HCLSoftware

Unica Content Integration V12.1.6 Developer Guide

Contents

Chapter 1. Overview

Unica Content Integration facilitates easy integration with Content Management Systems and enables searching content from them.

The fetched content can be used by the client of Unica Content Integration for various content-oriented business use cases. A Unica Content Integration client is any product from Unica Suite which integrates with it to consume the content from the target systems.

Plugins

To integrate with different CMS, Unica Content Integration uses REST APIs. Since each CMS has a unique programming interface, Unica Content Integration uses custom plugins or modules written specifically for the target CMS.

You can implement plugins using Java programming language. Unica Content Integration does not enforce any dependency of any third-party library for developing such plugins. You can customize plugins to utilize any third-party library for its implementation. Plugins can be used to fill in the logical gaps related to the target system.

Plugins non-intrusively augment Unica Content Integration to facilitate various content-oriented use cases.

Integration support and plugin development approach

Unica Content Integration provides out-of-the-box support for easy integration with RESTful interfaces. It also facilitates alternative approach of plugin development to integrate with non-RESTful systems such as database, file systems, or any other content repository.

A typical plugin written for REST API integration does not contain any logic to establish connection with the target system, and to handle protocol level success and failure conditions. Such responsibilities are handled by the Content Integration Framework. Plugins provide only system-specific pieces of information, such as:

- absolute location of the target API
- HTTP method to be used
- headers to be supplied
- request body to be sent
- type of the response to be expected
- transformer for the received response

An alternate plugin development approach for non-RESTful integration involves thorough implementation. For example, a plugin written for fetching content from database needs to address everything involved in making DB connection, executing SQLs, closing connections, result set hydration, failure handling etc.

Content Integration Framework follows Inversion of Control (IoC) pattern while working with plugins. Framework delegates to and/or calls back plugin services whenever respective user requests or system events occur.

RESTful content search flow

The following figure shows the end-to-end execution flow for RESTful content search:

Figure 1. RESTful content search flow

When Content Integration Framework receives content search request from user for the target system, it consults with the respective plugin to gather request specific logical information and makes an API call to the target system. It consults with the plugin once again to transform the API response into an expected format and responds to the user.

Non-RESTful content search flow

The following figure shows the end-to-end execution flow for Non-RESTful content search:

Figure 2. Non-RESTful content search flow
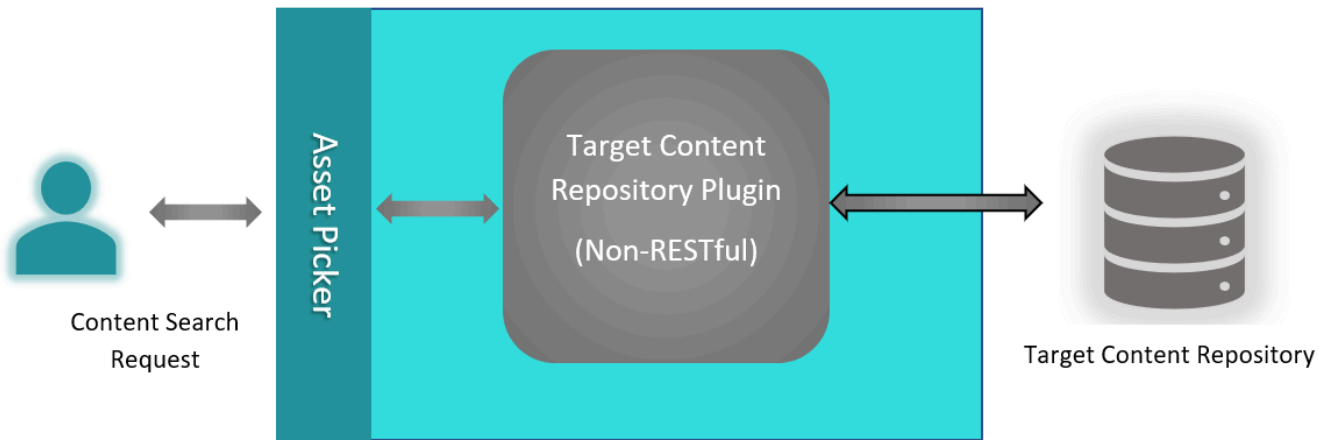
Non-RESTful plugins interact with the content repository and provides the search results to Content Integration Framework. Unlike RESTful repositories, Content Integration Framework will not know the type, architecture, protocol and the authentication mechanism used for communicating with the target repository.

