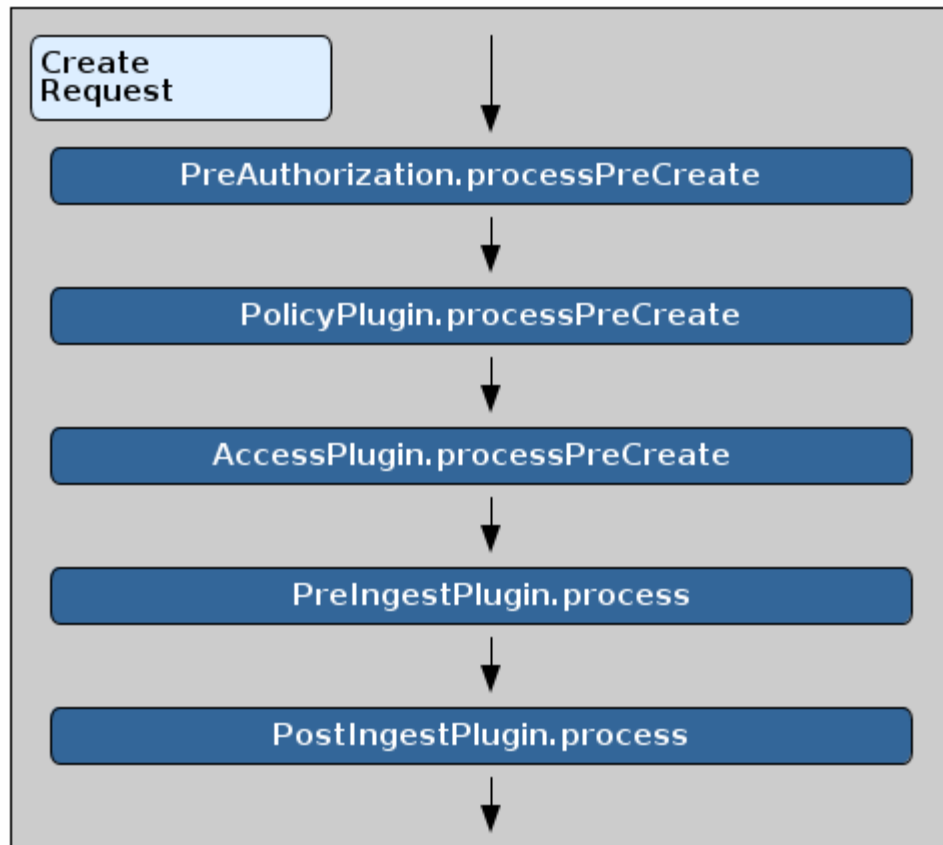
Create Request Plugin Call Order
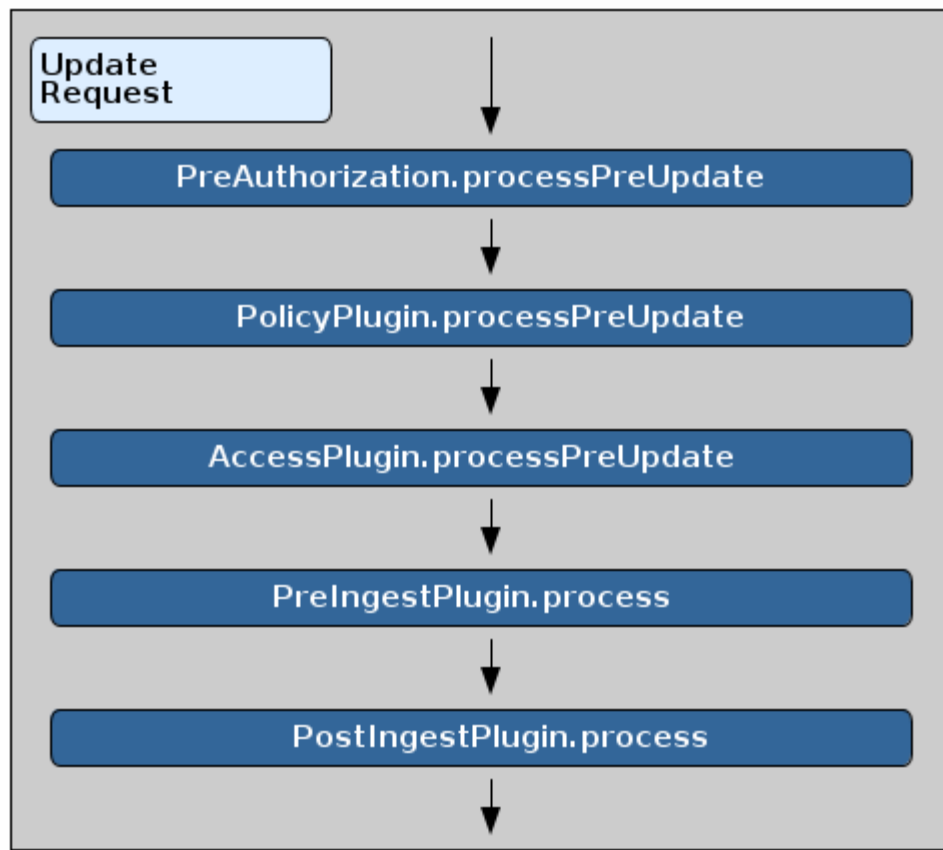
Update Request Plugin Call Order

Delete Request Plugin Call Order

Resource Request Plugin Call Order
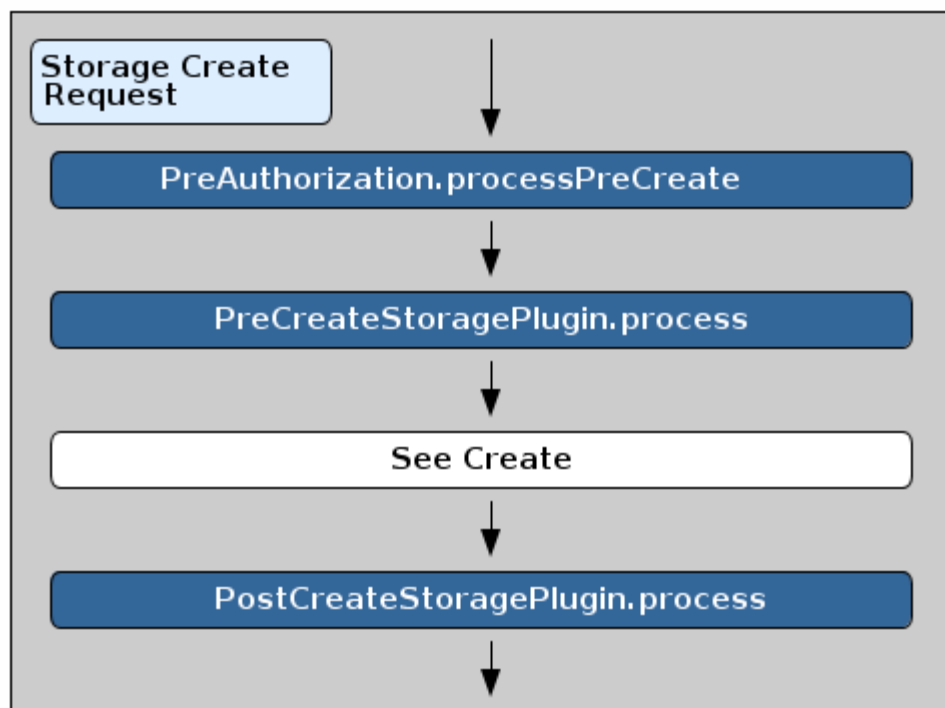
Storage Create Request Plugin Call Order
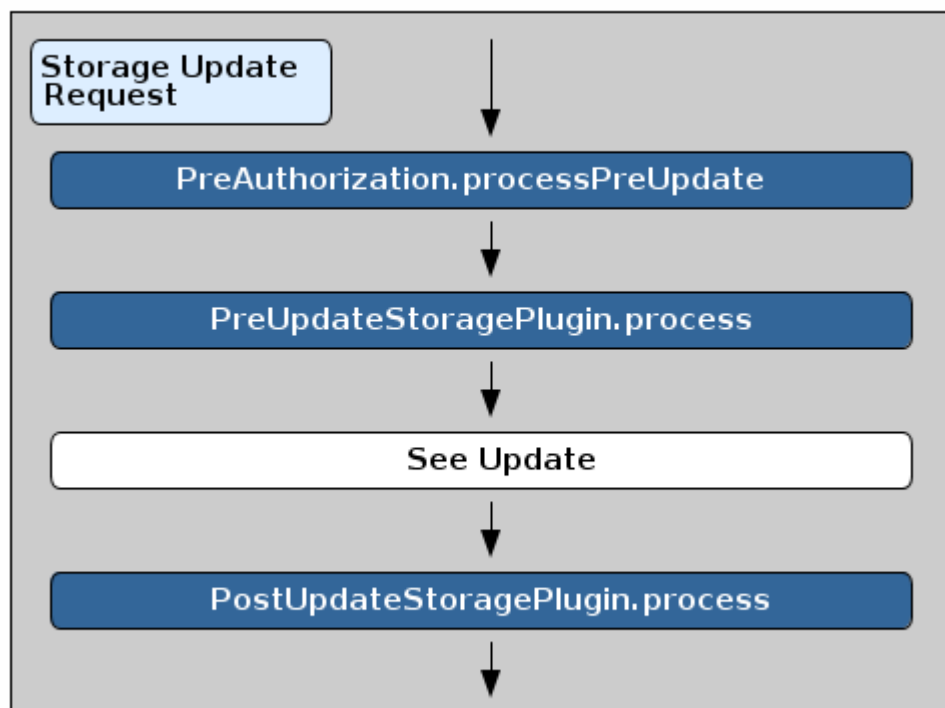
Storage Update Request Plugin Call Order

Table 15. Catalog Plugin Compatibility

Plugin	Pre-Authorization Plugins	Policy Plugins	Access Plugins	Pre-Ingest Plugins	Post-Ingest Plugins	Pre-Query Plugins	Post-Query Plugins	Post-Process Plugins
Catalog Backup Plugin					x			
Catalog Metrics Plugin					x	x	x	
Catalog Policy Plugin		x						
Client Info Plugin	x							
Content URI Access Plugin			x					
Event Processor					x			
Expiration Date Pre-Ingest Plugin				x				
Filter Plugin			x					
GeoCoder Plugin				x				
Historian Policy Plugin		x						
Identification Plugin				x	x			
JPEG2000 Thumbnail Converter							x	

Plugin	Pre-Authoriz ation Plugins	Policy Plugins	Access Plugins	Pre-Ingest Plugins	Post-Ingest Plugins	Pre-Query Plugins	Post-Query Plugins	Post-Process Plugins
Metacard Attribute Security Policy Plugin		x						
Metacard Backup File Storage Provider					x			
Metacard Backup S3 Storage Provider					x			
Metacard Groomer				x				
Metacard Resource Size Plugin							x	
Metacard Validity Filter Plugin		x						
Metacard Validity Marker				x				
Metacard Ingest Network Plugin	x							
Operation Plugin			x					
Point of Contact Policy Plugin		x						

Plugin	Pre-Authorization Plugins	Policy Plugins	Access Plugins	Pre-Ingest Plugins	Post-Ingest Plugins	Pre-Query Plugins	Post-Query Plugins	Post-Process Plugins
Processing Post-Ingest Plugin					x			
Registry Policy Plugin		x						
Resource URI Policy Plugin		x						
Security Audit Plugin			x					
Security Logging Plugin				x	x	x	x	
Security Plugin			x					
Source Metrics Plugin					x	x	x	
Workspace Access Plugin			x					
Workspace Pre-Ingest Plugin				x				
Workspace Sharing Policy Plugin		x						
XML Attribute Security Policy Plugin		x						

Table 16. Catalog Plugin Compatibility, Cont.

Plugin	Pre-Federated-Query Plugins	Post-Federated-Query Plugins	Pre-Resource Plugins	Post-Resource Plugins	Pre-Create Storage Plugins	Post-Create Storage Plugins	Pre-Update Storage Plugins	Post-Update Storage Plugins	Pre-Subscription Plugins	Pre-Delivery Plugins
Catalog Metrics Plugin				x						
Checksum Plugin					x		x			
Resource Usage Plugin			x	x						
Security Logging Plugin	x!	x	x	x	x	x	x	x		
Source Metrics Plugin				x						
Video Thumbnail Plugin						x		x		

3.1.1. Pre-Authorization Plugins

Pre-delivery plugins are invoked before any security rules are applied. This is an opportunity to take any action before authorization, including but not limited to:

- logging.
- adding network-specific information.
- adding user-identifying information.

3.1.1.1. Available Pre-Authorization Plugins

Client Info Plugin

Injects request-specific network information into a request.

Metacard Ingest Network Plugin

Adds attributes for network info from ingest request.

3.1.2. Policy Plugins

Policy plugins are invoked to set up the policy for a request/response. This provides an opportunity to attach custom requirements on operations or individual metacards. All the 'requirements' from each Policy plugin will be combined into a single policy that will be included in the request/response. Access plugins will be used to act on this combined policy.

3.1.2.1. Available Policy Plugins

Catalog Policy Plugin

Configures user attributes required for catalog operations.

Historian Policy Plugin

Protects metacard history from being edited by users without the history role.

Metacard Attribute Security Policy Plugin

Collects attributes into a security field for the metacard.

Metacard Validity Filter Plugin

Determines whether to filter metacards with validation errors or warnings.

Point of Contact Policy Plugin

Adds a policy if Point of Contact is updated.

Registry Policy Plugin

Defines user access polices for registry operations.

Resource URI Policy Plugin

Configures required user attributes for setting or altering a resource URI.

Workspace Sharing Policy Plugin

Collects attributes for a workspace to identify the appropriate policy to allow sharing.

XML Attribute Security Policy Plugin

Finds security attributes contained in a metacard's metadata.

3.1.3. Access Plugins

Access plugins are invoked directly after the Policy plugins have been successfully executed. This is an opportunity to either stop processing or modify the request/response based on policy information.

3.1.3.1. Available Access Plugins

Content URI Access Plugin

Prevents a Metacard's resource URI from being overridden by an incoming UpdateRequest.

Filter Plugin

Performs filtering on query responses as they pass through the framework.

Operation Plugin

Validates a user or subject's security attributes.

Security Audit Plugin

Audits specific metacard attributes.

Security Plugin

Identifies the subject for an operation.

Workspace Access Plugin

Prevents non-owner users from changing workspace permissions.

3.1.4. Pre-Ingest Plugins
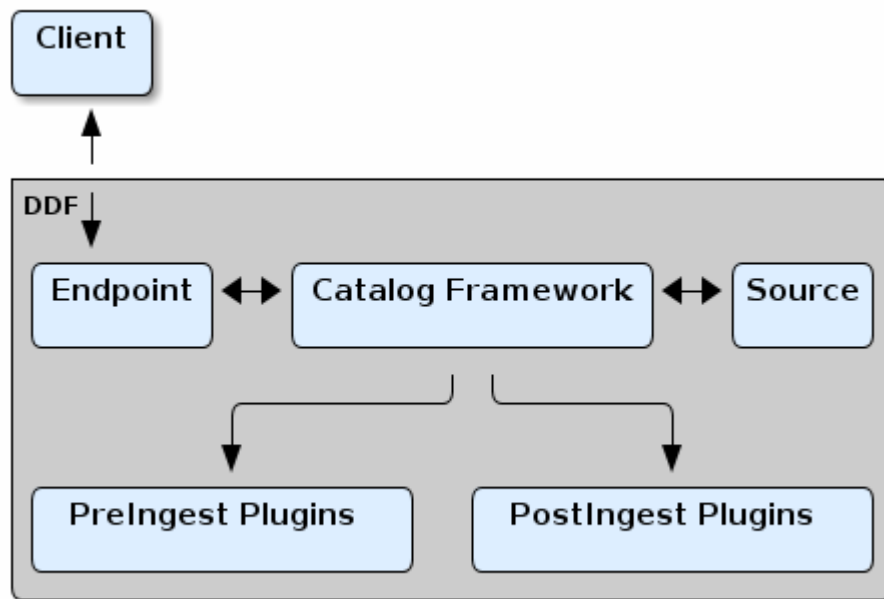
Ingest Plugin Flow

Pre-ingest plugins are invoked before an ingest operation is sent to the catalog. They are not run on a query. This is an opportunity to take any action on the ingest request, including but not limited to:

- validation.
- logging.
- auditing.
- optimization.

- security filtering.

3.1.4.1. Available Pre-Ingest Plugins

Expiration Date Pre-Ingest Plugin

Adds or updates expiration dates for the resource.

GeoCoder Plugin

Populates the `Location.COUNTRY_CODE` attribute if the Metacard has an associated location.

Identification Plugin

Manages IDs on registry metacards.

Metacard Groomer

Modifies metacards when created or updated.

Metacard Validity Marker

Modifies metacards when created or ingested according to metacard validator services.

Security Logging Plugin

Logs operations to the security log.

Workspace Pre-Ingest Plugin

Verifies that a workspace has an associated email to enable sharing.

3.1.5. Post-Ingest Plugins
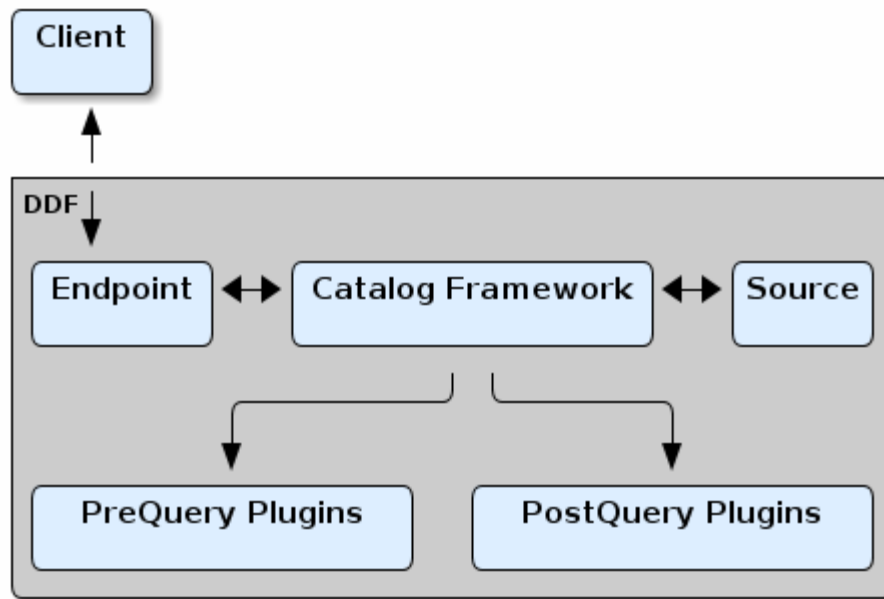
Query Plugin Flow

Post-ingest plugins are invoked after data has been created, updated, or deleted in a Catalog Provider.

3.1.5.1. Available Post-Ingest Plugins

Catalog Backup Plugin

Enables backup of the catalog and its metacards.

Catalog Metrics Plugin

Captures metrics on catalog operations.

Event Processor

Creates, updates, and deletes subscriptions.

Identification Plugin

Manages IDs on registry metacards.

Metacard Backup File Storage Provider

Stores backed-up metacards.

Metacard Backup S3 Storage Provider

Stores backed-up metacards in a specified S3 bucket and key.

Processing Post-Ingest Plugin

Submits catalog Create, Update, or Delete requests to the Processing Framework.

Security Logging Plugin

Logs operations to the security log.

Source Metrics Plugin

Captures metrics on catalog operations.

3.1.6. Post-Process Plugins

NOTE	This code is experimental. While this interface is functional and tested, it may change or be removed in a future version of the library.

Post-Process Plugins are invoked after a metacard has been created, updated, or deleted and committed to the Catalog. They are the last plugins to run and are triggered by a **Post-Ingest Plugin**. Post-Process plugins are well-suited for asynchronous tasks. See the Asynchronous Processing Framework for more information about how **Post-Process Plugins** are used.

3.1.6.1. Available Post-Process Plugins

None.

3.1.7. Pre-Query Plugins

Pre-query plugins are invoked before a query operation is sent to any of the Sources. This is an opportunity to take any action on the query, including but not limited to:

- validation.
- logging.
- auditing.
- optimization.
- security filtering.

3.1.7.1. Available Pre-Query Plugins

Catalog Metrics Plugin

Captures metrics on catalog operations.

Security Logging Plugin

Logs operations to the security log.

Source Metrics Plugin

Captures metrics on catalog operations.

3.1.8. Pre-Federated-Query Plugins

Pre-federated-query plugins are invoked before a federated query operation is sent to any of the Sources. This is an opportunity to take any action on the query, including but not limited to:

- validation.
- logging.
- auditing.
- optimization.
- security filtering.

3.1.8.1. Available Pre-Federated-Query Plugins

Security Logging Plugin

    Logs operations to the security log.

Tags Filter Plugin

    Updates queries without filters.