Chapter 2. Plugin development overview

Unica Content Integration facilitates easy integration with new content repositories without having to alter the core Content Integration framework.

Unica Content Integration seamlessly integrates with system-specific, independent plugins. Once the plugin is developed and placed under the `<ASSET_PICKER_HOME>/plugins/custom` directory on the application server hosting Content Integration, the corresponding content repository can be onboarded in the Unica product suite by updating a few configurations in Unica Platform. For more information, see *Unica Content Integration Administrator's Guide*

> 📝 **Note:** *<ASSET_PICKER_HOME>* refers to the base installation directory of Unica Content Integration placed within Platform home. Hence, any further use of *<ASSET_PICKER_HOME>* in this guide should be considered as a path to the Content Integration directory within Platform home.

Unica Content Integration is shipped with a development kit containing the dependencies, reference projects, and a starter project to quick start the plugin development. Development kit is placed within the `<ASSET_PICKER_HOME>/dev-kits` directory. Four reference projects, named `aem-integration`, `wcm-integration`, `dx-integration`, and `commerce-integration` are available for Adobe Experience Manager (AEM), IBM Web Content Manager (WCM), HCL Digital Experience and HCL Commerce, respectively.

Components of plugin

A typical plugin contains the following components:

- Service declarations on page 4
- Service implementations on page 15
- I18n resource bundle

The term Service represents a Java class, which either indirectly aids in consuming an external REST service, or directly interacts with external web service(s) or system(s) for a designated purpose. External system need not be a standard Content Management System and external services need not belong to any standard CMS. It can be any system or an API.

Any service implemented by the plugin must be declared in a centrally managed service declaration file. A service declaration file is an YML configuration file containing the list of services implemented by all the available plugins. The service declaration file must be named `custom-plugin-services.yml`. It should be available within the `<ASSET_PICKER_HOME>/conf` directory. Structure of `custom-plugin-services.yml` file must be similar to the `plugin-services.yml` file, which exists in the same directory. The `plugin-services.yml` file contains service declarations for out-of-the-box system integrations. A service can either be a standard service or a custom service.

Standard services carry special semantics and purpose in Unica Content Integration. Implementation of certain standard services is mandatory for Content Integration Framework to work with the content repository.

Service declarations

Reference service declarations can be found inside `asset-integration-starter` project within `dev-kits\asset-integration-starter\src\main\resources\META-INF` directory.

The following are example service declarations from `asset-integration-starter` project:

```
services:
  -
    systemId: Foo
    serviceName: simple-search
    factoryClass: com.example.service.rest.SimpleSearchService
    params:
      supportedContentTypes: # Standard parameter, applicable only for simple-search service
        Images: Images
      customParam1: p1Value # String parameter
      customParam2: 1234.56 # Numeric parameter
      customParam3: # Key-value/Dictionary/Map parameter
        p3Key1: p3Value1
        p3Key2: p3Value2
        p3Key3: p3Value3
      customParam4: # Array parameter
        - p4Value1
        - p4Value2
        - p4Value3

  -
    systemId: Foo
    serviceName: resource-loader
    factoryClass: com.example.service.rest.ResourceLoaderService
    params:
      customParam1: p1Value # String parameter
      customParam2: 1234.56 # Numeric parameter
      customParam3: # Key-value/Dictionary/Map parameter
        p3Key1: p3Value1
        p3Key2: p3Value2
        p3Key3: p3Value3
      customParam4: # Array parameter
        - p4Value1
        - p4Value2
        - p4Value3
  -
    systemId: Foo
    serviceName: asset-selection-callback
    factoryClass: com.example.service.rest.ContentSelectionCallbackService
    params:
      customParam1: p1Value # String parameter
      customParam2: 1234.56 # Numeric parameter
      customParam3: # Key-value/Dictionary/Map parameter
        p3Key1: p3Value1
        p3Key2: p3Value2
        p3Key3: p3Value3
      customParam4: # Array parameter
        - p4Value1
        - p4Value2
        - p4Value3
  -
    systemId: Foo
```

```
serviceName: custom-service
factoryClass: com.example.service.rest.ExampleExampleCustomService
params:
  customParam1: p1Value # String parameter
  customParam2: 1234.56 # Numeric parameter
  customParam3: # Key-value/Dictionary/Map parameter
    p3Key1: p3Value1
    p3Key2: p3Value2
    p3Key3: p3Value3
  customParam4: # Array parameter
    - p4Value1
    - p4Value2
    - p4Value3
```

Service declaration file

Service declaration file contains `services` element, which is an array of individual service declarations. A service declaration is a dictionary containing three mandatory elements named `systemId`, `serviceName`, and `factoryClass`, and one optional element named `params`. Details of the elements are as follows:

- `systemId`

  This string value uniquely identifies a target content repository. This identifier should preferably contain only English alphanumeric characters. Use dots, dashes, and underscores to enhance readability. Avoid any other special characters and unicode characters. Identifier once chosen for the target system must remain consistent across all service declarations for the same system. This identifier is also used in Unica Platform configuration for onboarding the respective system.

  The following are some examples of valid system identifiers:

  ```
  WCM
  AEM
  Example
  WCM_1.0
  AEM_1_1
  DX-CORE
  DX
  ```

  You can write different plugins for different versions of the same system. In such case, different identifiers must be used to identify each version distinctly. Alternatively, the same plugin may contain different versions of service implementations specific to the respective versions of the corresponding system. In such case, different systemIds must be carefully assigned to the respective service declarations. For example, two different versions of WCM, namely 1.0 and 2.0 may contain different APIs for content search service, thereby causing following service entries for respective versions:

  ```
  -
    systemId: WCM_1.0
    serviceName: simple-search
    factoryClass: com.hcl.wcm.service_1_0.WcmSimpleSearchService

  -
    systemId: WCM_2.0
    serviceName: simple-search
    factoryClass: com.hcl.wcm.service_2_0.WcmSimpleSearchService
  ```

The two entries may belong to the same plugin or may be placed in two different plugins for the sake of implementation clarity. Content Integration Framework does not impose any restrictions.

- `serviceName`

  This string value uniquely identifies the given service for corresponding system. It can either be a name of Standard service, or an appropriately chosen name for the custom service. Standard services are discussed in subsequent sections.

- `factoryClass`

  This is a fully qualified path to the Java class providing service implementation.

- `params`

  Provides a way to supply static parameters to the service to control, or modify, service behavior according to the parameter values. In short, `params` can be used to hold static key-value configuration for service implementations. This can include certain standard service parameters as well as any custom parameters that a service might want to use. Parameter values are converted into the objects of closest matching primitive wrapper classes, such as Integer, Long, Double, String etc. A parameter value can also be a map, array, or list of other values (plugins must verify the runtime-type of these values before using them).