3.1.9. Post-Query Plugins

Post-query plugins are invoked after a query has been executed successfully, but before the response is returned to the endpoint. This is an opportunity to take any action on the query response, including but not limited to:

- logging.
- auditing.
- security filtering/redaction.
- deduplication.

3.1.9.1. Available Post-Query Plugins

Catalog Metrics Plugin

    Captures metrics on catalog operations.

JPEG2000 Thumbnail Converter

    Creates thumbnails for jpeg2000 images.

Metacard Resource Size Plugin

    Updates the resource size attribute of a metacard.

Security Logging Plugin

    Logs operations to the security log.

Source Metrics Plugin

Captures metrics on catalog operations.

3.1.10. Post-Federated-Query Plugins

Post-federated-query plugins are invoked after a federated query has been executed successfully, but before the response is returned to the endpoint. This is an opportunity to take any action on the query response, including but not limited to:

- logging.

- auditing.

- security filtering/redaction.

- deduplication.

3.1.10.1. Available Post-Federated-Query Plugins

Security Logging Plugin

Logs operations to the security log.

3.1.11. Pre-Resource Plugins

Pre-Resource plugins are invoked before a request to retrieve a resource is sent to a Source. This is an opportunity to take any action on the request, including but not limited to:

- validation.

- logging.

- auditing.

- optimization.

- security filtering.

3.1.11.1. Available Pre-Resource Plugins

Resource Usage Plugin

Monitors and limits system data usage.

Security Logging Plugin

Logs operations to the security log.

3.1.12. Post-Resource Plugins

Post-resource plugins are invoked after a resource has been retrieved, but before it is returned to the endpoint. This is an opportunity to take any action on the response, including but not limited to:

- logging.

- auditing.

- security filtering/redaction.

3.1.12.1. Available Post-Resource Plugins

Catalog Metrics Plugin

Captures metrics on catalog operations.

Resource Usage Plugin

Monitors and limits system data usage.

Security Logging Plugin

Logs operations to the security log.

Source Metrics Plugin

Captures metrics on catalog operations.

3.1.13. Pre-Create Storage Plugins

Pre-Create storage plugins are invoked immediately before an item is created in the content repository.

3.1.13.1. Available Pre-Create Storage Plugins

Checksum Plugin

Creates a unique checksum for ingested resources.

Security Logging Plugin

Logs operations to the security log.

3.1.14. Post-Create Storage Plugins

Post-Create storage plugins are invoked immediately after an item is created in the content repository.

3.1.14.1. Available Post-Create Storage Plugins

Security Logging Plugin

Logs operations to the security log.

Video Thumbnail Plugin

Generates thumbnails for video files.

3.1.15. Pre-Update Storage Plugins

Pre-Update storage plugins are invoked immediately before an item is updated in the content

repository.

3.1.15.1. Available Pre-Update Storage Plugins

Checksum Plugin

 Creates a unique checksum for ingested resources.

Security Logging Plugin

 Logs operations to the security log.

3.1.16. Post-Update Storage Plugins

Post-Update storage plugins are invoked immediately after an item is updated in the content repository.

3.1.16.1. Available Post-Update Storage Plugins

Security Logging Plugin

 Logs operations to the security log.

Video Thumbnail Plugin

 Generates thumbnails for video files.

3.1.17. Pre-Subscription Plugins

Pre-subscription plugins are invoked before a Subscription is activated by an Event Processor. This is an opportunity to take any action on the Subscription, including but not limited to:

- validation.
- logging.
- auditing.
- optimization.
- security filtering.

3.1.17.1. Available Pre-Subscription Plugins

None.

3.1.18. Pre-Delivery Plugins

Pre-delivery plugins are invoked before a Delivery Method is invoked on a Subscription. This is an opportunity to take any action before event delivery, including but not limited to:

- logging.

- auditing.
- security filtering/redaction.

3.1.18.1. Available Pre-Delivery Plugins

None.

3.2. Catalog Plugin Details

Installation and configuration details listed by plugin name.

3.2.1. Catalog Backup Plugin

The Catalog Backup Plugin is used to enable data backup of the catalog and the metacards it contains.

WARNING	*Catalog Backup Plugin Considerations* Using this plugin may impact performance negatively.

3.2.1.1. Installing the Catalog Backup Plugin

The Catalog Backup Plugin is installed by default with a standard installation in the Catalog application.

3.2.1.2. Configuring the Catalog Backup Plugin

To configure the Catalog Backup Plugin:

1. Navigate to the **Admin Console**.
2. Select **Catalog** application.
3. Select **Configuration** tab.
4. Select **Backup Post-Ingest Plugin**.

See Catalog Backup Plugin configurations for all possible configurations.

3.2.1.3. Usage Limitations of the Catalog Backup Plugin

- May affect performance.
- Must be installed prior to ingesting any content.
- Once enabled, disabling *may* cause incomplete backups.

3.2.2. Catalog Metrics Plugin

The Catalog Metrics Plugin captures metrics on catalog operations. These metrics can be viewed and analyzed using the Metrics Reporting Application in the Admin Console.

3.2.2.1. Related Components to the Source Metrics Plugin

- Source Metrics Plugin.

3.2.2.2. Installing the Catalog Metrics Plugin

The Catalog Metrics Plugin is installed by default with a standard installation in the Catalog application.

3.2.2.3. Configuring the Catalog Metrics Plugin

The Catalog Metrics Plugin has no configurable properties.

3.2.3. Catalog Policy Plugin

The Catalog Policy Plugin configures the attributes required for users to perform Create, Read, Update, and Delete operations on the catalog.

3.2.3.1. Installing the Catalog Policy Plugin

The Catalog Policy Plugin is installed by default with a standard installation in the Catalog application.

3.2.3.2. Configuring the Catalog Policy Plugin

To configure the Catalog Policy Plugin:

1. Navigate to the **Admin Console**.
2. Select Catalog application.
3. Select **Configuration** tab.
4. Select **Catalog Policy Plugin**.

See Catalog Policy Plugin configurations for all possible configurations.

3.2.4. Checksum Plugin

The Checksum plugin creates a unique checksum for resources input into the system to identify updated content.

3.2.4.1. Installing the Checksum Plugin

The Checksum is installed by default with a standard installation in the Catalog application.

3.2.4.2. Configuring the Checksum Plugin

The Checksum Plugin has no configurable properties.

3.2.5. Client Info Plugin

The client info plugin injects request-specific network information into request properties, such as Remote IP Address, Remote Host Name, Servlet Scheme, and Servlet Context.

3.2.5.1. Related Components to the Client Info Plugin

- Client info filter
- Metacard Ingest Network Plugin

3.2.5.2. Installing the Client Info Plugin

The Client Info Plugin is installed by default with a standard installation in the Catalog application.

3.2.5.3. Configuring the Client Info Plugin

The Client Info Plugin has no configurable properties.

3.2.6. Content URI Access Plugin

The Content URI Access Plugin prevents a Metacard's resource URI from being overridden by an incoming UpdateRequest.

3.2.6.1. Installing the Content URI Access Plugin

The Content URI Access Plugin is installed by default with a standard installation in the Catalog application.

3.2.6.2. Configuring the Content URI Access Plugin

The Content URI Access Plugin has no configurable properties.

3.2.7. Event Processor

The Event Processor creates, updates, and deletes subscriptions for event notification. These subscriptions optionally specify a filter criteria so that only events of interest to the subscriber are posted for notification.

As metacards are created, updated, and deleted, the Catalog's Event Processor is invoked (as a post-ingest plugin) for each of these events. The Event Processor applies the filter criteria for each registered subscription to each of these ingest events to determine if they match the criteria.

For more information on creating subscriptions, see Creating a Subscription.

3.2.7.1. Installing the Event Processor

The Event Processor is installed by default with a standard installation in the Catalog application.

3.2.7.2. Configuring the Event Processor

The Event Processor has no configurable properties.

3.2.7.3. Usage Limitations of the Event Processor

The Standard Event processor currently broadcasts federated events and should not. It should only broadcast events that were generated locally, all other events should be dropped. See DDF-3151 for status.

3.2.8. Expiration Date Pre-Ingest Plugin

The Expiration Date plugin adds or updates expiration dates which can be used later for archiving old data.

3.2.8.1. Installing the Expiration Date Pre-Ingest Plugin

The Expiration Date Pre-Ingest Plugin is not installed by default with a standard installation. To install:

1. Navigate to the **Admin Console**.
2. Select the **Catalog** application.
3. Select the **Configuration** tab.
4. Select the **Expiration Data Pre-Ingest Plugin**.

3.2.8.2. Configuring the Expiration Date Pre-Ingest Plugin

To configure the Expiration Date Pre-Ingest Plugin:

1. Navigate to the **Admin Console**.

2. Select the **Catalog** application.

3. Select the **Configuration** tab.

4. Select the **Expiration Date Pre-Ingest Plugin**.

See Expiration Date Plugin configurations for all possible configurations.

3.2.9. Filter Plugin

The Filter Plugin performs filtering on query responses as they pass through the framework.

Each metacard result can contain security attributes that are pulled from the metadata record after being processed by a `PolicyPlugin` that populates this attribute. The security attribute is a Map containing a set of keys that map to lists of values. The metacard is then processed by a filter plugin that creates a `KeyValueCollectionPermission` from the metacard's security attribute. This permission is then checked against the user subject to determine if the subject has the correct claims to view that metacard. The decision to filter the metacard eventually relies on the installed Policy Decision Point (PDP). The PDP that is being used returns a decision, and the metacard will either be filtered or allowed to pass through.

How a metacard gets filtered is left up to any number of FilterStrategy implementations that might be installed. Each FilterStrategy will return a result to the filter plugin that says whether or not it was able to process the metacard, along with the metacard or response itself. This allows a metacard or entire response to be partially filtered to allow some data to pass back to the requester. This could also include filtering any products sent back to a requester.

The security attributes populated on the metacard are completely dependent on the type of the metacard. Each type of metacard must have its own `PolicyPlugin` that reads the metadata being returned and then returns the appropriate attributes.

Example (represented as simple XML for ease of understanding):

```
<metacard>
    <security>
        <map>
            <entry assertedAttribute1="A,B" />
            <entry assertedAttribute2="X,Y" />
            <entry assertedAttribute3="USA,GBR" />
            <entry assertedAttribute4="USA,AUS" />
        </map>
    </security>
</metacard>
```

```
<user>
    <claim name="subjectAttribute1">
        <value>A</value>
        <value>B</value>
    </claim>
    <claim name="subjectAttribute2">
        <value>X</value>
        <value>Y</value>
    </claim>
    <claim name="subjectAttribute3">
        <value>USA</value>
    </claim>
    <claim name="subjectAttribute4">
        <value>USA</value>
    </claim>
</user>
```

In the above example, the user's claims are represented very simply and are similar to how they would actually appear in a SAML 2 assertion. Each of these user (or subject) claims will be converted to a `KeyValuePermission` object. These permission objects will be implied against the permission object generated from the metacard record. In this particular case, the metacard might be allowed if the policy is configured appropriately because all of the permissions line up correctly.

3.2.9.1. Installing the Filter Plugin

The Filter Plugin is installed by default with a standard installation in the Catalog application.

3.2.9.2. Configuring the Filter Plugin

The Filter Plugin has no configurable properties.

3.2.10. GeoCoder Plugin

The GeoCoder Plugin is a pre-ingest plugin that is responsible for populating the Metacard's `Location.COUNTRY_CODE` attribute if the Metacard has an associated location. If there is a valid country code for the Metacard, it will be in ISO 3166-1 alpha-3 format. If the metacard's country code is already populated, the plugin will **not** override it. The GeoCoder relies on either the WebService or Offline Gazetteer to retrieve country code information.

| WARNING | For a polygon or polygons, this plugin takes the center point of the bounding box to assign the country code. |

3.2.10.1. Installing the GeoCoder Plugin

The GeoCoder Plugin is installed by default with the Spatial application, when the WebService or Offline Gazetteer is started.

3.2.10.2. Configuring the GeoCoder Plugin

To configure the GeoCoder Plugin:

1. Navigate to the **Admin Console**.
2. Select **Spatial** application.
3. Select **Configuration** tab.
4. Select **GeoCoder Plugin**.

See GeoCoder Plugin configurations for all possible configurations.

3.2.11. Historian Policy Plugin

The Historian Policy Plugin protects metacard history from being edited or deleted by users without the history role (a `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/role` of `system-history`).

3.2.11.1. Installing the Historian Policy Plugin

The Historian is installed by default with a standard installation in the Catalog application.

3.2.11.2. Configuring the Historian Policy Plugin

The Historian Policy Plugin has no configurable properties.

3.2.12. Identification Plugin

The Identification Plugin assigns IDs to registry metacards and adds/updates IDs on create and update.

3.2.12.1. Installing the Identification Plugin

The Identification Plugin is not installed by default in a standard installation. It is installed by default with the Registry application.

3.2.12.2. Configuring the Identification Plugin

The Identification Plugin has no configurable properties.

3.2.13. JPEG2000 Thumbnail Converter

The JPEG2000 Thumbnail converter creates thumbnails from images ingested in jpeg2000 format.

3.2.13.1. Installing the JPEG2000 Thumbnail Converter

The JPEG2000 Thumbnail Converter is installed by default with a standard installation in the Catalog application.

3.2.13.2. Configuring the JPEG2000 Thumbnail Converter

The JPEG2000 Thumbnail Converter has no configurable properties.

3.2.14. Metacard Attribute Security Policy Plugin

The Metacard Attribute Security Policy Plugin combines existing metacard attributes to make new attributes and adds them to the metacard. For example, if a metacard has two attributes, `sourceattribute1` and `sourceattribute2`, the values of the two attributes could be combined into a new attribute, `destinationattribute1`. The `sourceattribute1` and `sourceattribute2` are the *source attributes* and `destinationattribute1` is the *destination attribute.*

There are two way to combine the values of source attributes. The first, and most common, is to take all of the attribute values and put them together. This is called the union. For example, if the source attributes `sourceattribute1` and `sourceattribute2` had the values:

`sourceattribute1 = MASK, VESSEL`

`sourceattribute2 = WIRE, SACK, MASK`

...the **union** would result in the new attribute `destinationattribute1`:

`destinationattribute1 = MASK, VESSEL, WIRE, SACK`

The other way to combine attributes is use the values common to all of the attributes. This is called the intersection. Using our previous example, the **intersection** of `sourceattribute1` and `sourceattribute2` would create the new attribute `destinationattribute1`

`destinationattribute1 = MASK`

because only `MASK` is common to all of the source attributes.

The policy plugin could also be used to rename attributes. If there is only one source attribute, and the combination policy is union, then the attribute's values are effectively renamed to the destination attribute.

3.2.14.1. Installing the Metacard Attribute Security Policy Plugin

The Metacard Attribute Security Policy Plugin is installed by default with a standard installation in the Catalog application.

See Metacard Attribute Security Policy Plugin configurations for all possible configurations.

3.2.15. Metacard Backup File Storage Provider

The Metacard Backup File Storage Provider is a storage provider that will store backed-up metacards in a specified file system location.

3.2.15.1. Installing the Metacard Backup File Storage Provider

To install the Metacard Backup File Storage Provider

1. Navigate to the **Admin Console**.
2. Select the **System** tab.
3. Select the **Features** tab.
4. Install the `catalog-metacard-backup-filestorage` feature.

3.2.15.2. Configuring the Metacard Backup File Storage Provider

To configure the Metacard Backup File Storage Provider

1. Navigate to the **Admin Console**.
2. Select Catalog application.
3. Select **Configuration** tab.
4. Select **Metacard Backup File Storage Provider**.

See Metacard Backup File Storage Provider configurations for all possible configurations.

3.2.16. Metacard Backup S3 Storage Provider

The Metacard Backup S3 Storage Provider is a storage provider that will store backed up metacards in the specified S3 bucket and key.

3.2.16.1. Installing the Metacard S3 File Storage Provider

To install the Metacard Backup File Storage Provider

1. Navigate to the **System** tab.

2. Select the **Features** tab.

3. Install the `catalog-metacard-backup-s3storage` feature.

3.2.16.2. Configuring the Metacard S3 File Storage Provider

To configure the Metacard Backup S3 Storage Provider:

1. Navigate to the **Admin Console**.

2. Select Catalog application.

3. Select **Configuration** tab.

4. Select **Metacard Backup S3 Storage Provider**.

See Metacard Backup S3 Storage Provider configurations for all possible configurations.

3.2.17. Metacard Groomer

The Metacard Groomer Pre-Ingest plugin makes modifications to `CreateRequest` and `UpdateRequest` metacards.

Use this pre-ingest plugin as a convenience to apply basic rules for your metacards.

This plugin makes the following modifications when metacards are in a `CreateRequest`:

- Overwrites the `Metacard.ID` field with a generated, unique, 32 character hexadecimal value if missing or if the resource URI is not a catalog resource URI.
- Sets `Metacard.CREATED` to the current time stamp if not already set.
- Sets `Metacard.MODIFIED` to the current time stamp if not already set.
- Sets `Core.METACARD_CREATED` to the current time stamp if not present.
- Sets `Core.METACARD_MODIFIED` to the current time stamp.

In an `UpdateRequest`, the same operations are performed as a `CreateRequest`, except:

- If no value is provided for `Metacard.ID` in the new metacard, it will be set using the `UpdateRequest` ID if applicable.

3.2.17.1. Installing the Metacard Groomer

The Metacard Groomer is included in the `catalog-core-plugins` feature. It is not recommended to uninstall this feature.

3.2.17.2. Configuring the Metacard Groomer

The Metacard Groomer has no configurable properties.

3.2.18. Metacard Ingest Network Plugin

The Metacard Ingest Network Plugin allows the conditional insertion of new attributes on metacards during ingest based on network information from the ingest request; including IP address and hostname.

For the extent of this section, a 'rule' will refer to a configured, single instance of this plugin.

3.2.18.1. Related Components to the Metacard Ingest Network Plugin

- Client Info Plugin

3.2.18.2. Installing the Metacard Ingest Network Plugin

The Metacard Ingest Network Plugin is installed by default during a standard installation in the Catalog application.

3.2.18.3. Configuring the Metacard Ingest Network Plugin

To configure the Metacard Ingest Network Plugin:

- Navigate to the **Admin Console**.
- Select the Catalog application.
- Select the **Configuration** tab.
- Select the label *Metacard Ingest Network Plugin* to setup a network rule.

See Metacard Ingest Network Plugin configurations for all possible configurations.

Multiple instances of the plugin can be configured by clicking on its configuration title within the configuration tab of the Catalog app. Each instance represents a conditional statement, or a 'rule', that gets evaluated for each ingest request. For any request that meets the configured criteria of a rule, that rule will attempt to transform its list of key-value pairs to become new attributes on all metacards in that request.

The rule is divided into two fields: "Criteria" and "Expected Value". The "Criteria" field features a drop-down list containing the four elements for which equality can be tested:

- IP Address of where the ingest request came from
- Host Name of where the ingest request came from
- Scheme that the ingest request arrived on, for example, *http* vs *https*

- Context Path that the ingest request arrived on, for example, */services/catalog*

In order for a rule to evaluate to true and the attributes be applied, the value in the "Expected Value" field must be an exact match to the actual value of the selected criteria. For example, if the selected criteria is "IP Address" with an expected value of "192.168.0.1", the rule only evaluates to true for ingest requests coming from "192.168.0.1" and nowhere else.

IMPORTANT	*Check for IPv6* Verify your system's IP configuration. Rules using "IP Address" may need to be written in IPv6 format.

The key-value pairs within each rule should take the following form: "key = value" where the "key" is the name of the attribute and the "value" is the value assigned to that attribute. Whitespace is ignored unless it is within the key or value. Multi-valued attributes can be expressed in comma-separated format if necessary.

Examples of Valid Attribute Assignments

```
contact.contributor-name = John Doe
contact.contributor-email = john.doe@example.net
language = English
language = English, French, German
security.access-groups = SJ202, SR 101, JS2201
```

3.2.18.3.1. Useful Attributes

The following table provides some useful attributes that may commonly be set by this plugin:

Table 17. Useful Attributes

Attribute Name	Expected Format	Multi-Valued
expiration	ISO DateTime	no
description	Any String	no
metacard.owner	Any String	no
language	Any String	yes
security.access-groups	Any String	yes
security.access-individuals	Any String	yes

3.2.18.4. Usage Limitations of the Metacard Ingest Network Plugin

- This plugin only works for ingest (create requests) performed over a network; data ingested via command line does not get processed by this plugin.

- Any attribute that is already set on the metacard will not be overwritten by the plugin.

- The order of execution is not guaranteed. For any rule configuration where two or more rules add different values for the same attribute, it is undefined what the final value for that attribute will be in the case where more than one of those rules evaluates to true.

3.2.19. Metacard Resource Size Plugin

This post-query plugin updates the resource size attribute of each metacard in the query results if there is a cached file for the product and it has a size greater than zero; otherwise, the resource size is unmodified and the original result is returned.

Use this post-query plugin as a convenience to return query results with accurate resource sizes for cached products.

3.2.19.1. Installing the Metacard Resource Size Plugin

The Metacard Resource Size Plugin is installed by default with a standard installation.