Service declaration file also contains certain properties pertaining to the target content repository. These properties are covered under systems root element. The following is an example of such entry containing all the supported properties:

```yaml
systems:
  YOUR_SYSTEM_ID:
    params:
        param1: value1
        param2:
            k1: v1
            k2: v2
        param3: 100
    additionalFeatures:
        securityPolicy: false
        content:
            paginatedSearch: true
            paginatedList: true
            anonymousContent: true
            categorization: lenient
```

This example entry shows the default values considered for each property mentioned herein in case no such entry is present for the given target repository. Thus, this entry is optional unless one or more of these default considerations do not hold true for the target content repository. Below section briefs the significance of each property:

`params` - Provides a way to supply static parameters to the respective plugin to control or alter plugin behavior according to the parameter values. In short, params can be used to hold static key-value configuration for plugin implementations. This can include predefined standard system parameters as well as any custom parameters that a respective plugin might want to use. Parameter values are converted into the objects of closest matching primitive wrapper classes, such as Integer, Long, Double, String etc. A parameter value can also be a map, array, or list of other values (plugins must verify the runtime-type of these values before using them).

`additionalFeatures | securityPolicy` - This setting must be set to true when content is protected inside respective system using Unica's security policies.

`additionalFeatures | content | paginatedSearch` - This feature flag is used to convey whether content repository supports paginated content search results or not. User experience is altered accordingly for showing content search result.

`additionalFeatures | content | paginatedList` - This feature flag is used to convey whether content repository supports paginated content listing or not. User experience is altered accordingly for showing content list.

`additionalFeatures | content | anonymousContent` - This feature flag is used to convey whether publicly accessible content should be expected from the content repository or not. If it is set to true, plugin must return publicly accessible URL for each content. If contents cannot be made publicly accessible using HTTP(S) URL, plugin developer must set this flag to false. In such case, users will not be able to see or download the contents fetched from the repository. If the target system does not provide anonymously-accessible URL for the content, you must implement the `resource-loader` service to allow download of protected content.

`additionalFeatures | content | categorization` - This property can be set to either **strict** or **lenient**. If it is left out, it is assumed to be lenient by default. Content is generally organized in different categories. Value of this property provides a hint about the structural similarities of contents across categories. Strictly organized contents are assumed to be heterogeneous. Such contents may not share any attributes across categories. For example, insurance products can be considered as strictly categorized. Life insurance, Automobile insurance, Health insurance, Travel insurance etc. have very few attributes in common. Such products possess exclusive attributes relevant to the category they belong to. Whereas leniently categorized contents are assumed to be homogeneous by and large. They share most of their attributes across categories. For example, products in an e-commerce application can be considered leniently categorized since most of the products across all categories have similar attributes such as product title, description, SKU, price, manufacturer, discount, imagery, rating, offers, weight, warranty information etc.

Value of this property helps in deciding whether users should be allowed to pick contents across categories while using **Content Integration** feature in **Centralized Offers Management**. **Strict** will restrict users to the contents of relevant category, whereas **lenient** will allow users to choose content from any category regardless of the category used for setting up attribute mapping.

In addition to standard system parameters, any custom parameters can also be declared similarly. Likewise, the **Additional parameters** section in Platform configuration can also be utilized for admin configurable system parameters. Note that the **Additional parameters** in Platform configuration does not support nested & collection (array & dictionary) parameters. It must be a simple **key: value** pair, wherein key & value is separated by a colon & a space. Each key/value pair must be on separate line.

For example,

```
param1: value1
param2: 2
param3: 1.2
```

The **Additional parameters** in Platform configuration as well as system parameters from YML file can be obtained by invoking **getParams()** method on standard **SystemConfig** object available to each service.

Standard services

The following table introduces the standard services of Unica Content Integration. Hence, none of the service names listed herein should be used for any custom service implementation. Content Integration SDK provides standard interfaces and types to implement these standard services. These interfaces and types are discussed in more detail in subsequent sections.

Table 1. Standard services and their description

Standard service name	Description
`simple-search`	Simple search service responds to the content search requests received by Content Integration Framework. This service accepts the search query string along with required result pagination details. Based on the success of search operation, it returns the search result for given search query and according to the required pagination. This is a mandatory service for the plugin.
`list-folders`	This is an optional service. Folder is a general term used to represent a container object used in target system to hierarchically organize the contents. This service is invoked to render the list of folders & sub-folders to facilitate navigation through such hierarchically organized contents. ✏️ **Note:** `list-folders` and `list-contents` are correlated services. Implementation for both services must exist for content navigation to function properly.
`list-contents`	This is an optional service. This service is invoked for listing the contents belonging to a particular folder. ✏️ **Note:** `list-folders` and `list-contents` are correlated services. Implementation for both services must exist for content navigation to function properly.
`get-content-details`	Implementation of this service is useful for retrieving the details of an individual content. Contents obtained using `simple-search` & `list-contents services` are referenced further in other Unica products. Users might want to see the details of already referenced content at later point of

Table 1. Standard services and their description (continued)

Standard service name	Description
	time. Therefore, we encourage to implement this service to facilitate users to see the content details on demand.
`get-object-schema`	This is an optional service. Object schema in simple terms refers to the meta information about the properties or attributes present in the given type of object. Implementation of this service is useful for - • allowing Centralized Offer Management users to map Offer attributes with content attributes • creating Custom Actions in Centralized Offer Management • letting Journey users create Entry sources by leveraging attribute mapping between Data Definitions & Content/Object received from respective external system
`resource-loader`	This service is useful when direct download of the content from target system is not feasible. This service is not mandatory and should be implemented only when following challenges are encountered: • **If no direct web link exists to download the contents** Contents returned by the `simple-search` and `list-contents` services must include an absolute URL to the respective content so that Content Integration client can download it directly over the web. If no such direct web link to the content is present, then it is necessary to implement the `resource-loader` service by overriding the default implementation provided by Content Integration Framework. For example, if the contents are maintained in a database table, then the `simple-search` and `list-contents` services will fetch records from the database. Since the items are loaded from the database, there may not be any URL directly pointing to each record. In such case, the `resource-loader` service can make use of the content identifier to locate and provide the

Table 1. Standard services and their description (continued)

Standard service name	Description
	appropriate data whenever content download is requested. All content download requests will go through the Content Integration Framework, which will delegate the downloading task to the `resource-loader` service by providing it the content URL and its identifier.
	• **If web links to the contents are protected**
	Certain systems may not provide anonymous access to the contents despite of the availability of direct web links. In such cases, access is generally provided only after supplying required authentication details. By default, Content Integration Framework registers an out of the box implementation of `resource-loader` service for each plugin. This default implementation makes use of the real content URL to download the content from remote system by supplying appropriate authentication details subject to the configurations in Unica Platform. (For more information on system onboarding configurations, see *Unica Content Integration Administrator's Guide*).
	Alternatively, plugins can override the default resource-loader implementation to alter the content downloading behavior (using content URL or content identifier). If the `resource-loader` service is overridden using RESTful approach, Content Integration Framework will continue to take care of supplying authentication details based on the Platform configuration.
	Note: Content must be made anonymously accessible if it is expected to be seen/accessed by the external audience. In such case, usage of resource-loader service is not encouraged in production systems. Usage of `resource-loader` service can be turned off any time by

Table 1. Standard services and their description (continued)

Standard service name	Description
	setting the Anonymous Content property to `Yes` in Platform configuration. Likewise, it can be turned on by setting the same property to `No`.
`list-category-folders`	This is an optional service. Some systems might organize contents/entities into different folders. Likewise, content categories might also be organized in folders. Implementation of this service helps to navigate folders for selecting desired content category. The list-content-categories service mentioned further in this section receives the ID of desired folder to list the categories from.
`list-content-categories`	Content can be logically categorized by its natural classification. For example, Digital content can be categorized into Images, Documents, Multimedia (audios and videos), Archives etc. Similarly, E-commerce products can be categorized into several broad categories, such as Electronics, Healthcare, Books, Furniture etc. Content Integration Framework allows following ways of conveying such content categorization to facilitate searching contents within specific category. • **`supportedContentTypes` service parameter** A standard service level parameter, `supportedContentTypes`, can be used to statically supply a dictionary of supported content types under `simple-search` service declaration. • **`getSupportedContentTypes()` method in search service implementation** `getSupportedContentTypes()` method can be overridden to dynamically generate a map of supported content types, wherein key serves as the category identifier and value serves for the label displayed on the UI. This method is executed during the application startup, hence no remote API call can be made using Content Integration Framework's capabilities since application might