3.2.19.2. Configuring the Metacard Resource Size Plugin

The Metacard Resource Size Plugin has no configurable properties.

3.2.20. Metacard Validity Filter Plugin

The Metacard Validity Filter Plugin determines whether metacards with validation errors or warnings are filtered from query results.

3.2.20.1. Related Components to the Metacard Validity Filter Plugin

- Metacard Validity Marker.

3.2.20.2. Installing the Metacard Validity Filter Plugin

The Metacard Validity Filter Plugin is installed by default with a standard installation in the Catalog application.

3.2.21. Metacard Validity Marker

The Metacard Validity Marker Pre-Ingest plugin modifies the metacards contained in create and update requests.

The plugin runs each metacard in the `CreateRequest` and `UpdateRequest` against each registered `MetacardValidator` service.

	This plugin can make it seem like ingested products are not successfully ingested if a
NOTE	user does not have permissions to access invalid metacards. If an ingest did not fail, there are no errors in the ingest log, but the expected results do not show up after a query, verify either that the ingested data is valid or that the Metacard Validity Filter Plugin is configured to show warnings and/or errors.

3.2.21.1. Related Components to the Metacard Validity Marker

- Metacard Validity Filter Plugin.

3.2.21.2. Installing Metacard Validity Marker

This plugin is installed by default with a standard installation in the Catalog application.

3.2.21.3. Configuring Metacard Validity Marker

See Metacard Validity Marker Plugin configurations for all possible configurations.

3.2.21.4. Using Metacard Validity Marker

Use this pre-ingest plugin to validate metacards against metacard validators, which can check schemas, schematron, or any other logic.

3.2.22. Operation Plugin

The operation plugin validates the subject's security attributes to ensure they are adequate to perform the operation.

3.2.22.1. Installing the Operation Plugin

The Operation Plugin is installed by default with a standard installation in the Catalog application.

3.2.22.2. Configuring the Operation Plugin

The Operation Plugin has no configurable properties.

3.2.23. Point of Contact Policy Plugin

The Point of Contact Policy Plugin is a PreUpdate plugin that will check if the point-of-contact attribute has changed. If it does, then it adds a policy to that metacard's policy map that cannot be implied. This will deny such an update request, which essentially makes the point-of-contact attribute read-only.

3.2.23.1. Related Components to Point of Contact Policy Plugin

Point of Contact Update Plugin

3.2.23.2. Installing the Point of Contact Policy Plugin

The Point of Contact Policy Plugin is installed by default with a standard installation in the Catalog application.

3.2.23.3. Configuring the Point of Contact Policy Plugin

The Point of Contact Policy Plugin has no configurable properties.

3.2.24. Processing Post-Ingest Plugin

The Processing Post Ingest Plugin is responsible for submitting catalog Create, Update, and Delete (CUD) requests to the Processing Framework.

3.2.24.1. Related Components to Processing Post-Ingest Plugin

None.

3.2.24.2. Installing the Processing Post-Ingest Plugin

The Processing Post-Ingest Plugin is not installed by default with a standard installation, but is installed by default when the in-memory Processing Framework is installed.

3.2.24.3. Configuring the Processing Post-Ingest Plugin

The Processing Post-Ingest Plugin has no configurable properties.

3.2.25. Registry Policy Plugin

The Registry Policy Plugin defines the policies for user access to registry entries and operations.

3.2.25.1. Installing the Registry Policy Plugin

The Registry Policy Plugin is not installed by default on a standard installation. It is installed with the Registry application.

3.2.25.2. Configuring the Registry Policy Plugin

The Registry Policy Plugin can be configured from the Admin Console:

1. Navigate to the **Admin Console**.

2. Select the **Registry** application.

3. Select the **Configuration** tab.

4. Select **Registry Policy Plugin**.

See Registry Policy Plugin configurations for all possible configurations.

3.2.26. Resource URI Policy Plugin

The Resource URI Policy Plugin configures the attributes required for users to set the resource URI when creating a metacard or alter the resource URI when updating an existing metacard in the catalog.

3.2.26.1. Installing the Resource URI Policy Plugin

The Resource URI Policy Plugin is installed by default with a standard installation in the Catalog application.

3.2.26.2. Configuring the Resource URI Policy Plugin

To configure the Resource URI Policy Plugin:

1. Navigate to the **Admin Console**.

2. Select Catalog application.

3. Select **Configuration** tab.

4. Select **Resource URI Policy Plugin**.

See Resource URI Policy Plugin configurations for all possible configurations.

3.2.27. Resource Usage Plugin

The Resource Usage Plugin monitors and limits data usage, and enables cancelling long-running queries.

3.2.27.1. Installing the Resource Usage Plugin

The Resource Usage Plugin is not installed by default with a standard installation. It is installed with the Resource Management application.

3.2.27.2. Configuring the Resource Usage Plugin

The Resource Usage Plugin can be configured from the Admin Console:

1. Navigate to the **Admin Console**.

2. Select the **Resource Management** application.

3. Select the **Configuration** tab.

4. Select **Data Usage**.

See Resource Usage Plugin configurations for all possible configurations.

3.2.28. Security Audit Plugin

The Security Audit Plugin is used to allow the auditing of specific metacard attributes. Any time a metacard attribute listed in the configuration is updated, a log will be generated in the security log.

3.2.28.1. Installing the Security Audit Plugin

The Security Audit Plugin is installed by default with a standard installation in the Catalog application.

3.2.29. Security Logging Plugin

The Security Logging Plugin logs operations to the security log.

3.2.29.1. Installing Security Logging Plugin

The Security Logging Plugin is installed by default in a standard installation in the Security application.

3.2.29.2. Enhancing the Security Log

The security log contains attributes related to the subject acting on the system. To add additional attributes related to the subject to the logs, append the attribute's key to the comma separated values assigned to `security.logger.extra_attributes` in `/etc/custom.system.properties`.

3.2.30. Security Plugin

The Security Plugin identifies the subject for an operation.

3.2.30.1. Installing the Security Plugin

The Security Plugin is installed by default with a standard installation in the Catalog application.

3.2.30.2. Configuring the Security Plugin

The Security Plugin has no configurable properties.

3.2.31. Source Metrics Plugin

The Source Metrics Plugin captures metrics on catalog operations. These metrics can be viewed and analyzed using the Metrics Reporting Application in the Admin Console.

3.2.31.1. Related Components to the Source Metrics Plugin

- Catalog Metrics Plugin.

3.2.31.2. Installing the Source Metrics Plugin

The Source Metrics Plugin is installed by default with a standard installation in the Catalog application.

3.2.31.3. Configuring the Source Metrics Plugin

The Source Metrics Plugin has no configurable properties.

3.2.32. Tags Filter Plugin

The Tags Filter Plugin updates queries without filters for tags, and adds a default tag of `resource`. For backwards compatibility, a filter will also be added to include metacards without any tags attribute.

3.2.32.1. Related Components to Tags Filter Plugin

None.

3.2.32.2. Installing the Tags Filter Plugin

The Tags Filter Plugin is installed by default with a standard installation in the Catalog application.

3.2.32.3. Configuring the Tags Filter Plugin

The Tags Filter Plugin has no configurable properties.

3.2.33. Video Thumbnail Plugin

The Video Thumbnail Plugin provides the ability to generate thumbnails for video files stored in the Content Repository.

It is an implementation of both the `PostCreateStoragePlugin` and `PostUpdateStoragePlugin` interfaces. When installed, it is invoked by the Catalog Framework immediately after a content item has been created or updated by the Storage Provider.

This plugin uses a custom 32-bit LGPL build of FFmpeg (a video processing program) to generate

thumbnails. When this plugin is installed, it places the FFmpeg executable appropriate for the current operating system in `<DDF_HOME>/bin_third_party/ffmpeg`. When invoked, this plugin runs the FFmpeg binary in a separate process to generate the thumbnail. The `<DDF_HOME>/bin_third_party/ffmpeg` directory is deleted when the plugin is uninstalled.

| NOTE | Prebuilt FFmpeg binaries are provided for Linux, Mac, and Windows only. |

3.2.33.1. Installing the Video Thumbnail Plugin

The Video Thumbnail Plugin is installed by default with a standard installation in the Catalog application.

3.2.33.2. Configuring the Video Thumbnail Plugin

To configure the Video Thumbnail Plugin:

1. Navigate to the **Admin Console**.
2. Select the **Catalog** application.
3. Select the **Configuration** tab.
4. Select the **Video Thumbnail Plugin**.

See Video Thumbnail Plugin configurations for all possible configurations.

3.2.34. Workspace Access Plugin

The Workspace Access Plugin prevents non-owner users from changing workspace permissions.

3.2.34.1. Related Components to The Workspace Access Plugin

- Workspace Sharing Policy Plugin.
- Workspace Pre-Ingest Plugin.
- Workspace Extension.

3.2.34.2. Installing the Workspace Access Plugin

The Workspace Access Plugin is installed by default with a standard installation in the Catalog application.

3.2.34.3. Configuring the Workspace Access Plugin

The Workspace Access Plugin has no configurable properties.

3.2.35. Workspace Pre-Ingest Plugin

The Workspace Pre-Ingest Plugin verifies that a workspace has an associated email to enable sharing and assigns that email as "owner".

3.2.35.1. Related Components to The Workspace Pre-Ingest Plugin

- Workspace Sharing Policy Plugin.
- Workspace Access Plugin.
- Workspace Extension.

3.2.35.2. Installing the Workspace Pre-Ingest Plugin

The Workspace Pre-Ingest Plugin is installed by default with a standard installation in the Catalog application.

3.2.35.3. Configuring the Workspace Pre-Ingest Plugin

The Workspace Pre-Ingest Plugin has no configurable properties.

3.2.36. Workspace Sharing Policy Plugin

The Workspace Sharing Policy Plugin collects attributes for a workspace to identify the appropriate policy to apply to allow sharing.

3.2.36.1. Related Components to The Workspace Sharing Policy Plugin

- Workspace Access Plugin.
- Workspace Pre-Ingest Plugin.
- Workspace Extension.

3.2.36.2. Installing the Workspace Sharing Policy Plugin

The Workspace Sharing Policy Plugin is installed by default with a standard installation in the Catalog application.

3.2.36.3. Configuring the Workspace Sharing Policy Plugin

The Workspace Sharing Policy Plugin has no configurable properties.

3.2.37. XML Attribute Security Policy Plugin

The XML Attribute Security Policy Plugin parses XML metadata contained within a metacard for security attributes on any number of XML elements in the metadata. The configuration for the plugin contains one field for setting the XML elements that will be parsed for security attributes and the other two configurations contain the XML attributes that will be pulled off of those elements. The **Security Attributes (union)** field will compute the union of values for each attribute defined and the **Security Attributes (intersection)** field will compute the intersection of values for each attribute defined.

3.2.37.1. Installing the XML Attribute Security Policy Plugin

The XML Attribute Security Policy Plugin is installed by default with a standard installation in the Security application.

4. Operations
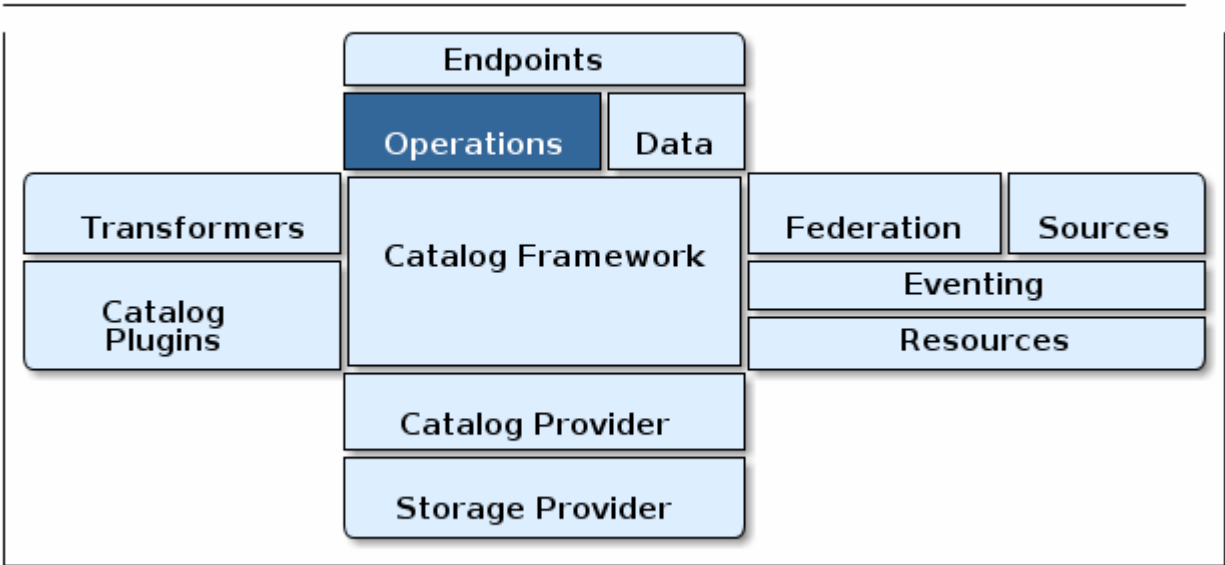
The Catalog provides the capability to query, create, update, and delete metacards; retrieve resources; and retrieve information about the sources in the enterprise.

Each of these operations follow a request/response paradigm. The request is the input to the operation and contains all of the input parameters needed by the Catalog Framework's operation to communicate with the Sources. The response is the output from the execution of the operation that is returned to the client, which contains all of the data returned by the sources. For each operation there is an associated request/response pair, e.g., the `QueryRequest` and `QueryResponse` pair for the Catalog Framework's query operation.

All of the request and response objects are extensible in that they can contain additional key/value properties on each request/response. This allows additional capability to be added without changing the Catalog API, helping to maintain backwards compatibility.

5. Resources
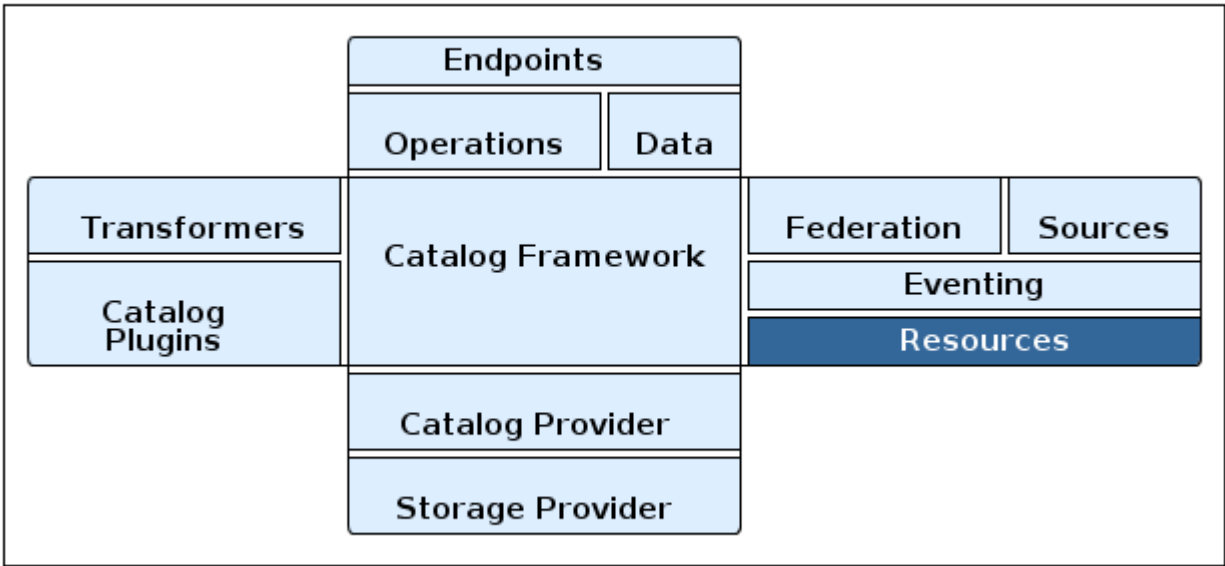
Resources Architecture

Resources are the data that is represented by the cataloged metadata in DDF.

Metacards are used to describe those resources through metadata. This metadata includes the time the resource was created, the location where the resource was created, etc. A DDF `Metacard` contains the `getResourceUri` method, which is used to locate and retrieve its corresponding resource.
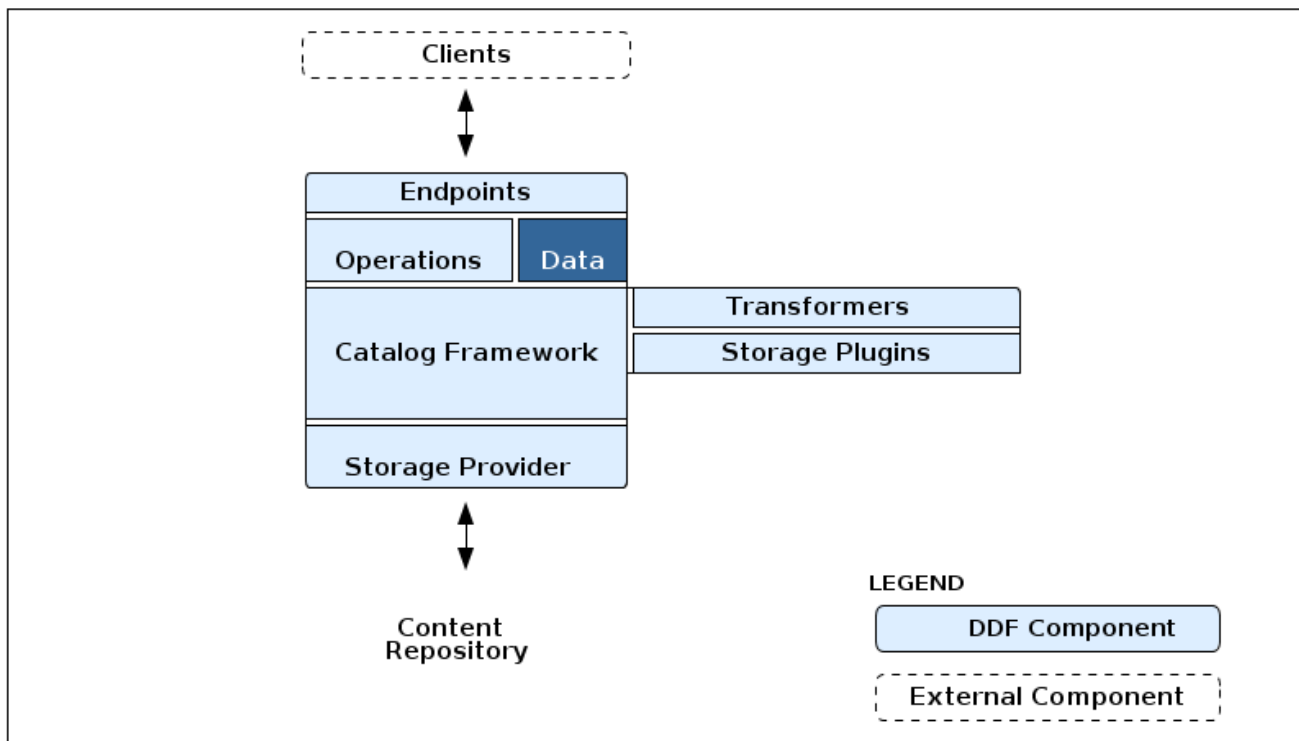
Content Data Component Architecture

5.1. Content Item

ContentItem is the domain object populated by the Storage Provider that represents the information about the content to be stored or content that has been stored in the Storage Provider. A ContentItem encapsulates the content's globally unique ID, mime type, and input stream (i.e., the actual content). The unique ID of a ContentItem will always correspond to a Metacard ID.

5.1.1. Retrieving Resources

When a client attempts to retrieve a resource, it must provide a metacard ID or URI corresponding to a unique resource. As mentioned above, the resource URI is obtained from a `Metacard's` `getResourceUri` method. The `CatalogFramework` has three methods that can be used by clients to obtain a resource: `getEnterpriseResource`, `getResource`, and `getLocalResource`. The `getEnterpriseResource` method invokes the `retrieveResource` method on a local `ResourceReader` as well as all the `Federated` and `Connected` Sources inthe DDF enterprise. The second method, `getResource`, takes in a source ID as a parameter and only invokes `retrieveResource` on the specified `Source`. The third method invokes `retrieveResource` on a local `ResourceReader`.

The parameter for each of these methods in the `CatalogFramework` is a `ResourceRequest`. DDF includes two implementations of `ResourceRequest`: `ResourceRequestById` and `ResourceRequestByProductUri`. Since these implementations extend `OperationImpl`, they can pass a `Map` of generic properties through

the `CatalogFramework` to customize how the resource request is carried out. One example of this is explained in the Retrieving Resource Options section below. The following is a basic example of how to create a `ResourceRequest` and invoke the `CatalogFramework` resource retrieval methods to process the request.