Table 1. Standard services and their description (continued)

Standard service name	Description
	be in partially initialized state when this method is invoked. • `list-content-categories` **service** Optionally, list-content-categories service can be implemented to address the limitation of `getSupportedContentTypes()` method. It enables remote API calls to be made for fetching the content categories even more dynamically. If implemented, this service overrides the earlier mentioned approaches. Content Integration Framework invokes this service whenever content search popup is rendered.
`get-content-category-details`	This is an optional service. This service helps to retrieve details of a specific content category using its identifier. It is advised to implement this service if list-content-categories is implemented for dynamically fetching categories from remote system or categories are listed using folder hierarchy. This service need not be implemented if categories are maintained using supportedContentTypes parameter to simple-search service or getSupportedContentTypes() method in simple-search service.
`get-cognitive-analysis`	This is an optional service. If implemented, it is used to fetch cognitive details associated with the given image, subject to the "Preferred cognitive service provider" configuration in Unica Platform.
`Object extension service`	This is an optional service. This standard service does not mandate any standard service name. There can be more than one object extension services. The core purpose of any object extension service is to enhance the capability of any object (such as Offer in Centralized Offer Management) by leveraging the attribute mapping feature. Centralized Offer Management leverages object extension services by means of Custom Actions in Offers. 📝 **Note:** `Object extension` and `get-object-schema` are correlated services. Implementation of

Table 1. Standard services and their description (continued)

Standard service name	Description
	`get-object-schema` service is mandatory for object extension feature to function properly. get-object-schema service is expected to support schema generation for the contents in the context of **Content Integration** capabilities in **Centralized Offer Management**. In case of object extension capability, `get-object-schema` service is additionally expected to support the schema generation for the object returned by the respective object extension service. `get-object-schema` service accepts the type/category of the object/content to generate the schema for. In case of object extension service, it receives the type in **{service-name}.request & {service-name}.response** format. For example, if `fetch-additional-attributes` is the name given to the object extension service, then `get-object-schema` receives **fetch-additional-attributes.request & fetch-additional-attributes.response** as the type of object whenever schema for `fetch-additional-attribute` service's request & response objects is required.
`Object event interpreter service (Webhook)`	This is an optional service. Unlike the services listed above, this is in fact an inbound RESTful service. If implemented, it lets the plugin expose webhooks in Content Integration to be able to receive content lifecycle events from the respective content repository. The core responsibility of this service is to help Content Integration Framework interpret the semantics associated with incoming event so that Content Integration Framework can process the event accordingly. Incoming content events are relayed to **Centralized Offer Management** for automatically synchronizing Offers using associated content details. It leverages the attribute mapping feature to synchronize Offers automatically.

Table 1. Standard services and their description (continued)

Standard service name	Description
	There is no standard name for this service. Plugin can choose any appropriate name. Likewise, Unica Journey can also use Webhooks for syndicating Entry Sources.
`Object event interpreter service (Kafka listener)`	This is an optional service. This service adds listener(s) to the Kafka topic(s) for handling object events coming via Kafka. There is no standard name for this service. Plugin can choose any appropriate name. Events received either via Webhook or Kafka are used for automatically synchronizing/update various objects in Unica products, such as Offers in Centralized Offer Management, and user Journeys in Unica Journey. Auto synchronization happens entirely based on the attribute mappings facilitated by `get-object-schema` service.
`Kafka message publisher service`	This is an optional service. There is no standard name for this service. Plugin can choose any appropriate name. This service allows plugins to publish messages (JSON and/or Avro) onto Kafka topic(s). Content Integration Framework does not directly invoke this service. Plugin can treat it as custom service and invoke it wherever required. There are no special semantics associated with any Kafka publisher service. Single Kafka publisher service can publish same message in different supported formats onto different topics. Plugin can have more than one publisher service if different messages need to be sent onto different topics.