Retrieve Resource Example

```
Map<String, Serializable> properties = new HashMap<String, Serializable>();
properties.put("PropertyKey1", "propertyA"); //properties to customize Resource retrieval
ResourceRequestById resourceRequest = new ResourceRequestById(
"0123456789abcdef0123456789abcdef", properties); //object containing ID of Resource to be
retrieved
String sourceName = "LOCAL_SOURCE"; //the Source ID or name of the local Catalog or a
Federated Source
ResourceResponse resourceResponse; //object containing the retrieved Resource and the
request that was made to get it.
resourceResponse = catalogFramework.getResource(resourceRequest, sourceName); //Source-
based retrieve Resource request
Resource resource = resourceResponse.getResource(); //actual Resource object containing
InputStream, mime type, and Resource name
```

`DDF.catalog.resource.ResourceReader` instances can be discovered via the OSGi Service Registry. The system can contain multiple `ResourceReaders`. The `CatalogFramework` determines which one to call based on the scheme of the resource's URI and what schemes the `ResourceReader` supports. The supported schemes are obtained by a `ResourceReader's` `getSupportedSchemes` method. As an example, one `ResourceReader` may know how to handle file-based URIs with the scheme `file`, whereas another `ResourceReader` may support HTTP-based URIs with the scheme `http`.

The `ResourceReader` or `Source` is responsible for locating the resource, reading its bytes, adding the binary data to a `Resource` implementation, then returning that `Resource` in a `ResourceResponse`. The `ResourceReader` or `Source` is also responsible for determining the `Resource's name and mime type`, which it sends back in the `Resource` implementation.

5.1.1.1. BinaryContent

`BinaryContent` is an object used as a container to store translated or transformed DDF components. `Resource` extends `BinaryContent` and includes a `getName` method. `BinaryContent` has methods to get the `InputStream`, `byte` array, MIME type, and size of the represented binary data. An implementation of `BinaryContent` (`BinaryContentImpl`) can be found in the Catalog API in the `DDF.catalog.data` package.

5.1.2. Retrieving Resource Options

Options can be specified on a retrieve resource request made through any of the supporting endpoint. To specify an option for a retrieve resource request, the endpoint needs to first instantiate a `ResourceRequestByProductUri` or a `ResourceRequestById`. Both of these `ResourceRequest` implementations allow a `Map` of properties to be specified. Put the specified option into

the `Map` under the key `RESOURCE_OPTION`.

Retrieve Resource with Options

```
Map<String, Serializable> properties = new HashMap<String, Serializable>();
properties.put("RESOURCE_OPTION", "OptionA");
ResourceRequestById resourceRequest = new ResourceRequestById(
"0123456789abcdef0123456789abcdef", properties);
```

Depending on the support that the `ResourceReader` or `Source` provides for options, the `properties``Map` will be checked for the `RESOURCE_OPTION` entry. If that entry is found, the option will be handled. If the `ResourceReader` or `Source` does not support options, that entry will be ignored.

A new `ResourceReader` or `Source` implementation can be created to support options in a way that is most appropriate. Since the option is passed through the catalog framework as a property, the `ResourceReader` or `Source` will have access to that option as long as the endpoint supports options.

5.1.3. Storing Resources

Resources are saved using a `ResourceWriter`. `DDF.catalog.resource.ResourceWriter` instances can be discovered via the OSGi Service Registry. Once retrieved, the `ResourceWriter` instance provides clients with a way to store resources and get a corresponding URI that can be used to subsequently retrieve the resource via a `ResourceReader`. Simply invoke either of the `storeResource` methods with a resource and any potential arguments. The `ResourceWriter` implementation is responsible for determining where the resource is saved and how it is saved. This allows flexibility for a resource to be saved in any one of a variety of data stores or file systems. The following is an example of how to use a generic implementation of `ResourceWriter`.

Using a ResourceWriter

```
InputStream inputStream = <Video_Input_Stream>; //InputStream of raw Resource data
MimeType mimeType = new MimeType("video/mpeg"); //Mime Type or content type of Resource
String name = "Facility_Video";  //Descriptive Resource name
Resource resource = new ResourceImpl(inputStream, mimeType, name);
Map<String, Object> optionalArguments = new HashMap<String, Object>();
ResourceWriter writer = new ResourceWriterImpl();
URI resourceUri; //URI that can be used to retrieve Resource
resourceUri = writer.storeResource(resource, optionalArguments); //Null can be passed in
here
```

5.2. Resource Components

Resource components are used when working with resources

A resource is a URI-addressable entity that is represented by a metacard. Resources may also be known

as **products** or **data**.

Resources may exist either locally or on a remote data store.

Examples of resources include:

- NITF image
- MPEG video
- Live video stream
- Audio recording
- Document

A resource object in DDF contains an `InputStream` with the binary data of the resource. It describes that resource with a name, which could be a file name, URI, or another identifier. It also contains a mime type or content type that a client can use to interpret the binary data.

5.3. Resource Readers

A resource reader retrieves resources associated with metacards via URIs. Each resource reader must know how to interpret the resource's URI and how to interact with the data store to retrieve the resource.

There can be multiple resource readers in a Catalog instance. The `Catalog Framework` selects the appropriate resource reader based on the scheme of the resource's URI.

In order to make a resource reader available to the Catalog Framework, it must be exported to the OSGi Service Registry as a `DDF.catalog.resource.ResourceReader`.

5.3.1. URL Resource Reader

The `URLResourceReader` is an implementation of `ResourceReader` which is included in the DDF Catalog. It obtains a resource given an http, https, or file-based URL. The `URLResourceReader` will connect to the provided Resource URL and read the resource's bytes into an `InputStream`.

WARNING	When a resource linked using a file-based URL is in the product cache, the `URLResourceReader`'s rootResourceDirectories is not checked when downloading the product. It is downloaded from the product cache which bypasses the `URLResourceReader`. For example, if path `/my/valid/path` is configured in the `URLResourceReader`'s rootResourceDirectories and one downloads the product with resource-uri `file:///my/valid/path/product.txt` and then one removes `/my/valid/path` from the `URLResourceReader`'s `rootResourceDirectories` configuration, the product will still be accessible via the product cache.

5.3.1.1. Installing the URL Resource Reader

The `URLResourceReader` is installed by default with a standard installation in the Catalog application.

5.3.1.2. Configuring Permissions for the URL Resource Reader

Configuring the URL Resource Reader to retrieve files requires adding Security Manager read permissions to the directory containing the resources. The following permissions, replacing <DIRECTORY_PATH> with the path of the directory containing resources should be placed in the URL Resource Reader section inside <DDF_HOME>/security/configurations.policy.

> **WARNING**
>
> *Adding New Permissions*
>
> After adding permissions, a system restart is required for them to take effect.

1. permission java.io.FilePermission "<DIRECTORY_PATH>", "read";
2. permission java.io.FilePermission "<DIRECTORY_PATH>${/}-", "read";

Trailing slashes after <DIRECTORY_PATH> have no effect on the permissions granted. For example, adding a permission for "${/}test${/}path" and "${/}test${/}path${/}" are equivalent. The recursive forms "${/}test${/}path${/}-", and "${/}test${/}path${/}${/}-" are also equivalent.

Line 1 gives the URL Resource Reader the permissions to read from the directory. Line 2 gives the URL Resource Reader the permissions to recursively read from the directory, specified by the directory path's suffix "${/}-".

5.3.1.3. Configuring the URL Resource Reader

Configure the URL Resource Reader from the Admin Console.

1. Navigate to the **Admin Console**.
2. Select the **Catalog** application.
3. Select the **Configuration** tab.
4. Select the **URL Resource Reader**.

See URL Resource Reader configurations for all possible configurations.

5.3.1.4. Using the URL Resource Reader

`URLResourceReader` will be used by the Catalog Framework to obtain a resource whose metacard is cataloged in the local data store. This particular `ResourceReader` will be chosen by the `CatalogFramework` if the requested resource's URL has a protocol of `http`, `https`, or `file`.

For example, requesting a resource with the following URL will make the Catalog Framework invoke the `URLResourceReader` to retrieve the product.

```
file:///home/users/DDF_user/data/example.txt
```

If a resource was requested with the URL `udp://123.45.67.89:80/SampleResourceStream`, the `URLResourceReader` would *not* be invoked.

Supported Schemes:

- http

- https

- file

NOTE	If a file-based URL is passed to the `URLResourceReader`, that file path needs to be accessible by the DDF instance.

5.4. Resource Writers

A resource writer stores a resource and produces a URI that can be used to retrieve the resource at a later time. The resource URI uniquely locates and identifies the resource. Resource writers can interact with an underlying data store and store the resource in the proper place. Each implementation can do this differently, providing flexibility in the data stores used to persist the resources.

Resource Writers should be used within the Content Framework if and when implementing a custom Storage Provider to store the product. The default Storage Provider that comes with the DDF writes the products to the file system.

6. Queries

Clients use `ddf.catalog.operation.Query` objects to describe which metacards are needed from Sources.

Query objects have two major components:

- Filters

- Query Options

A Source uses the Filter criteria constraints to find the requested set of metacards within its domain of metacards. The Query Options are used to further restrict the Filter's set of requested metacards.

6.1. Filters

An OGC Filter is a Open Geospatial Consortium (OGC) standard ⧉ that describes a query expression in

terms of Extensible Markup Language (XML) and key-value pairs (KVP). The OGC Filter is used to represent a query to be sent to sources and the Catalog Provider, as well as to represent a Subscription. The OGC Filter provides support for expression processing, such as adding or dividing expressions in a query, but that is not the intended use for DDF.

The Catalog Framework does not use the XML representation of the OGC Filter standard. DDF instead uses the Java implementation provided by GeoTools ⧉. GeoTools provides Java equivalent classes for OGC Filter XML elements. GeoTools originally provided the standard Java classes for the OGC Filter Encoding 1.0 under the package name `org.opengis.filter`. The same package name is used today and is currently used by DDF. Java developers do not parse or view the XML representation of a Filter in DDF. Instead, developers use only the Java objects to complete query tasks.

Note that the `ddf.catalog.operation.Query` interface extends the `org.opengis.filter.Filter` interface, which means that a Query object is an OGC Java Filter with Query Options.

A Query is an OGC Filter

```
public interface Query extends Filter
```

6.1.1. FilterBuilder API

To avoid the complexities of working with the Filter interface directly and implementing the DDF Profile of the Filter specification, the Catalog includes an API, primarily in `DDF.filter`, to build Filters using a fluent API.

To use the `FilterBuilder` API, an instance of `DDF.filter.FilterBuilder` should be used via the OSGi registry. Typically, this will be injected via a dependency injection framework. Once an instance of `FilterBuilder` is available, methods can be called to create and combine Filters.

| TIP | The fluent API is best accessed using an IDE that supports code-completion. For additional details, refer to the [Catalog API Javadoc]. |

6.1.2. Boolean Operators

Filters use a number of boolean operators.

`FilterBuilder.allOf(Filter ⋯)`
  creates a new Filter that requires all provided Filters are satisfied (Boolean AND), either from a List or Array of Filter instances.

`FilterBuilder.anyOf(Filter ⋯)`
  creates a new Filter that requires at least one of the provided Filters are satisfied (Boolean OR), either from a List or Array of Filter instances.

`FilterBuilder.not(Filter filter)`
  creates a new Filter that requires the provided Filter must not match (Boolean NOT).

6.1.3. Attribute

Filters can be based on specific attributes.

`FilterBuilder.attribute(String attributeName)`:: begins a fluent API for creating an Attribute-based Filter, i.e., a Filter that matches on Metacards with Attributes of a particular value.

6.1.4. XPath

Filters can be based on XML attributes.

`FilterBuilder.xpath(String xpath)`:: begins a fluent API for creating an XPath-based Filter, i.e., a Filter that matches on Metacards with Attributes of type XML that match when evaluating a provided XPath selector.

Contextual Operators

```
FilterBuilder.attribute(attributeName).is().like().text(String contextualSearchPhrase);
FilterBuilder.attribute(attributeName).is().like().caseSensitiveText(StringcaseSensitiveC
ontextualSearchPhrase);
FilterBuilder.attribute(attributeName).is().like().fuzzyText(String fuzzySearchPhrase);
```

7. Metrics

DDF includes a system of data-collection to enable monitoring system health, user interactions, and overall system performance: **Metrics Collection**.

The Metrics Collection Application collects data for all of the pre-configured metrics in DDF and stores them in custom JMX Management Bean (MBean) attributes. Samples of each metric's data is collected every 60 seconds and stored in the `<DDF_HOME>/data/metrics` directory with each metric stored in its own `.rrd` file. Refer to the Metrics Reporting Application for how the stored metrics data can be viewed.

| **WARNING** | Do not remove the `<DDF_HOME>/data/metrics` directory or any files in it. If this is done, all existing metrics data will be permanently lost. |
| | Also note that if DDF is uninstalled/re-installed that all existing metrics data will be permanently lost. |

Types of Metrics Collected

Catalog Metrics

Metrics collected about the catalog status.

Source Metrics

Metrics collected per source.

7.1. Metrics Collection Application

The Metrics Collection Application is responsible for collecting both Catalog and Source metrics.

Use Metrics Collection to collect historical metrics data, such as catalog query metrics, message latency, or individual sources' metrics type of data.

7.1.1. Installing Metrics Collection

The Metrics Collection application is installed by default with a standard installation.

The catalog-level metrics are packaged as the `catalog-core-metricsplugin` feature, and the source-level metrics are packaged as the `catalog-core-sourcemetricsplugin` feature.

7.1.2. Configuring Metrics Collection

No configuration is made for the Metrics Collection application. All metrics collected are either pre-configured in DDF or dynamically created as sources are created or deleted.

7.1.3. Catalog Metrics

Table 18. Catalog Metrics Collected

Metric	JMX MBean Name	MBean Attribute Name	Description
Catalog Exceptions	ddf.metrics.catalog:name=Exceptions	Count	The number of exceptions, of all types, thrown across all catalog queries executed.
Catalog Exceptions Federation	ddf.metrics.catalog:name=Exceptions.Federation	Count	The total number of Federation exceptions thrown across all catalog queries executed.
Catalog Exceptions Source Unavailable	ddf.metrics.catalog:name=Exceptions.SourceUnavailable	Count	The total number of `SourceUnavailable` exceptions thrown across all catalog queries executed. These exceptions occur when the source being queried is currently not available.
Catalog Exceptions Unsupported Query	ddf.metrics.catalog:name=Exceptions.UnsupportedQuery	Count	Total number of `UnsupportedQuery` exceptions thrown across all catalog queries executed. These exceptions occur when the query being executed is not supported or is invalid.
Catalog Ingest Created	ddf.metrics.catalog:name=Ingest.Created	Count	The number of catalog entries created in the Metadata Catalog.

Metric	JMX MBean Name	MBean Attribute Name	Description
Catalog Ingest Deleted	ddf.metrics.catalog:name=Ingest.Deleted	The number of catalog entries deleted from the Metadata Catalog.	Count
Catalog Ingest Updated	ddf.metrics.catalog:name=Ingest.Updated	Count	The number of catalog entries updated in the Metadata Catalog.
Catalog Queries	ddf.metrics.catalog:name=Queries	Count	The number of queries attempted.
Catalog Queries Comparison	ddf.metrics.catalog:name=Queries.Comparison	Count	The number of queries attempted that included a string comparison criteria as part of the search criteria, e.g., `PropertyIsLike`, `PropertyIsEqualTo`, etc.
Catalog Queries Federated	ddf.metrics.catalog:name=Queries.Federated	Count	The number of federated queries attempted.
Catalog Queries **Fuzzy**	ddf.metrics.catalog:name=Queries.Fuzzy	Count	The number of queries attempted that included a string comparison criteria with **fuzzy** searching enabled as part of the search criteria.
Catalog Queries Spatial	ddf.metrics.catalog:name=Queries.Spatial	Count	The number of queries attempted that included a spatial criteria as part of the search criteria.
Catalog Queries Temporal	ddf.metrics.catalog:name=Queries.Temporal	Count	The number of queries attempted that included a temporal criteria as part of the search criteria.
Catalog Queries Total Results	ddf.metrics.catalog:name=Queries.TotalResults	Mean	The average of the total number of results returned from executed queries. This total results data is averaged over the metric's sample rate.
Catalog Queries Xpath	ddf.metrics.catalog:name=Queries.Xpath	Count	The number of queries attempted that included a Xpath criteria as part of the search criteria.

Metric	JMX MBean Name	MBean Attribute Name	Description
Catalog Resource Retrieval	ddf.metrics.catalog:name=Resource	Count	The number of resources retrieved.
Services Latency	ddf.metrics.services:name=Latency	Mean	The response time (in milliseconds) from receipt of the request at the endpoint until the response is about to be sent to the client from the endpoint. This response time data is averaged over the metric's sample rate.

7.1.4. Source Metrics

Metrics are also collected on a per source basis for each configured Federated Source and Catalog Provider. When the source is configured, the metrics listed in the table below are automatically created. Metrics are collected for each request(whether enterprise query or a source-specific query). When the source is deleted (or renamed), the associated metrics' MBeans and Collectors are also deleted. However, the RRD file in the `data/metrics` directory containing the collected metrics remain indefinitely and remain accessible from the **Metrics** tab in the Admin Console.

In the table below, the metric name is based on the Source's ID (indicated by `<sourceId>`).

Table 19. Source Metrics Collected

Metric	JMX MBean Name	MBean AttributeName	Description
Source <sourceId> Exceptions	`ddf.metrics.catalog.source:name=<sourceId>.Exceptions`	Count	A count of the total number of exceptions, of all types, thrown from catalog queries executed on this source.
Source <sourceId> Queries	`ddf.metrics.catalog.source:name=<sourceId>.Queries`	Count	A count of the number of queries attempted on this source.
Source <sourceId> Queries Total Results	`ddf.metrics.catalog.source:name=<sourceId>.Queries.TotalResults`	Mean	An average of the total number of results returned from executed queries on this source. This total results data is averaged over the metric's sample rate.

For example, if a Federated Source was created with a name of `fs-1`, then the following metrics would be created for it:

- Source Fs1 Exceptions
- Source Fs1 Queries
- Source Fs1 Queries Total Results

If this federated source is then renamed to `fs-1-rename`, the MBeans and Collectors for the `fs-1` metrics are deleted, and new MBeans and Collectors are created with the new names:

- Source Fs1 Rename Exceptions
- Source Fs1 Rename Queries
- Source Fs1 Rename Queries Total Results

Note that the metrics with the previous name remain on the Metrics tab because the data collected while the Source had this name remains valid and thus needs to be accessible. Therefore, it is possible to access metrics data for sources renamed months ago, i.e., until DDF is reinstalled or the metrics data is deleted from the `<DDF_HOME>/data/metrics` directory. Also note that the source metrics' names are modified to remove all non-alphanumeric characters and renamed in camelCase.

7.2. Metrics Reporting Application

The DDF Metrics Reporting Application provides access to historical data in several formats: a graphic, a comma-separated values file, a spreadsheet, a PowerPoint file, XML, and JSON formats for system metrics collected while DDF is running. Aggregate reports (weekly, monthly, and yearly) are also provided where all collected metrics are included in the report. Aggregate reports are available in Excel and PowerPoint formats.

To use the Metrics Reporting Application:

1. Navigate to the **Admin Console**.

2. Select the **Platform** Application.

3. Select the **Metrics** tab.

With each metric in the list, a set of hyperlinks is displayed under each column. Each column's header is displayed with the available time ranges. The time ranges currently supported are 15 minutes, 1 hour, 1 day, 1 week, 1 month, 3 months, 6 months, and 1 year, measured from the time that the hyperlink is clicked.

All metrics reports are generated by accessing the collected metric data stored in the `<DDF_HOME>/data/metrics` directory. All files in this directory are generated by the JmxCollector using RRD4J, a Round Robin Database for a Java open source product. All files in this directory will have the `.rrd` file extension and are binary files, hence they cannot be opened directly. These files should only be accessed using the Metrics tab's hyperlinks. There is one RRD file per metric being collected. Each RRD file is sized at creation time and will never increase in size as data is collected. One year's worth of metric data requires approximately 1 MB file storage.

WARNING	Do not remove the `<DDF_HOME>/data/metrics` directory or any files in the directory. If this is done, all existing metrics data will be permanently lost.
	Also note that if DDF is uninstalled/re-installed, all existing metrics data will be permanently lost.

Hyperlinks are provided for each metric and each format in which data can be displayed. For example, the PNG hyperlink for 15m for the Catalog Queries metric maps to https://{FQDN}:{PORT}/services/internal/metrics/catalogQueries.png?dateOffset=900, where the `dateOffset=900` indicates the previous 900 seconds (15 minutes) to be graphed.

Note that the date format will vary according to the regional/locale settings for the server.

All of the metric graphs displayed are in PNG format and are displayed on their own page. The user may use the back button in the browser to return to the Admin Console, or, when selecting the hyperlink for a graph, they can use the right mouse button in the browser to display the graph in a separate browser tab or window, which will keep the Admin Console displayed. The user can also specify custom time ranges by adjusting the URL used to access the metric's graph. The Catalog Queries metric data may also be graphed for a specific time range by specifying the `startDate` and `endDate` query parameters in the URL.

For example, to map the Catalog Queries metric data for March 31, 6:00 am, to April 1, 2013, 11:00 am, (Arizona timezone, which is -07:00) the URL would be:

```
https://{FQDN}:{PORT}/services/internal/metrics/catalogQueries.png?startDate=2013-03-
31T06:00:00-07:00&endDate=2013-04-01T11:00:00-07:00
```

Or to view the last 30 minutes of data for the Catalog Queries metric, a custom URL with a `dateOffset=1800` (30 minutes in seconds) could be used:

```
https://{FQDN}:{PORT}/services/internal/metrics/catalogQueries.png?dateOffset=1800
```

7.2.1. Metrics Aggregate Reports

The Metrics tab also provides aggregate reports for the collected metrics. These are reports that include data for all of the collected metrics for the specified time range.

The aggregate reports provided are:

- Weekly reports for each week up to the past four **complete** weeks from current time. A complete week is defined as a week from Monday through Sunday. For example, if current time is Thursday, April 11, 2013, the past complete week would be from April 1 through April 7.
- Monthly reports for each month up to the past 12 **complete** months from current time. A complete

month is defined as the full month(s) preceding current time. For example, if current time is Thursday, April 11, 2013, the past complete 12 months would be from April 2012 through March 2013.

- Yearly reports for the past **complete** year from current time. A complete year is defined as the full year preceding current time. For example, if current time is Thursday, April 11, 2013, the past complete year would be 2012.

An aggregate report in XLS format would consist of a single workbook (spreadsheet) with multiple worksheets in it, where a separate worksheet exists for each collected metric's data. Each worksheet would display:

- the metric's name and the time range of the collected data,
- two columns: Timestamp and Value, for each sample of the metric's data that was collected during the time range, and
- a total count (if applicable) at the bottom of the worksheet.

An aggregate report in PPT format would consist of a single slideshow with a separate slide for each collected metric's data. Each slide would display:

- a title with the metric's name.
- the PNG graph for the metric's collected data during the time range.
- a total count (if applicable) at the bottom of the slide.

Hyperlinks are provided for each aggregate report's time range in the supported display formats, which include Excel (XLS) and PowerPoint (PPT). Aggregate reports for custom time ranges can also be accessed directly via the URL:

```
https://{FQDN}:{PORT}/services/internal/metrics/report.<format>?startDate=<start_date_val
ue>&endDate=<end_date_value>
```

where `<format>` is either `xls` or `ppt` and the `<start_date_value>` and `<end_date_value>` specify the custom time range for the report.

These example reports represent custom aggregate reports. NOTE: all example URLs begin with https://{FQDN}:{PORT}, which is omitted in the table for brevity.

Table 20. Example Aggregate Reports

Description	URL
XLS aggregate report for March 15, 2013 to April 15, 2013	`/services/internal/metrics/report.xls?startDate=2013-03-15T12:00:00-07:00&endDate=2013-04-15T12:00:00-07:00`
XLS aggregate report for last 8 hours	`/services/internal/metrics/report.xls?dateOffset=28800`

Description	URL
PPT aggregate report for March 15, 2013 to April 15, 2013	`/services/internal/metrics/report.ppt?startDate=2013-03-15T12:00:00-07:00&endDate=2013-04-15T12:00:00-07:00`
PPT aggregate report for last 8 hours	`/services/internal/metrics/report.ppt?dateOffset=28800`

7.2.2. Viewing Metrics

The Metrics Viewer has reports in various formats.

1. Navigate to the **Admin Console**.

2. Select the **Platform** application.

3. Select the **Metrics** tab.

Reports are organized by timeframe and output format.

Standard time increments: * `15m`: 15 Minutes * `1h`: 1 Hour * `1d`: 1 Day * `1w`: 1 Week * `1M`: 1 Month * `3M`: 3 Month * `6M`: 6 Month * `1y`: 1 Year

Custom timeframes are also available via the selectors at the bottom of the page.

Output formats: * PNG * CSV (Comma-separated values) * XLS

NOTE	Based on the browser's configuration, either the `.xls` file will be downloaded or automatically displayed in Excel.

8. Action Framework

The Action Framework was designed as a way to limit dependencies between applications (apps) in a system. For instance, a feature in an app, such as an Atom feed generator, might want to include an external link as part of its feed's entries. That feature does not have to be coupled to a REST endpoint to work, nor does it have to depend on a specific implementation to get a link. In reality, the feature does not identify how the link is generated, but it does identify whether the link works or does not work when retrieving the intended entry's metadata. Instead of creating its own mechanism or adding an unrelated feature, it could use the Action Framework to query the OSGi container for any service that can provide a link. This does two things: it allows the feature to be independent of implementations, and it encourages reuse of common services.

The Action Framework consists of two major Java interfaces in its API:

1. `ddf.action.Action`

2. `ddf.action.ActionProvider`

Actions

Specific tasks that can be performed as services.

Action Providers

Lists of related actions that a service is capable of performing.

8.1. Action Providers

Included Action Providers

Download Resource ActionProvider

Downloads a resource to the local product cache.

IdP Logout Action Provider

Identity Provider Logout.

Karaf Logout Action

Local Logout.

LDAP Logout Action

Ldap Logout.

Overlay ActionProvider

Provides a metacard URL that transforms the metacard into a geographically aligned image (suitable for overlaying on a map).

View Metacard ActionProvider

Provides a URL to a metacard.

Metacard Transformer ActionProvider

Provides a URL to a metacard that has been transformed into a specified format.

9. Asynchronous Processing Framework

NOTE
: This code is experimental. While this interface is functional and tested, it may change or be removed in a future version of the library.

The **Asynchronous Processing Framework** is a way to run plugins asynchronously. Generally, plugins that take a significant amount of processing time and whose results are not immediately required are good candidates for being asynchronously processed. A **Processing Framework** can either be run on the local or remote system. Once the **Processing Framework** finishes processing incoming requests, it may submit `(Create|Update|Delete)Request`s to the Catalog. The type of plugins that a **Processing Framework** runs are the **Post-Process Plugins**. The **Post-Process Plugins** are triggered by the

Processing Post Ingest Plugin, which is a **Post-Ingest Plugin**. **Post-Ingest Plugins** are run after the metacard has been ingested into the Catalog. This feature is uninstalled by default.

WARNING	The **Processing Framework** does not support partial updates to the Catalog. This means that if any changes are made to a metacard in the Catalog between the time asynchronous processing starts and ends, those changes will be overwritten by the **ProcessingFramework** updates sent back to the Catalog. This feature should be used with caution.
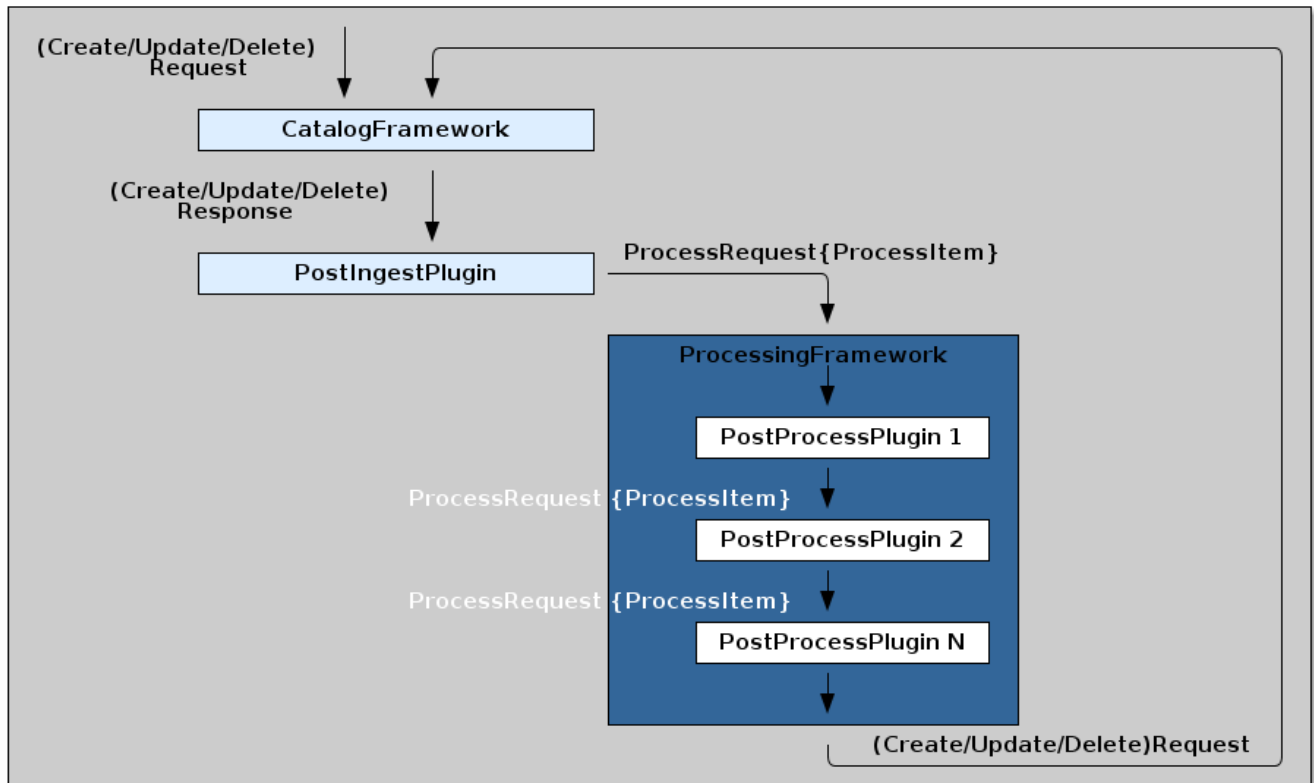
Processing Framework Architecture

The Asynchronous Processing Framework API Interfaces

1. `org.codice.ddf.catalog.async.processingframework.api.internal.ProcessingFramework`
2. `org.codice.ddf.catalog.async.plugin.api.internal.PostProcessPlugin`
3. `org.codice.ddf.catalog.async.data.api.internal.ProcessItem`
4. `org.codice.ddf.catalog.async.data.api.internal.ProcessCreateItem`
5. `org.codice.ddf.catalog.async.data.api.internal.ProcessUpdateItem`
6. `org.codice.ddf.catalog.async.data.api.internal.ProcessDeleteItem`
7. `org.codice.ddf.catalog.async.data.api.internal.ProcessRequest`
8. `org.codice.ddf.catalog.async.data.api.internal.ProcessResoure`
9. `org.codice.ddf.catalog.async.data.api.internal.ProcessResourceItem`
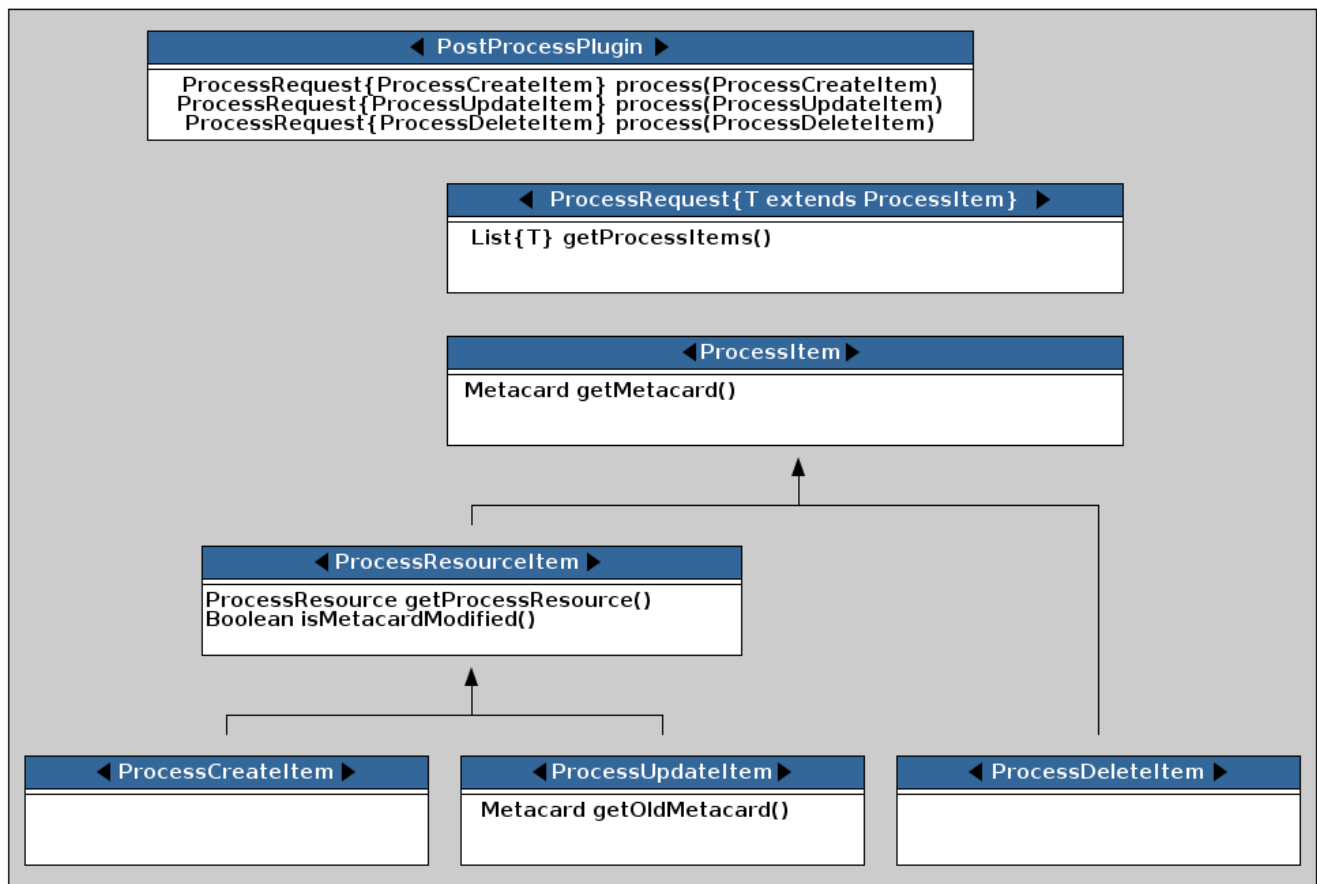
Processing Framework Interface Diagram

ProcessingFramework

The `ProcessingFramework` is responsible for processing incoming `ProcessRequest`s that contain a `ProcessItem`. A `ProcessingFramework` should never block. It receives its `ProcessRequest`s from a `PostIngestPlugin` on all CUD operations to the Catalog. In order to determine whether or not asynchronous processing is required by the `ProcessingFramework`, the `ProcessingFramework` should mark any request it has submitted back the Catalog, otherwise a processing loop may occur. For example, the default **In-Memory Processing Framework** adds a `POST_PROCESS_COMPLETE` flag to the Catalog CUD request after processing. This flag is checked by the `ProcessingPostIngestPlugin` before a `ProcessRequest` is sent to the `ProcessingFramework`. For an example of a `ProcessingFramework`, please refer to the `org.codice.ddf.catalog.async.processingframework.impl.InMemoryProcessingFramework`.

ProcessRequest

A `ProcessRequest` contains a list of `ProcessItem`s for the `ProcessingFramework` to process. Once a `ProcessRequest` has been processed by a `ProcessingFramework`, the `ProcessingFramework` should mark the `ProcessRequest` as already been processed, so that it does not process it again.

PostProcessPlugin

The `PostProcessPlugin` is a plugin that will be run by the `ProcessingFramework`. It is capable of processing `ProcessCreateItem`s, `ProcessUpdateItem`s, and `ProcessDeleteItem`s.

WARNING	*ProcessItem*
	Do not implement `ProcessItem` directly; it is intended for use only as a common base interface for `ProcessResourceItem` and `ProcessDeleteItem`.

The `ProcessItem` is contained by a `ProcessRequest`. It can be either a `ProcessCreateItem`, `ProcessUpdateItem`, or `ProcessDeleteItem`.

ProcessResource

The `ProcessResource` is a piece of content that is attached to a metacard. The piece of content can be either local or remote.

ProcessResourceItem

The `ProcessResourceItem` indicates that the item being processed may have a `ProcessResource` associated with it.

WARNING	*ProcessResourceItem Warning*
	Do not implement `ProcessResourceItem` directly; it is intended for use only as a common base interface for `ProcessCreateItem` and `ProcessUpdateItem`.

ProcessCreateItem

The `ProcessCreateItem` is an item for a metacard that has been created in the Catalog. It contains the created metacard and, optionally, a `ProcessResource`.

ProcessUpdateItem

The `ProcessUpdateItem` is an item for a metacard that has been updated in the Catalog. It contains the original metacard, the updated metacard and, optionally, a `ProcessResource`.

ProcessDeleteItem

The `ProcessDeleteItem` is an item for a metacard that has been deleted in the Catalog. It contains the deleted metacard.

10. Migration API

NOTE	This code is experimental. While the interfaces and classes provided are functional and tested, they may change or be removed in a future version of the library.

DDF currently has an experimental API for making bundles migratable. Interfaces and classes in `platform/migration/platform-migratable-api` are used by the system to identify bundles that provide implementations for export and import operations.

The migration API provides a mechanism for bundles to handle exporting data required to clone or

backup/restore a DDF system. The migration process is meant to be flexible, so an implementation of `org.codice.ddf.migration.Migratable` can handle exporting data for a single bundle or groups of bundles such as applications. For example, the `org.codice.ddf.platform.migratable.impl.PlatformMigratable` handles exporting core system files for the Platform application. Each migratable must provide a unique identifier via its `getId()` method used by the migration API to uniquely identify the migratable between exports and imports. DDF defines migratables of its own to export/import all configurations stored in `org.osgi.service.cm.ConfigurationAdmin`.

These do not need to be handled by implementations of `org.codice.ddf.migration.Migratable`.

An export and an import operation can be performed through the Command Console.

When an export operation is processed, the migration API will do a look-up for all registered OSGi services that are implementing `Migratable` and call their `doExport()` method. As part of the exported data, information about the migratable as required by the `org.codice.ddf.platform.services.common.Describable` interface will be included. In particular the version string returned will help the migration API identify the version of the exported data from the corresponding migratable and must be provided as a non-blank string.

When an import operation is processed, the migration API will do another look-up for all registered OSGi services that are implementing `Migratable` and call their `doImport()` or `doIncompatibleImport()` methods based on whether the recorded version string at export time is equal to the version string currently provided by the migratable or not. The `doMissingImport()` method will be called instead of one of the other two methods when the migration API detects that the corresponding migratable data is missing from the exported data. Any migratables that are tagged using the `OptionalMigratable` tag interface will automatically be skipped unless otherwise specified when the import phase is initiated.

The services that implement the migratable interface will be called one at a time based on their service ranking order, and do not need to be thread safe. A bundle or a feature can have as many services implementing the interfaces as needed.

10.1. The Migration API Interfaces and Classes

1. `org.codice.ddf.migration.Migratable`
2. `org.codice.ddf.migration.OptionalMigratable`
3. `org.codice.ddf.migration.MigrationContext`
4. `org.codice.ddf.migration.ExportMigrationContext`
5. `org.codice.ddf.migration.ImportMigrationContext`
6. `org.codice.ddf.migration.MigrationEntry`
7. `org.codice.ddf.migration.ExportMigrationEntry`
8. `org.codice.ddf.migration.ImportMigrationEntry`
9. `org.codice.ddf.migration.MigrationOperation`
10. `org.codice.ddf.migration.MigrationReport`

11. `org.codice.ddf.migration.MigrationMessage`
12. `org.codice.ddf.migration.MigrationException`
13. `org.codice.ddf.migration.MigrationWarning`
14. `org.codice.ddf.migration.MigrationInformation`
15. `org.codice.ddf.migration.MigrationSuccessfulInformation`

10.1.1. `Migratable`

The contract for a migratable is stored here. This is the only interface that should be implemented by implementers and registered as an OSGi service. All other interfaces will be implemented by the migration API that provides support for migratables.

The `org.codice.ddf.migration.Migratable` interface defines these methods:

- `String getId()`
- `String getVersion()`
- `String getTitle()`
- `String getDescription()`
- `String getOrganization()`
- `void doExport(ExportMigrationContext context)`
- `void doImport(ImportMigrationContext context)`
- `void doIncompatibleImport(ImportMigrationContext context)`
- `void doMissingImport(ImportMigrationContext context)`

The `getId()` method returns a unique identifier for this migratable that must remain constant between the export and the import operations in order for the migration API to correlate the exported data with the migratable during the import operation. It must be unique across all migratables.
The `getVersion()` method returns a unique version string which is meant to identify the version of the data exported or supported at import time by the migratable. It cannot be blank and its format is left to the migratable. The only noticeable requirement is that when the string compares equal using the `String.equals()` method, the migration API will call `doImport()` instead of `doIncompatibleImport()` to restore previously exported data for the migratable.
The `getTitle()` method returns a simple title for the migratable.
The `getDescription()` method returns a short description of the type of data exported by the migratable.
The `getOrganization()` method provides the name of the organization responsible for the migratable.
The `doExport()` method is called by the migration API along with a context for the current export operation to store data.
The `doImport()` method is called by the migration API along with a context for the current import operation when the version of exported data matches the current version reported by the migratable. This method can be used to restore previously exported data.
The `doIncompatibleImport()` method is called to restore incompatible data which might require transformation. It is provided a context for the current import operation and the previously exported version. It can then proceed with restoring incompatible data which might require transformation.