Standard collaboration services – Collaboration services facilitate integration with external collaboration systems such as Slack, Microsoft Teams etc. **Unica Plan** leverages such services to integrate its Message Board with desired collaboration systems. Content Integration Framework provides out of the box integration with Slack. Content Integration Framework allows integrating with any other similar system by means of these service implementations.

`collaboration.create-channel`	This is a mandatory service. It is used for creating a communication channel in respective collaboration system. All subsequent collaborative operations happen using the same channel. For example, addition of

	members/collaborators to the channel, removal of members from channel, posting messages etc.
`collaboration.update-channel`	This is a mandatory service. It is used for updating details of existing collaboration channel. As of current release, it is used mainly for renaming a channel.
`collaboration.get-channel-details`	This is a mandatory service. It is used for fetching details of already known channel using its unique identifier.
`collaboration.add-members-to-channel`	This is a mandatory service. It is used for adding new members/collaborators to an existing collaboration channel.
`collaboration.remove-members-from-channel`	This is a mandatory service. It is used for removing existing members/collaborators from already created channel.
`collaboration.post-message-to-channel`	This is a mandatory service. It is used for posting a new message to existing collaboration channel.

Service implementations

For each service declared in the service declaration file, there must be an implementation present inside the respective `factoryClass`.

The Content Integration Framework provides an SDK to streamline the service implementation and facilitates rapid development of plugins. The Content Integration SDK allows two different approaches for service implementations: RESTful and Functional.

This section will provide a brief introduction to these approaches. For additional information, refer the `asset-integration-starter` project.

This topic also introduces certain types, interfaces, their generic type parameters, and enums from Content Integration SDK. For additional details, see .

RESTful approach

The `com.example.service.rest.ExampleCustomService` class helps you understand REST based service implementation.

This class is an implementation of `RestService` interface, and thus represents a REST based service. Since REST is completely based on HTTP standards, the `RestService` interface in Content Integration SDK is extended from `HttpService` interface and is defined as a marker interface. The `RestService` interface does not declare any additional method of its own. Listed below are the methods declared in `HttpService` interface, which REST based service implementation must implement. Not all methods are mandatory. All methods accept `ExecutionContext` object, which contains all the contextual information necessary for every method to perform its designated task. The generic type parameter to the `ExecutionContext` class represents the type of input required for the respective service on its invocation.

- **HttpRequest buildRequest(ExecutionContext<RQ> executionContext)**

This is a mandatory method. It returns an object of type `com.hcl.unica.system.model.request.HttpRequest`. The `HttpRequest` class provides builder API to construct the object with applicable details. This object comprises all the required details for making an HTTP request, such as endpoint URL, HTTP method, HTTP headers, and HTTP request body. The `HttpRequest` builder API accepts the following arguments:

- ◦ **String endpointUrl**

  An absolute URL to target API.

- ◦ **HttpMethod httpMethod**

  HTTP method to be used for making API call. Must be one of the values from `com.hcl.unica.system.integration.service.HttpMethod` enum.

- ◦ **Optional<Map<String, Object>> headers**

  An optional Map of HTTP headers. It can include standard as well as custom HTTP headers. Header names must be specified in terms of Map keys, and header values must be supplied as corresponding values in the Map. In the absence of this optional value, no custom headers will be sent along with the outgoing HTTP request.

  Note: Although the header Map accepts values of type Object (or its subtypes), only String objects are supported as of current implementation of Content Integration Framework. Any other type of value will be ignored, and following warning will be logged:

  ```
  Header '{HEADER_NAME}' with value '{TO_STRING_REPRESENTATION}' will not be set since it is
    not a String and no Converter is available.
  ```

- ◦ **Optional<?> payload**

  If the target service expects any request body, then this argument can be supplied with the desired HTTP request body. It can be any valid object so long as appropriate `Content-Type` header is supplied in the headers Map. In the absence of this argument, empty request body will be sent along with the outgoing HTTP request.