Finally, the `doMissingImport()` method will be called along with the context for the current import operation when data had not been exported for the corresponding migratable. This will be the case when a migratable is later introduced in the software distribution.

In order to create a `Migratable` for a module of the system, the `org.codice.ddf.migration.Migratable` interface must be implemented and the implementation must be registered under the `org.codice.ddf.migration.Migratable` interface as an OSGi service in the OSGi service registry. Creating an OSGi service allows for the migration API to lookup all implementations of `org.codice.ddf.migration.Migratable` and command them to export or import.

10.1.2. `OptionalMigratable`

This interface is designed as a tagged interface to identify optional migratables. An optional migratable will be skipped by default during the import phase. It can still be manually marked as mandatory when initiating the import phase.

10.1.3. `MigrationContext`

The `org.codice.ddf.migration.MigrationContext` provides contextual information about an operation in progress for a given migratable. This is a sort of sandbox that is unique to each migratable. This interface defines the following methods:

- `MigrationReport getReport()`
- `String getId()`

The `getReport()` method returns a migration report that can be used to record messages while processing an export or an import operation.
The `getId()` method returns the identifier for the currently processing migratable.

10.1.4. `ExportMigrationContext`

The export migration context provides methods for creating new migration entries and system property referenced migration entries to track exported migration files for a given migratable while processing an export migration operation. It defines the following methods:

- `Optional<ExportMigrationEntry> getSystemPropertyReferencedEntry(String name)`
- `Optional<ExportMigrationEntry> getSystemPropertyReferencedEntry(String name, BiPredicate<MigrationReport, String> validator)`
- `ExportMigrationEntry getEntry(Path path)`
- `Stream<ExportMigrationEntry> entries(Path path)`
- `Stream<ExportMigrationEntry> entries(Path path, PathMatcher filter)`
- `Stream<ExportMigrationEntry> entries(Path path, boolean recurse)`
- `Stream<ExportMigrationEntry> entries(Path path, boolean recurse, PathMatcher filter)`

The `getSystemPropertyReferencedEntry()` methods create a migration entry to track a file referenced by a given system property value.

The `getEntry()` method creates a migration entry given the path for a specific file or directory.
The `entries()` methods create multiple entries corresponding to all files recursively (or not) located underneath a given path with an optional path matcher to filter which files to create entries for.

Once an entry is created, it is not stored with the exported data. It is the migratable's responsibility to store the data using one of the entry's provided methods. Entries are uniquely identified using a relative path and are specific to each migratable meaning that an entry with the same path in two migratables will not conflict with each other. Each migratable is given its own context (a.k.a. sandbox) to work with.

10.1.5. `ImportMigrationContext`

The import migration context provides methods for retrieving migration entries and system property referenced migration entries corresponding to exported files for a given migratable while processing an import migration operation. It defines the following methods:

- `Optional<ImportMigrationEntry> getSystemPropertyReferencedEntry(String name)`
- `ImportMigrationEntry getEntry(Path path)`
- `Stream<ImportMigrationEntry> entries(Path path)`
- `Stream<ImportMigrationEntry> entries(Path path, PathMatcher filter)`

The `getSystemPropertyReferencedEntry()` method retrieves a migration entry for a file that was referenced by a given system property value.
The `getEntry()` method retrieves a migration entry given the path for a specific file or directory.
The `entries()` methods retrieve multiple entries corresponding to all exported files recursively located underneath a given relative path with an optional path matcher to filter which files to retreive entries for.

Once an entry is retrieved, its exported data is not restored. It is the migratable's responsibility to restore the data using one of the entry's provided methods. Entries are uniquely identified using a relative path and are specific to each migratable meaning that an entry with the same path in two migratables will not conflict with each other. Each migratable is given its own context (a.k.a. sandbox) to work with.

10.1.6. `MigrationEntry`

This interface provides supports for exported files. It defines the following methods:

- `MigrationReport getReport()`
- `String getId()`
- `String getName()`
- `Path getPath()`
- `boolean isDirectory()`
- `boolean isFile()`
- `long getLastModifiedTime()`

The `getReport()` method provides access to the associated migration report where messages can be recorded.

The `getId()` method returns the identifier for the migratable responsible for this entry.

The `getName()` method provides the unique name for this entry in an OS-independent way.

The `getPath()` method provides the unique path to the corresponding file for this entry in an OS-specific way.

The `isDirectory()` method indicates if the entry represents a directory. The `isFile()` method indicates if the entry represents a file. The `getLastModifiedTime()` method provides the last modification time for the corresponding file or directory as available when the file or directory is exported.

10.1.7. `ExportMigrationEntry`

The export migration entry provides additional methods available for entries created at export time. It defines the following methods:

- `Optional<ExportMigrationEntry> getPropertyReferencedEntry(String name)`
- `Optional<ExportMigrationEntry> getPropertyReferencedEntry(String name, BiPredicate<MigrationReport, String> validator)`
- `boolean store()`
- `boolean store(boolean required)`
- `boolean store(PathMatcher filter)`
- `boolean store(boolean required, PathMatcher filter)`
- `boolean store(BiThrowingConsumer<MigrationReport, OutputStream, IOException> consumer)`
- `OutputStream getOutputStream() throws IOException`

The `getPropertyReferencedEntry()` methods create another migration entry for a file that was referenced by a given property value in the file represented by this entry.

The `store()` and `store(boolean required)` methods will automatically copy the content of the corresponding file as part of the export making sure the file exists (if required) on disk otherwise an error will be recorded. If the path represents a directory then all files recursively found under the path will be automatically exported.

The `store(PathMatcher filter)` and `store(boolean required, PathMatcher filter)` methods will automatically copy the content of the corresponding file if it matches the filter as part of the export making sure the file exists (if required) on disk otherwise an error will be recorded. If the path represents a directory then all matching files recursively found under the path will be automatically exported.

The `store(BiThrowingConsumer<MigrationReport, OutputStream, IOException> consumer)` method allows the migratable to control the export process by specifying a callback consumer that will be called back with an output stream where the data can be writen to instead of having a file on disk being copied by the migration API. The `OutputStream getOutputStream()` method provides access to the low-level output stream where the migratable can write data directly as opposed to having a file on disk copied automatically.

10.1.8. `ImportMigrationEntry`

The import migration entry provides additional methods available for entries retrieved at import time. It defines the following methods:

- `Optional<ImportMigrationEntry> getPropertyReferencedEntry(String name)`
- `boolean restore()`
- `boolean restore(boolean required)`
- `boolean restore(PathMatcher filter)`
- `boolean restore(boolean required, PathMatcher filter)`
- `boolean restore(BiThrowingConsumer<MigrationReport, Optional<InputStream>, IOException> consumer)`
- `Optional<InputStream getInputStream() throws IOException`

The `getPropertyReferencedEntry()` method retrieves another migration entry for a file that was referenced by a given property value in the file represented by this entry.

The `restore()` and `restore(boolean required)` methods will automatically copy the exported content of the corresponding file back to disk if it was exported; otherwise an error will be recorded. If the path represents a directory then all file entries originally recursively exported under this entry's path will be automatically imported. If the directory had been completely exported using one of the `store()` or `store(boolean required)` methods then in addition to restoring all entries recursively, calling this method will also remove any existing files or directories that were not on the original system.

The `restore(PathMatcher filter)` and `restore(boolean required, PathMatcher filter)` methods will automatically copy the exported content of the corresponding file if it matches the filter back to disk if it was exported; otherwise an error will be recorded. If the path represents a directory then all matching file entries originally recursively exported under this entry's path will be automatically imported.

The `restore(BiThrowingConsumer<MigrationReport, Optional<InputStream>, IOException> consumer)` method allows the migratable to control the import process by specifying a callback consumer that will be called back with an optional input stream (empty if the data was not exported) where the data can be read from instead of having a file on disk being created or updated by the migration API.

The `Optional<InputStream> getInputStream()` method provides access to the optional low-level input stream (empty if the data was not exported) where the migratable can read data directly as opposed to having a file on disk created or updated automatically.

10.1.9. `MigrationOperation`

The `org.codice.ddf.migration.MigrationOperation` provides a simple enumeration for identifying the various migration operations available.

10.1.10. `MigrationReport`

The `org.codice.ddf.migration.MigrationReport` interface provides information about the execution of a migration operation. It defines the following methods:

- `MigrationOperation getOperation()`
- `Instant getStartTime()`
- `Optional<Instant> getEndTime()`
- `MigrationReport record(String msg)`
- `MigrationReport record(String format, @Nullable Object… args)`
- `MigrationReport record(MigrationMessage msg)`
- `MigrationReport doAfterCompletion(Consumer<MigrationReport> code)`
- `Stream<MigrationMessage> messages()`
- `default Stream<MigrationException> errors()`
- `Stream<MigrationWarning> warnings()`
- `Stream<MigrationInformation> infos()`
- `boolean wasSuccessful()`
- `boolean wasSuccessful(@Nullable Runnable code)`
- `boolean wasIOSuccessful(@Nullable ThrowingRunnable<IOException> code) throws IOException`
- `boolean hasInfos()`
- `boolean hasWarnings()`
- `boolean hasErrors()`
- `void verifyCompletion()`

The `getOperation()` method provides the type of migration operation (i.e. export or import) currently in progress.

The `getStartTime()` method provides the time at which the corresponding operation started.

The `getEndTime()` method provides the optional time at which the corresponding operation ended. The time is only available if the operation has ended.

The `record()` methods enable messages to be recorded with the report. Messages are displayed on the console for the administrator.

The `doAfterCompletion()` methods enable code to be registered such that it is invoked at the end before a successful result is returned. Such code can still affect the result of the operation.

The `messages()` method provides access to all recorded messages so far.

The `errors()` method provides access to all recorded error messages so far.

The `warnings()` method provides access to all recorded warning messages so far.

The `infos()` method provides access to all recorded informational messages so far.

The `wasSuccessful()` method provides a quick check to see if the report is successful. A successful report might have warnings recorded but cannot have errors recorded.

The `wasSuccessful(Runnable code) method allows code to be executed. It will return true if no new errors are recorded as a result of executing the provided code.`

The `wasIOSuccessful(ThrowingRunnable<IOException> code) method allows code to be executed which can throw I/O exceptions which are automatically recorded as errors. It will return true if no new errors are recorded as a result of executing the provided code.`

The `hasInfos() method will return true if at least one information message has been recorded so far.

The `hasWarnings()` method will return true if at least one warning message has been recorded so far.

The `hasErrors()` method will return true if at least one error message has been recorded so far.

The `verifyCompletion() method will verify if the report is successful and if not, it will throw back the

first recorded exception and attach as suppressed exceptions all other recorded exceptions.

10.1.11. `MigrationMessage`

The `org.codice.ddf.migration.MigrationException` is defined as a base class for all recordable messages during migration operations. It defines the following methods:

- `String getMessage()`

The `getMessage()` method provides a message for the corresponding exception, warning, or info that will be displayed to the administrator on the console.

10.1.12. `MigrationException`

An `org.codice.ddf.migration.MigrationException` should be thrown when an unrecoverable exception occurs that prevents the export or the import operation from continuing. It is also possible to simply record one or many exception(s) with the migration report in order to fail the export or import operation while not aborting it right away. This provides for the ability to record as many errors as possible and report all of them back to the administrator. All migration exception messages are displayed to the administrator.

10.1.13. `MigrationWarning`

An `org.codice.ddf.migration.MigrationWarning` should be used when a migratable wants to warn the administrator that certain aspects of the export or the import may cause problems. For example, if an absolute path is encountered, that path may not exist on the target system and cause the installation to fail. All migration warning messages are displayed to the administrator.

10.1.14. `MigrationInformation`

An `org.codice.ddf.migration.MigrationInformation` should be used when a migratable simply wants to provide useful information to the administrator. All migration information messages are displayed to the administrator.

10.1.15. `MigrationSuccessfulInformation`

The `org.codice.ddf.migration.MigrationSuccessfulInformation` can be used to further qualify an information message as representing the success of an operation.

11. Security Framework

The DDF Security Framework utilizes Apache Shiro as the underlying security framework. The classes mentioned in this section will have their full package name listed, to make it easy to tell which classes come with the core Shiro framework and which are added by DDF.

11.1. Subject

`ddf.security.Subject <extends> org.apache.shiro.subject.Subject`

The Subject is the key object in the security framework. Most of the workflow and implementations revolve around creating and using a Subject. The Subject object in DDF is a class that encapsulates all information about the user performing the current operation. The Subject can also be used to perform permission checks to see if the calling user has acceptable permission to perform a certain action (e.g., calling a service or returning a metacard). This class was made DDF-specific because the Shiro interface cannot be added to the Query Request property map.

Table 21. Implementations of Subject:

Classname	Description
ddf.security.impl.SubjectImpl	Extends `org.apache.shiro.subject.support.DelegatingSubject`

11.1.1. Security Manager

`ddf.security.service.SecurityManager`

The Security Manager is a service that handles the creation of Subject objects. A proxy to this service should be obtained by an endpoint to create a Subject and add it to the outgoing `QueryRequest`. The Shiro framework relies on creating the subject by obtaining it from the current thread. Due to the multi-threaded and stateless nature of the DDF framework, utilizing the Security Manager interface makes retrieving Subjects easier and safer.

Table 22. Implementations of Security Managers:

Classname	Description
`ddf.security.service.SecurityManagerImpl`	This implementation of the Security Manager handles taking in both `org.apache.shiro.authc.` AuthenticationToken and `org.apache.cxf.ws.security.tokenstore.SecurityToken` objects.

11.1.2. Realms

DDF uses Apache Shiro for the concept of Realms for Authentication and Authorization. Realms are components that access security data such as such as users or permissions.

11.1.2.1. Authenticating Realms

`org.apache.shiro.realm.AuthenticatingRealm`

Authenticating Realms are used to authenticate an incoming authentication token and create a Subject on successful authentication. A Subject is an application user and all available security-relevant

information about that user.

Table 23. Implementations of Authenticating Realms in DDF:

Classname	Description
`ddf.security.realm.sts.StsRealm`	This realm delegates authentication to the Secure Token Service (STS). It creates a `RequestSecurityToken` message from the incoming Authentication Token and converts a successful STS response into a Subject.

11.1.2.2. Authorizing Realms

`org.apache.shiro.realm.AuthorizingRealm`

Authorizing Realms are used to perform authorization on the current Subject. These are used when performing both service authorization and filtering. They are passed in the `AuthorizationInfo` of the Subject along with the permissions of the object wanting to be accessed. The response from these realms is a true (if the Subject has permission to access) or false (if the Subject does not).

Table 24. Other implementations of the Security API within DDF

Classname	Description
`org.codice.ddf.platform.filter.delegate.DelegateServletFilter`	The `DelegateServletFilter` detects any servlet filters that have been exposed as OSGi services implementing `org.codice.ddf.platform.filter.SecurityFilter` and places them in-order in front of any servlet or web application running on the container.
`org.codice.ddf.security.filter.websso.WebSSOFilter`	This filter is the main security filter that works with a number of handlers to protect a variety of web contexts, each using different authentication schemes and policies.
`org.codice.ddf.security.handler.saml.SAMLAssertionHandler`	This handler is executed by the WebSSOFilter for any contexts configured to use it. This handler should always come first when configured in the Web Context Policy Manager, as it provides a caching capability to web contexts that use it. The handler will first check for the existence of an HTTP Authorization header of type SAML, whose value is a Base64 + deflate SAML assertion. If that is not found, then the handler will check for the existence of the deprecated `org.codice.websso.saml.token` cookie with the same value. Failing that, it will check for a JSESSIONID cookie to use as a reference to a cached assertion. If the JSESSIONID is valid, the `SecurityToken` will be retrieved from the cache.
`org.codice.ddf.security.handler.basic.BasicAuthenticationHandler`	Checks for basic authentication credentials in the http request header. If they exist, they are retrieved and passed to the `LoginFilter` for exchange.
`org.codice.ddf.security.handler.pki.PKIHandler`	Handler for PKI based authentication. X509 chain will be extracted from the HTTP request and converted to a `BinarySecurityToken`.

Classname	Description
`org.codice.ddf.security.handler.guest.GuestHandler`	Handler that allows guest user access via a guest user account. The guest account credentials are configured via the org.codice.ddf.security.claims.guest.GuestClaimsHandler. The `GuestHandler` also checks for the existence of basic auth credentials or PKI credentials that might be able to override the use of the guest user.
`org.codice.ddf.security.filter.login.LoginFilter`	This filter runs immediately after the WebSSOFilter and exchanges any authentication information found in the request with a Subject via Shiro.
`org.codice.ddf.security.filter.authorization.AuthorizationFilter`	This filter runs immediately after the `LoginFilter` and checks any permissions assigned to the web context against the attributes of the user via Shiro.
`org.apache.shiro.realm.AuthenticatingRealm`	This is an abstract authenticating realm that exchanges an `org.apache.shiro.authc.AuthenticationToken` for a `ddf.security.Subject` in the form of an `org.apache.shiro.authc.AuthenticationInfo`
`ddf.security.realm.sts.StsRealm`	This realm is an implementation of `org.apache.shiro.realm.AuthenticatingRealm` and connects to an STS (configurable) to exchange the authentication token for a Subject.
`ddf.security.service.AbstractAuthorizingRealm`	This is an abstract authorizing realm that takes care of caching and parsing the Subject's `AuthorizingInfo` and should be extended to allow the implementing realm to focus on making the decision.
`ddf.security.pdp.realm.AuthZRealm`	This realm performs the authorization decision and may or may not delegate out to the external XACML processing engine. It uses the incoming permissions to create a decision. However, it is possible to extend this realm using the ddf.security.policy.extension.PolicyExtension interface. This interface allows an integrator to add additional policy information to the PDP that can't be covered via its generic matching policies. This approach is often easier to configure for those that are not familiar with XACML.
`org.codice.ddf.security.validator.*`	A number of STS validators are provided for X.509 (BinarySecurityToken), UsernameToken, SAML Assertion, and DDF custom tokens. The DDF custom tokens are all `BinarySecurityTokens` that may have PKI or username/password information as well as an authentication realm (correlates to JAAS realms installed in the container). The authentication realm allows an administrator to restrict which services they wish to use to authenticate users. For example: installing the `security-sts-ldaplogin` feature will enable a JAAS realm with the name "ldap". This realm can then be specified on any context using the Web Context Policy Manager. That realm selection is then passed via the token sent to the STS to determine which validator to use.

NOTE	Using the SAML Web SSO Identity Provider for authentication will ignore any realm settings and simply use all configured JAAS realms.

WARNING	An update was made to the SAML Assertion Handler to pass SAML assertions through the Authorization HTTP header. Cookies *are* still accepted and processed to maintain legacy federation compatibility, but assertions are sent in the header on outbound requests. While a machine's identity will still federate between versions, a user's identity will ONLY be federated when a DDF version 2.7.x server communicates with a DDF version 2.8.x+ server, or between two servers whose versions are 2.8.x or higher.

11.2. Security Core

The Security Core application contains all of the necessary components that are used to perform security operations (authentication, authorization, and auditing) required in the framework.

11.2.1. Security Core API

The Security Core API contains all of the DDF APIs that are used to perform security operations within DDF.

11.2.1.1. Installing the Security Core API

The Security Services App installs the Security Core API by default. Do not uninstall the Security Core API as it is integral to system function and all of the other security services depend upon it.

11.2.1.2. Configuring the Security Core API

The Security Core API has no configurable properties.

11.2.2. Security Core Implementation

The Security Core Implementation contains the reference implementations for the Security Core API interfaces that come with the DDF distribution.

11.2.2.1. Installing the Security Core Implementation

The Security Core app installs this bundle by default. It is recommended to use this bundle as it contains the reference implementations for many classes used within the Security Framework.

11.2.2.2. Configuring the Security Core Implementation

The Security Core Implementation has no configurable properties.

11.2.3. Security Core Commons

The Security Core Commons bundle contains helper and utility classes that are used within DDF to help with performing common security operations. Most notably, this bundle contains the `ddf.security.common.audit.SecurityLogger` class that performs the security audit logging within DDF.

11.2.3.1. Configuring the Security Core Commons

The Security Core Commons bundle has no configurable properties.

11.3. Security Encryption

The Security Encryption application offers an encryption framework and service implementation for other applications to use. This service is commonly used to encrypt and decrypt default passwords that are located within the metatype and Admin Console.

The encryption service and encryption command, which are based on Tink ⧉, provide an easy way for developers to add encryption capabilities to DDF.

11.3.1. Security Encryption API

The Security Encryption API bundle provides the framework for the encryption service. Applications that use the encryption service should use the interfaces defined within it instead of calling an implementation directly.

11.3.1.1. Installing Security Encryption API

This bundle is installed by default as part of the `security-encryption` feature. Many applications that come with DDF depend on this bundle and it should not be uninstalled.

11.3.1.2. Configuring the Security Encryption API

The Security Encryption API has no configurable properties.

11.3.2. Security Encryption Implementation

The Security Encryption Implementation bundle contains all of the service implementations for the Encryption Framework and exports those implementations as services to the OSGi service registry.

11.3.2.1. Installing Security Encryption Implementation

This bundle is installed by default as part of the `security-encryption` feature. Other projects are dependent on the services this bundle exports and it should not be uninstalled unless another security service implementation is being added.

11.3.2.2. Configuring Security Encryption Implementation

The Security Encryption Implementation has no configurable properties.

11.3.3. Security Encryption Commands

The Security Encryption Commands bundle enhances the DDF system console by allowing administrators and integrators to encrypt and decrypt values directly from the console.

The `security:encrypt` command allows plain text to be encrypted using HMAC + AES for encryption with a randomly generated key that is created when the system is installed. This is useful when displaying password fields in a GUI.

Below is an example of the security:encrypt command used to encrypt the plain text "myPasswordToEncrypt". The output, `bR9mJpDVo8bTRwqGwIFxHJ5yFJzatKwjXjIo/8USWm8=`, is the encrypted value.

```
ddf@local>security:encrypt myPasswordToEncrypt

bR9mJpDVo8bTRwqGwIFxHJ5yFJzatKwjXjIo/8USWm8=
```

11.3.3.1. Installing the Security Encryption Commands

This bundle is installed by default with the `security-encryption` feature. This bundle is tied specifically to the DDF console and can be uninstalled if not needed. When uninstalled, however, administrators will not be able to encrypt and decrypt data from the console.

11.3.3.2. Configuring the Security Encryption Commands

The Security Encryption Commands have no configurable properties.

11.4. Security LDAP

The DDF LDAP application allows the user to configure either an embedded or a standalone LDAP server. The provided features contain a default set of schemas and users loaded to help facilitate authentication and authorization testing.

11.4.1. Embedded LDAP Server

DDF includes an embedded LDAP server (OpenDJ) for testing and demonstration purposes.

| WARNING | The embedded LDAP server is intended for testing purposes only and is not recommended for production use. |

11.4.1.1. Installing the Embedded LDAP Server

The embedded LDAP server is not installed by default with a standard installation.

1. Navigate to the **Admin Console**.

2. Select the **System** tab.

3. Select the **Features** tab.

4. Install the `opendj-embedded` feature.

11.4.1.2. Configuring the Embedded LDAP

Configure the Embedded LDAP from the Admin Console:

1. Navigate to the **Admin Console**.

2. Select the **OpenDj Embedded** application.

3. Select the **Configuration** tab.

Table 25. OpenDJ Embedded Configurable Properties

Configuration Name	Description
LDAP Port	Sets the port for LDAP (plaintext and startTLS). 0 will disable the port.
LDAPS Port	Sets the port for LDAPS. 0 will disable the port.
Base LDIF File	Location on the server for a LDIF file. This file will be loaded into the LDAP and overwrite any existing entries. This option should be used when updating the default groups/users with a new LDIF file for testing. The LDIF file being loaded may contain any LDAP entries (schemas, users, groups, etc.). If the location is left blank, the default base LDIF file will be used that comes with DDF.

11.4.1.3. Connecting to Standalone LDAP Servers

DDF instances can connect to external LDAP servers by installing and configuring the `security-sts-ldaplogin` and `security-sts-ldapclaimshandler` features detailed here.

In order to connect to more than one LDAP server, configure these features for each LDAP server.

11.4.1.4. Embedded LDAP Configuration

The Embedded LDAP application contains an LDAP server (OpenDJ version 2.6.2) that has a default set of schemas and users loaded to help facilitate authentication and authorization testing.

Table 26. Embedded LDAP Default Ports Settings

Protocol	Default Port
LDAP	1389

Protocol	Default Port
LDAPS	1636
StartTLS	1389

Table 27. Embedded LDAP Default Users

Username	Password	Groups	Description
testuser1	password1		General test user for authentication
testuser2	password2		General test user for authentication
nromanova	password1	avengers	General test user for authentication
lcage	password1	admin, avengers	General test user for authentication, Admin user for karaf
jhowlett	password1	admin, avengers	General test user for authentication, Admin user for karaf
pparker	password1	admin, avengers	General test user for authentication, Admin user for karaf
jdrew	password1	admin, avengers	General test user for authentication, Admin user for karaf
tstark	password1	admin, avengers	General test user for authentication, Admin user for karaf
bbanner	password1	admin, avengers	General test user for authentication, Admin user for karaf
srogers	password1	admin, avengers	General test user for authentication, Admin user for karaf
admin	admin	admin	Admin user for karaf

Table 28. Embedded LDAP Default Admin User Settings

Username	Password	Groups	Attributes	Description
admin	secret			Administrative User for LDAP

11.4.1.5. Schemas

The default schemas loaded into the LDAP instance are the same defaults that come with OpenDJ.

Table 29. Embedded LDAP Default Schemas

Schema File Name	Schema Description [↗]
`00-core.ldif`	This file contains a core set of attribute type and objectclass definitions from several standard LDAP documents, including `draft-ietf-boreham-numsubordinates`, `draft-findlay-ldap-groupofentries`, `draft-furuseth-ldap-untypedobject`, `draft-good-ldap-changelog`, `draft-ietf-ldup-subentry`, `draft-wahl-ldap-adminaddr`, RFC 1274, RFC 2079, RFC 2256, RFC 2798, RFC 3045, RFC 3296, RFC 3671, RFC 3672, RFC 4512, RFC 4519, RFC 4523, RFC 4524, RFC 4530, RFC 5020, and X.501.
`01-pwpolicy.ldif`	This file contains schema definitions from `draft-behera-ldap-password-policy`, which defines a mechanism for storing password policy information in an LDAP directory server.
`02-config.ldif`	This file contains the attribute type and `objectclass` definitions for use with the directory server configuration.
`03-changelog.ldif`	This file contains schema definitions from `draft-good-ldap-changelog`, which defines a mechanism for storing information about changes to directory server data.
`03-rfc2713.ldif`	This file contains schema definitions from RFC 2713, which defines a mechanism for storing serialized Java objects in the directory server.
`03-rfc2714.ldif`	This file contains schema definitions from RFC 2714, which defines a mechanism for storing CORBA objects in the directory server.
`03-rfc2739.ldif`	This file contains schema definitions from RFC 2739, which defines a mechanism for storing calendar and vCard objects in the directory server. Note that the definition in RFC 2739 contains a number of errors, and this schema file has been altered from the standard definition in order to fix a number of those problems.
`03-rfc2926.ldif`	This file contains schema definitions from RFC 2926, which defines a mechanism for mapping between Service Location Protocol (SLP) advertisements and LDAP.
`03-rfc3112.ldif`	This file contains schema definitions from RFC 3112, which defines the authentication password schema.
`03-rfc3712.ldif`	This file contains schema definitions from RFC 3712, which defines a mechanism for storing printer information in the directory server.
`03-uddiv3.ldif`	This file contains schema definitions from RFC 4403, which defines a mechanism for storing UDDIv3 information in the directory server.
`04-rfc2307bis.ldif`	This file contains schema definitions from the `draft-howard-rfc2307bis` specification, used to store naming service information in the directory server.
`05-rfc4876.ldif`	This file contains schema definitions from RFC 4876, which defines a schema for storing Directory User Agent (DUA) profiles and preferences in the directory server.
`05-samba.ldif`	This file contains schema definitions required when storing Samba user accounts in the directory server.
`05-solaris.ldif`	This file contains schema definitions required for Solaris and OpenSolaris LDAP naming services.

Schema File Name	Schema Description 🗗
`06-compat.ldif`	This file contains the attribute type and `objectclass` definitions for use with the directory server configuration.

11.4.1.6. Starting and Stopping the Embedded LDAP

The embedded LDAP application installs a feature with the name `ldap-embedded`. Installing and uninstalling this feature will start and stop the embedded LDAP server. This will also install a fresh instance of the server each time. If changes need to persist, stop then start the `embedded-ldap-opendj` bundle (rather than installing/uninstalling the feature).

All settings, configurations, and changes made to the embedded LDAP instances are persisted across DDF restarts. If DDF is stopped while the LDAP feature is installed and started, it will automatically restart with the saved settings on the next DDF start.

11.4.1.7. Limitations of the Embedded LDAP

Current limitations for the embedded LDAP instances include:

- Inability to store the LDAP files/storage outside of the DDF installation directory. This results in any LDAP data (i.e., LDAP user information) being lost when the `ldap-embedded` feature is uninstalled.
- Cannot be run standalone from DDF. In order to run `embedded-ldap`, the DDF must be started.

11.4.1.8. External Links for the Embedded LDAP

Location to the default base LDIF file in the DDF source code 🗗.

OpenDJ documentation 🗗

11.4.1.9. LDAP Administration

OpenDJ provides a number of tools for LDAP administration. Refer to the OpenDJ Admin Guide 🗗.

11.4.1.10. Downloading the Admin Tools

Download OpenDJ (Version 2.6.4) 🗗 and the included tool suite.

11.4.1.11. Using the Admin Tools

The admin tools are located in `<opendj-installation>/bat` for Windows and `<opendj-installation>/bin` for `nix. These tools can be used to administer both local and remote LDAP servers by setting the *host` and **port** parameters appropriately.

In this example, the user **Bruce Banner (uid=bbanner)** is disabled using the **manage-account** command on Windows. Run **manage-account --help** for usage instructions.

Example Commands for Disabling/Enabling a User's Account

```
D:\OpenDJ-2.4.6\bat>manage-account set-account-is-disabled -h localhost -p 4444 -O true
-D "cn=admin" -w secret -b "uid=bbanner,ou=users,dc=example,dc=com"
The server is using the following certificate:
    Subject DN:  CN=Win7-1, O=Administration Connector Self-Signed Certificate
    Issuer DN:  CN=Win7-1, O=Administration Connector Self-Signed Certificate
    Validity:  Wed Sep 04 15:36:46 MST 2013 through Fri Sep 04 15:36:46 MST 2015
Do you wish to trust this certificate and continue connecting to the server?
Please enter "yes" or "no":yes
Account Is Disabled:  true
```

Notice `Account Is Disabled: true` in the listing:

Verifying an Account is Disabled

```
D:\OpenDJ-2.4.6\bat>manage-account get-all -h localhost -p 4444  -D "cn=admin" -w secret
-b "uid=bbanner,ou=users,dc=example,dc=com"
The server is using the following certificate:
    Subject DN:  CN=Win7-1, O=Administration Connector Self-Signed Certificate
    Issuer DN:  CN=Win7-1, O=Administration Connector Self-Signed Certificate
    Validity:  Wed Sep 04 15:36:46 MST 2013 through Fri Sep 04 15:36:46 MST 2015
Do you wish to trust this certificate and continue connecting to the server?
Please enter "yes" or "no":yes
Password Policy DN:  cn=Default Password Policy,cn=Password Policies,cn=config
Account Is Disabled:  true
Account Expiration Time:
Seconds Until Account Expiration:
Password Changed Time:  19700101000000.000Z
Password Expiration Warned Time:
Seconds Until Password Expiration:
Seconds Until Password Expiration Warning:
Authentication Failure Times:
Seconds Until Authentication Failure Unlock:
Remaining Authentication Failure Count:
Last Login Time:
Seconds Until Idle Account Lockout:
Password Is Reset:  false
Seconds Until Password Reset Lockout:
Grace Login Use Times:
Remaining Grace Login Count:  0
Password Changed by Required Time:
Seconds Until Required Change Time:
Password History:
```

Enabling an Account

```
D:\OpenDJ-2.4.6\bat>manage-account clear-account-is-disabled  -h localhost -p 4444  -D
"cn=admin" -w secret -b "uid=bbanner,ou=users,dc=example,dc=com"
The server is using the following certificate:
    Subject DN:  CN=Win7-1, O=Administration Connector Self-Signed Certificate
    Issuer DN:  CN=Win7-1, O=Administration Connector Self-Signed Certificate
    Validity:  Wed Sep 04 15:36:46 MST 2013 through Fri Sep 04 15:36:46 MST 2015
Do you wish to trust this certificate and continue connecting to the server?
Please enter "yes" or "no":yes
Account Is Disabled:  false
```

Notice `Account Is Disabled: false` in the listing.

Verifying an Account is Enabled

```
D:\OpenDJ-2.4.6\bat>manage-account get-all -h localhost -p 4444  -D "cn=admin" -w secret
-b "uid=bbanner,ou=users,dc=example,dc=com"
The server is using the following certificate:
    Subject DN:  CN=Win7-1, O=Administration Connector Self-Signed Certificate
    Issuer DN:  CN=Win7-1, O=Administration Connector Self-Signed Certificate
    Validity:  Wed Sep 04 15:36:46 MST 2013 through Fri Sep 04 15:36:46 MST 2015
Do you wish to trust this certificate and continue connecting to the server?
Please enter "yes" or "no":yes
Password Policy DN:  cn=Default Password Policy,cn=Password Policies,cn=config
Account Is Disabled:  false
Account Expiration Time:
Seconds Until Account Expiration:
Password Changed Time:  19700101000000.000Z
Password Expiration Warned Time:
Seconds Until Password Expiration:
Seconds Until Password Expiration Warning:
Authentication Failure Times:
Seconds Until Authentication Failure Unlock:
Remaining Authentication Failure Count:
Last Login Time:
Seconds Until Idle Account Lockout:
Password Is Reset:  false
Seconds Until Password Reset Lockout:
Grace Login Use Times:
Remaining Grace Login Count:  0
Password Changed by Required Time:
Seconds Until Required Change Time:
Password History:
```

11.5. Security PDP

The Security Policy Decision Point (PDP) module contains services that are able to perform authorization decisions based on configurations and policies. In the Security Framework, these components are called realms, and they implement the `org.apache.shiro.realm.Realm` and `org.apache.shiro.authz.Authorizer` interfaces. Although these components perform decisions on access control, enforcement of this decision is performed by components within the notional PEP application.

11.5.1. Security PDP AuthZ Realm

The Security PDP AuthZ Realm exposes a realm service that makes decisions on authorization requests using the attributes stored within the metacard to determine if access should be granted. This realm can use XACML and will delegate decisions to an external processing engine if internal processing fails. Decisions are first made based on the "match-all" and "match-one" logic. Any attributes listed in the "match-all" or "match-one" sections will not be passed to the XACML processing engine and they will be matched internally. It is recommended to list as many attributes as possible in these sections to avoid going out to the XACML processing engine for performance reasons. If it is desired that all decisions be passed to the XACML processing engine, remove all of the "match-all" and "match-one" configurations. The configuration below provides the mapping between user attributes and the attributes being asserted - one map exists for each type of mapping (each map may contain multiple values).

Match-All Mapping:: This mapping is used to guarantee that all values present in the specified metacard attribute exist in the corresponding user attribute. Match-One Mapping:: This mapping is used to guarantee that at least one of the values present in the specified metacard attribute exists in the corresponding user attribute.

11.5.1.1. Configuring the Security PDP AuthZ Realm

1. Navigate to the **Admin Console**.
2. Select **Security** Application.
3. Select **Configuration** tab.
4. Select **Security AuthZ Realm**.

See Security AuthZ Realm for all possible configurations.

11.5.2. Guest Interceptor

The goal of the `GuestInterceptor` is to allow non-secure clients (such as SOAP requests without security headers) to access secure service endpoints.

All requests to secure endpoints must satisfy the WS-SecurityPolicy that is included in the WSDL.

Rather than reject requests without user credentials, the guest interceptor detects the missing credentials and inserts an assertion that represents the "guest" user. The attributes included in this guest user assertion are configured by the administrator to represent any unknown user on the

current network.

11.5.2.1. Installing Guest Interceptor

The `GuestInterceptor` is installed by default with Security Application.

11.5.2.2. Configuring Guest Interceptor

Configure the Guest Interceptor from the Admin Console:

1. Navigate to the **Admin Console** at https://{FQDN}:{PORT}/admin
2. Select the **Security** application.
3. Select the **Configuration** tab.
4. Select the **Security STS Guest Claims Handler** configuration.
5. Select the **+** next to Attributes to add a new attribute.
6. Add any additional attributes that will apply to every user.
7. Select **Save changes**.

Once these configurations have been added, the GuestInterceptor is ready for use. Both secure and non-secure requests will be accepted by all secure DDF service endpoints.

11.6. Web Service Security Architecture

The Web Service Security (WSS) functionality that comes with DDF is integrated throughout the system. This is a central resource describing how all of the pieces work together and where they are located within the system.

DDF comes with a **Security Framework** and **Security Services**. The Security Framework is the set of APIs that define the integration with the DDF framework and the Security Services are the reference implementations of those APIs built for a realistic end-to-end use case.

11.6.1. Securing REST
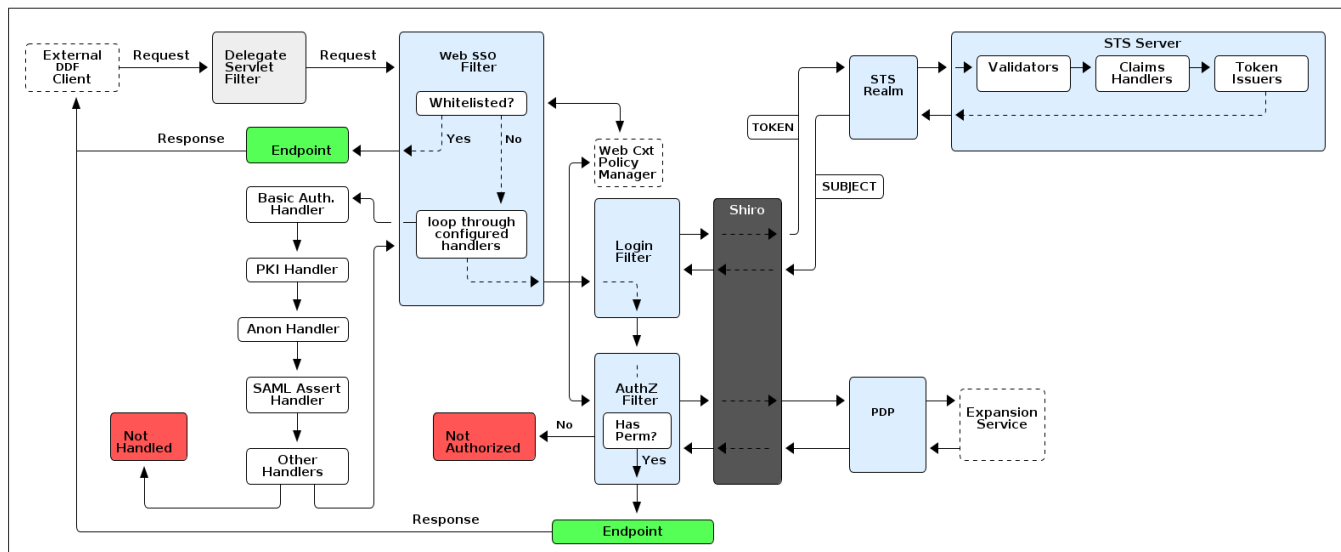
Security Architecture

The Delegate Servlet Filter is the topmost filter of all web contexts. It initializes all Security Filters and runs them in order according to service ranking:

1. The **Web SSO Filter** reads from the web context policy manager and functions as the first decision point. If the request is from a whitelisted context, no further authentication is needed and the request goes directly to the desired endpoint. If the context is not on the whitelist, the filter will attempt to get a claims handler for the context. The filter loops through all configured context handlers until one signals that it has found authentication information that it can use to build a token. This configuration can be changed by modifying the web context policy manager configuration. If unable to resolve the context, the filter will return an authentication error and the process stops. If a handler is successfully found, an auth token is assigned and the request continues to the login filter.

2. The **Login Filter** receives a token and returns a subject. To retrieve the subject, the token is sent through Shiro to the STS Realm where the token will be exchanged for a SAML assertion through a SOAP call to an STS server.

3. If the Subject is returned, the request moves to the **AuthZ Filter** to check permissions on the user. If the user has the correct permissions to access that web context, the request can hit the endpoint.
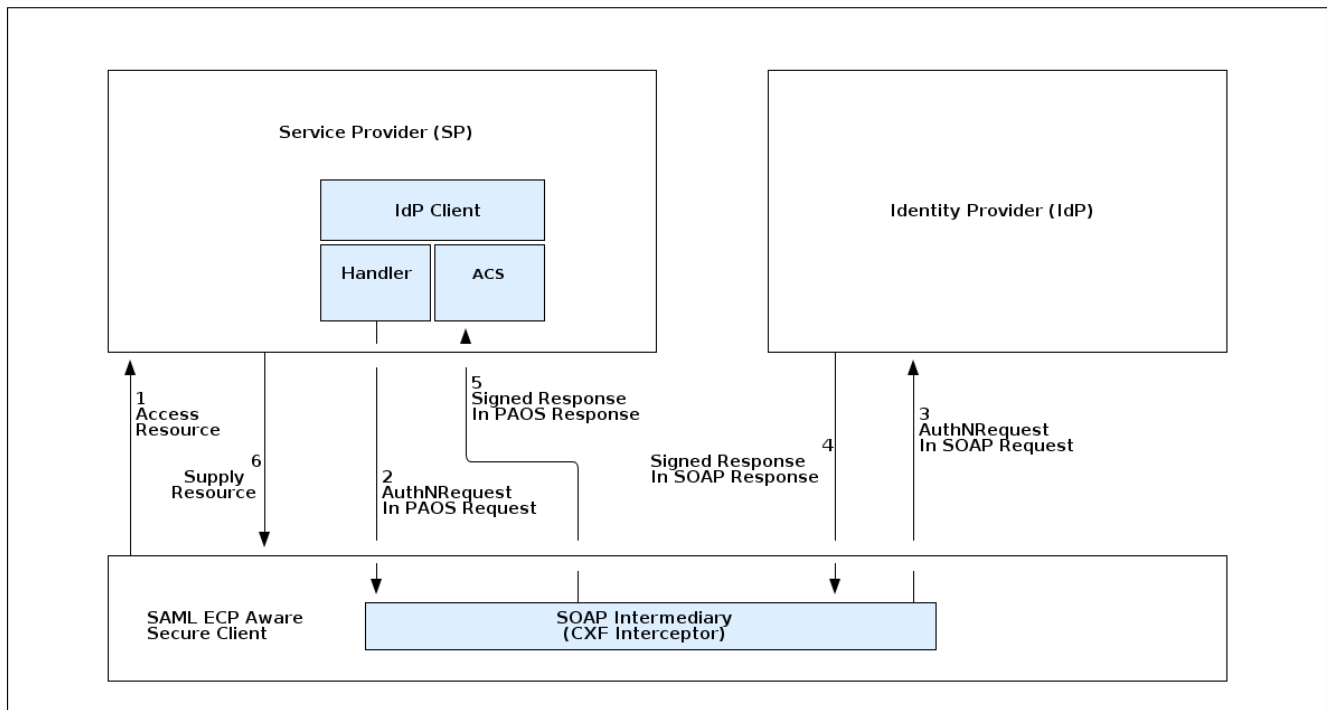
IdP Architecture
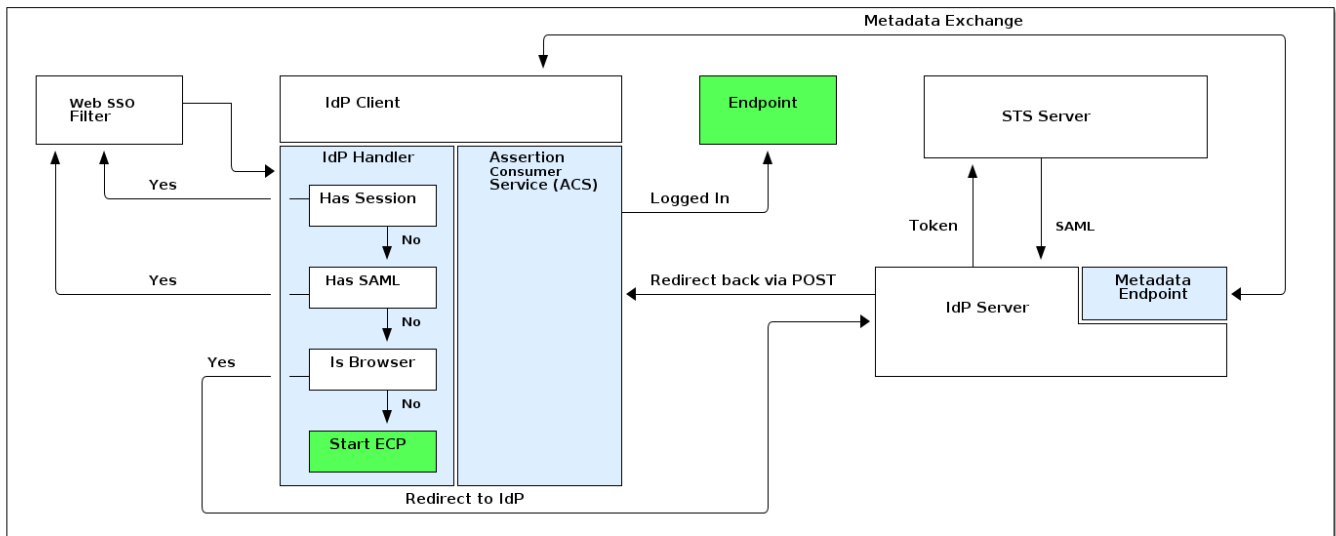
The IdP Handler is a configured handler on the Web SSO Filter just like the other handlers in the previous diagram. The IdP Handler and the Assertion Consumer Service are both part of the IdP client that can be used to interface with any compliant SAML 2.0 Web SSO Identity Provider.

The Metadata Exchange happens asynchronously from any login event. The exchange can happen via HTTP or File, or the metadata XML itself can be pasted into the configuration for either the IdP client or the IdP server that the system ships with. The metadata contains information about what bindings are accepted by the client or server and whether or not either expects messages to be signed, etc. The redirect from the Assertion Consumer Service to the Endpoint will cause the client to pass back through the entire filter chain, which will get caught at the `Has Session` point of the IdP Handler. The request will proceed through the rest of the filters as any other connection would in the previous diagram.

Unauthenticated non-browser clients that pass the HTTP headers signaling that they understand SAML ECP can authenticate via that mechanism as explained below.

Ecp Architecture

SAML ECP can be used to authenticate a non-browser client or non-person entity (NPE). This method of authentication is useful when there is no human in the loop, but authentication with an IdP is still desired. The IdP Handler will send a PAOS (Reverse SOAP) request as an initial response back to the Secure Client, assuming the client has sent the necessary HTTP headers to declare that it supports this function. That response does not complete the request/response loop, but is instead caught by a SOAP intermediary, which is implemented through a CXF interceptor. The PAOS response contains an `<AuthNRequest>` request message, which is intended to be rerouted to an IdP via SOAP. The SOAP intermediary will then contact an IdP (selection of the IdP is not covered by the spec). The IdP will either reject the login attempt, or issue a Signed `<Response>` that is to be delivered to the Assertion Consumer Service by the intermediary. The method of logging into the IdP is not covered by the spec and is up to the implementation. The SP is then signaled to supply the originally requested resource, assuming the signed Response message is valid and the user has permission to view the resource.

The ambiguity in parts of the spec with regard to selecting an IdP to use and logging into that IdP can lead to integration issues between different systems. However, this method of authentication is not necessarily expected to work by default with anything other than other instances of DDF. It does, however, provide a starting point that downstream projects can leverage in order to provide ECP based authentication for their particular scenario or to connect to other systems that utilize SAML ECP.

11.6.2. Securing SOAP
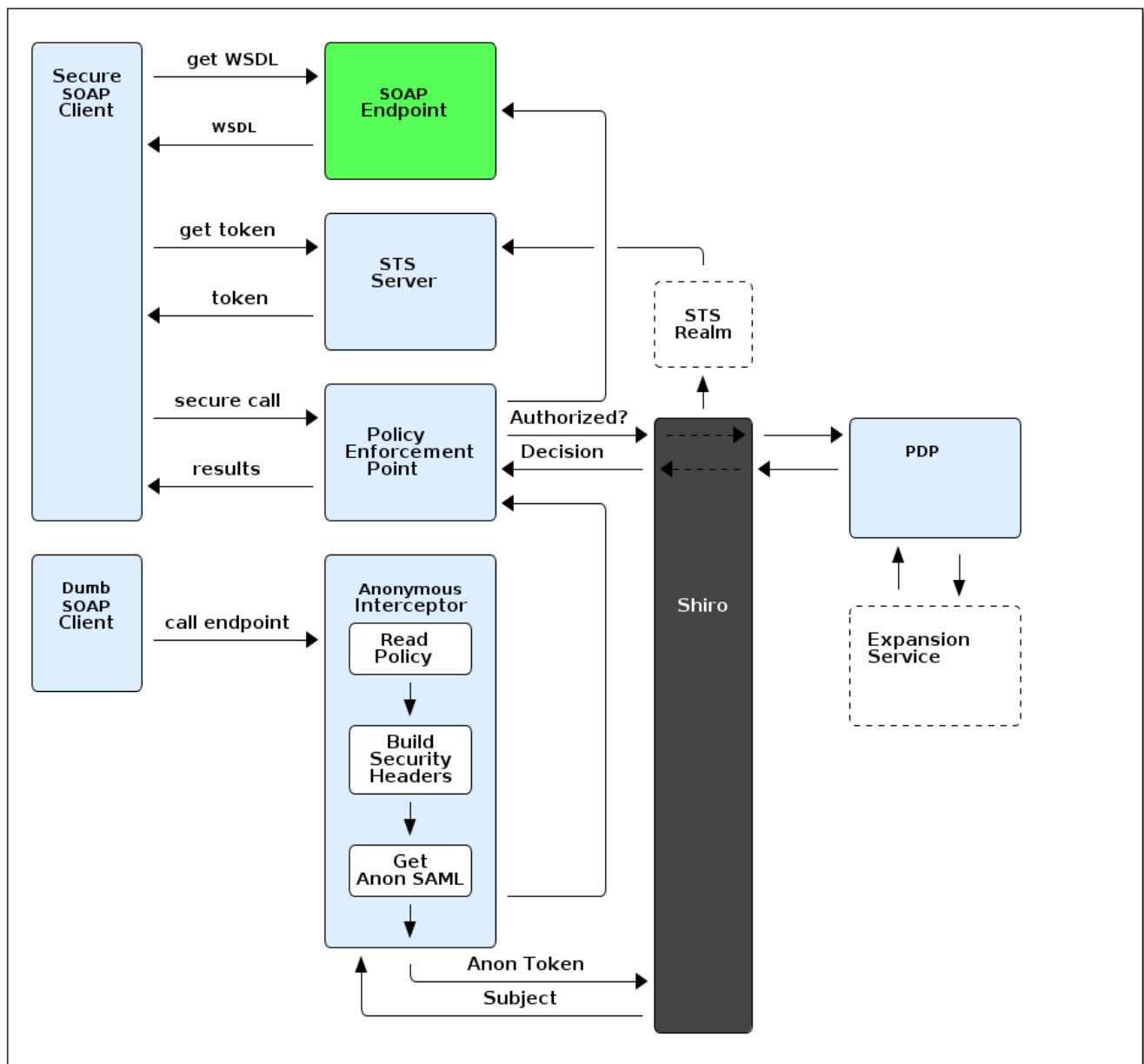
11.6.2.1. SOAP Secure Client

When calling to an endpoint from a SOAP secure client, it first requests the WSDL from the endpoint and the SOAP endpoint returns the WSDL. The client then calls to STS for authentication token to proceed. If the client receives the token, it makes a secure call to the endpoint and receives results.

11.6.2.2. Policy-unaware SOAP Client

If calling an endpoint from a non-secure client, at the point the of the initial call, the Guest Interceptor catches the request and prepares it to be accepted by the endpoint.

First, the interceptor reads the configured policy, builds a security header, and gets an anonymous SAML assertion. Using this, it makes a `getSubject` call which is sent through Shiro to the STS realm.

Upon success, the STS realm returns the subject and the call is made to the endpoint.

11.7. Security PEP

The Security Policy Enforcement Point (PEP) application contains bundles that allow for policies to be enforced at various parts of the system, for example: to reach contexts, view metacards, access catalog operations, and others.

11.7.1. Security PEP Interceptor

The Security PEP Interceptor bundle contains the `ddf.security.pep.interceptor.PEPAuthorizingInterceptor` class. This class uses CXF to intercept incoming SOAP messages and enforces service authorization policies by sending the service request to the security framework.

11.7.1.1. Installing the Security PEP Interceptor

This bundle is not installed by default but can be added by installing the `security-pep-serviceauthz` feature.

WARNING	To perform service authorization within a default install of DDF, this bundle MUST be installed.

11.7.1.2. Configuring the Security PEP Interceptor

The Security PEP Interceptor has no configurable properties.

11.8. Filtering

Metacard filtering is performed by the Filter Plugin after a query has been performed, but before the results are returned to the requestor.

Each metacard result will contain security attributes that are populated by the CatalogFramework based on the PolicyPlugins (Not provided! You must create your own plugin for your specific metadata!) that populates this attribute. The security attribute is a HashMap containing a set of keys that map to lists of values. The metacard is then processed by a filter plugin that creates a `KeyValueCollectionPermission` from the metacard's security attribute. This permission is then checked against the user subject to determine if the subject has the correct claims to view that metacard. The decision to filter the metacard eventually relies on the PDP (`feature:install security-pdp-authz`). The PDP returns a decision, and the metacard will either be filtered or allowed to pass through.

The security attributes populated on the metacard are completely dependent on the type of the metacard. Each type of metacard must have its own PolicyPlugin that reads the metadata being returned and returns the metacard's security attribute. If the subject permissions are missing during filtering, all resources will be filtered.

Example (represented as simple XML for ease of understanding):

```xml
<metacard>
    <security>
        <map>
            <entry key="entry1" value="A,B" />
            <entry key="entry2" value="X,Y" />
            <entry key="entry3" value="USA,GBR" />
            <entry key="entry4" value="USA,AUS" />
        </map>
    </security>
</metacard>
```

```xml
<user>
    <claim name="claim1">
        <value>A</value>
        <value>B</value>
    </claim>
    <claim name="claim2">
        <value>X</value>
        <value>Y</value>
    </claim>
    <claim name="claim3">
        <value>USA</value>
    </claim>
    <claim name="claim4">
        <value>USA</value>
    </claim>
</user>
```

In the above example, the user's claims are represented very simply and are similar to how they would actually appear in a SAML 2 assertion. Each of these user (or subject) claims will be converted to a KeyValuePermission object. These permission objects will be implied against the permission object generated from the metacard record. In this particular case, the metacard might be allowed if the policy is configured appropriately because all of the permissions line up correctly.

To enable filtering on a new type of record, implement a PolicyPlugin that is able to read the string metadata contained within the metacard record. Note that, in DDF, there is no default plugin that parses a metacard. A plugin must be created to create a policy for the metacard.

11.9. Expansion Service

The Expansion Service and its corresponding expansion-related commands provide an easy way for developers to add expansion capabilities to DDF during user attribute and metadata card processing. In addition to these two defined uses of the expansion service, developers are free to utilize the service

in their own implementations.

Each instance of the expansion service consists of a collection of rulesets. Each ruleset consists of a key value and its associated set of rules. Callers of the expansion service provide a key and a value to be expanded. The expansion service then looks up the set of rules for the specified key. The expansion service cumulatively applies each of the rules in the set, starting with the original value. The result is returned to the caller.

Table 30. Expansion Service Ruleset Format

Key (Attribute)	Rules (original → new)	
key1	`value1`	`replacement1`
	value2	`replacement2`
	value3	`replacement3`
key2	`value1`	`replacement1`
	value2	`replacement2`

The examples below use the following collection of rulesets:

Table 31. Expansion Service Example Ruleset

Key (Attribute)	Rules (original → new)	
Location	`Goodyear`	`Goodyear AZ`
	AZ	`AZ USA`
	CA	`CA USA`
Title	`VP-Sales`	`VP-Sales VP Sales`
	VP-Engineering	`VP-Engineering VP Engineering`

Note that the rules listed for each key are processed in order, so they may build upon each other, i.e., a new value from the new replacement string may be expanded by a subsequent rule. In the example `Location:Goodyear` would expand to `Goodyear AZ USA` and `Title:VP-Sales` would expand to `VP-Sales VP Sales`.

11.10. Security Token Service

The Security Token Service (STS) is a service running in DDF that generates SAML v2.0 assertions. These assertions are then used to authenticate a client allowing them to issue other requests, such as ingests or queries to DDF services.

The STS is an extension of Apache CXF-STS. It is a SOAP web service that utilizes WS-Trust. The generated SAML assertions contain attributes about a user and is used by the Policy Enforcement Point (PEP) in the secure endpoints. Specific configuration details on the bundles that come with DDF can be found on the Security STS application page. This page details all of the STS components that come out of the box with DDF, along with configuration options, installation help, and which services they

import and export.

The STS server contains validators, claim handlers, and token issuers to process incoming requests. When a request is received, the validators first ensure that it is valid. The validators verify authentication against configured services, such as LDAP, DIAS, PKI. If the request is found to be invalid, the process ends and an error is returned. Next, the claims handlers determine how to handle the request, adding user attributes or properties as configured. The token issuer creates a SAML 2.0 assertion and associates it with the subject. The STS server sends an assertion back to the requestor, which is used to authenticate and authorize subsequent SOAP and REST requests.

The STS can be used to generate SAML v2.0 assertions via a SOAP web service request. Out of the box, the STS supports authentication from existing SAML tokens, CAS proxy tickets, username/password, and x509 certificates. It also supports retrieving claims using LDAP and properties files.

11.10.1. STS Claims Handlers

Claims handlers are classes that convert the incoming user credentials into a set of attribute claims that will be populated in the SAML assertion. An example in action would be the LDAPClaimsHandler that takes in the user's credentials and retrieves the user's attributes from a backend LDAP server. These attributes are then mapped and added to the SAML assertion being created. Integrators and developers can add more claims handlers that can handle other types of external services that store user attributes.

11.11. Federated Identity

Each instance of DDF may be configured with its own security policy that determines the resources a user may access and the actions they may perform. To decide whether a given request is permitted, DDF references the SAML assertion stored internally in the requestor's Subject. This assertion is generated by the STS during authentication and contains a collection of attributes that identify the requestor. Based on these attributes and the configured policy, DDF makes an authorization decision. See Security PDP for more information.

This authorization process works when the requestor authenticates directly with DDF as they are guaranteed to have a Subject. However, when federating, DDF proxies requests to federated Sources and this poses a problem. The requestor doesn't authenticate with federated Sources, but Sources still need to make authorization decisions.

To solve this problem, DDF uses federated identity. When performing any federated request (query, resource retrival, etc), DDF attaches the requestor's SAML assertion to the outgoing request. The federated Source extracts the assertion and validates its signature to make sure it was generated by a trusted entity. If so, the federated Source will construct a Subject for the requestor and perform the request using that Subject. The Source can then make authorization decisions using the process already described.

How DDF attaches SAML assertions to federated requests depends on the endpoint used to connect to a

federated Source. When using a REST endpoint such as CSW, DDF places the assertion in the HTTP Authorization header. When using a SOAP endpoint, it places the assertion in the SOAP security header.

The figure below shows a federated query between two instances of DDF that support federated identity.
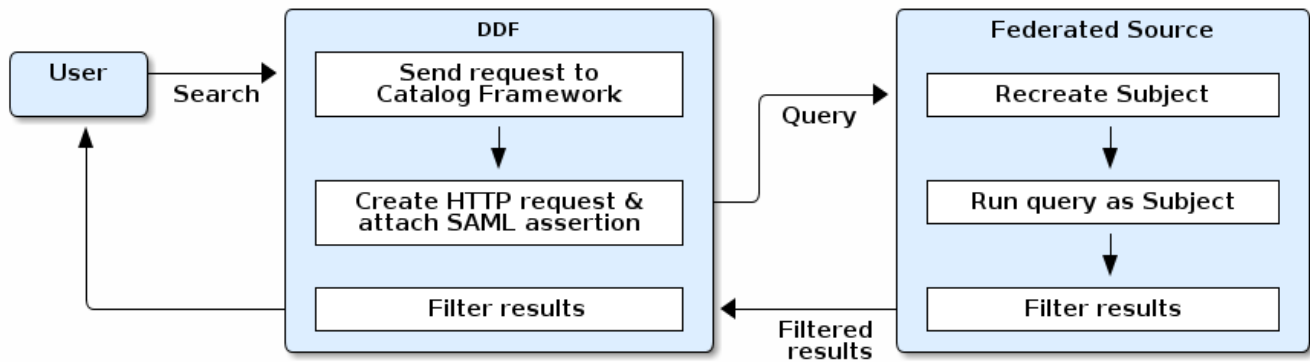
1. A user submits a search to DDF.

2. DDF generates a catalog request, attaches the user's Subject, and sends the request to the Catalog Framework.

3. The Catalog Framework extracts the SAML assertion from the Subject and sends an HTTP request to each federated Source with the assertion attached.

4. A federated Source receives this request and extracts the SAML assertion. The federated Source then validates the authenticity of the SAML Assertion. If the assertion is valid, the federated Source generates a Subject from the assertion to represent the user who initiated the request.

5. The federated Source filters all results that the user is not authorized to view and returns the rest to DDF.

6. DDF takes the results from all Sources, filters those that the user is not authorized to view and returns the remaining results to the user.

NOTE	With federated identity, results are filtered both by the federated Source and client DDF. This is important as each may have different authorization policies.

WARNING	Support for federated identity was added in DDF 2.8.x. Federated Sources older than this will not perform any filtering. Instead, they will return all available results and leave filtering up to the client.