  Note: Jackson and JAXB Support: Object serialization using Jackson and JAXB is completely supported by the Content Integration Framework. Thus, appropriately decorated object with Jackson or JAXB annotations can be set as the request payload. In such case, appropriate `Content-Type` header must be specified in headers Map. Serialization of supplied object into the request body is handled by the Content Integration Framework, hence no explicit serialization is required.

- **Object transformResponse(HttpResponse<RS> response, ExecutionContext<RQ> executionContext)**

This optional method transforms the HTTP response into a desired format. The first argument, `com.hcl.unica.system.model.response.HttpResponse`, to this method, represents the response received from the target system. The generic type parameter to the `HttpResponse` class represents the type of response body, or response payload, expected from the remote API. Response payload can be of any type, such as a String containing the entire text as received from the service, a byte array containing the response body, or a deserialized POJO representing the response JSON/XML. In addition to the response payload, `HttpResponse` object can be used to obtain response headers, status code, and cookies. Note that `transformResponse` method is invoked only when successful

(2xx) response is received from respective system. For error response, `HttpServiceExecutionException` is thrown by Content Integration Framework.

> **Note: Jackson and JAXB Support**: Object deserialization using Jackson and JAXB is completely supported by Content Integration Framework. Thus, appropriately decorated object with Jackson or JAXB annotations can be accepted as an argument to this method. Deserialization of response body into specified type is handled by Content Integration Framework, hence no explicit deserialization is required during response transformation inside this method.

In the absence of this implementation, no implicit transformation is performed by the Content Integration Framework.

In addition to these methods, there is one more method the `getServiceInterface` inherited from `com.hcl.unica.system.integration.service.AbstractService interface`, that needs to be implemented by the service. But its implementation is more relevant to the service invocation rather than service implementation.

Content Integration Framework takes care of real HTTP interaction with target system and simply consults with service object to obtain earlier mentioned details.

Error Handling: Errors or exceptions received during HTTP call are handled by the Content Integration Framework. Methods listed earlier must not throw any checked exception. Unchecked exceptions can be thrown if required.

Functional approach

Refer to the `com.example.service.functional.ExampleCustomService` class to understand the functional service implementation.

This class is an implementation of FunctionalService interface. Unlike REST based service, there are no HTTP specific callback methods in this type of service implementation. In fact, functional service may not necessarily be related to any HTTP invocation. This type of service can include any operation which has no out of the box support from Content Integration Framework. It can talk to the database, invoke third party web service, do the file system operation etc.

Implement the following method for a functional service. This method also accepts an argument of type `ExecutionContext`, containing the contextual information required for completing the desired task. The generic type parameter to the `ExecutionContext` class represents the type of input required for the respective service on its invocation.

- **RS execute(ExecutionContext<RQ> executionContext)**

  This method performs its designated task using the contextual information passed to it. In return, it gives the desired value after finishing its operation. The return value shown in this signature is a generic type and is based on the type used while implementing `FunctionalService` interface.

Error Handling

Above method must not throw any checked exception. Unchecked exceptions can be thrown if required.

Common methods

The following are the common methods applicable for RESTful as well as Functional services. These methods are inherited from `com.hcl.unica.system.integration.service.AbstractService` interface.

- **Class<? extends ServiceGateway<RQ, ?>> getServiceInterface()**

  Implementation of this method is more relevant to the service invocation rather than service implementation. For more information, see Plugin Development SDK on page 20.

- **void init(SystemConfig systemConfig, ServiceConfig serviceConfig)**

  Override this optional method to perform one-time initialization (after service object construction), prior to serving any request. Use the SystemConfig object and the ServiceConfig object, passed to this method, to obtain system and service-specific details respectively to make necessary initializations, such as obtaining a DB connection, opening a file handle etc. A separate object of your service class is created for each individual system configuration in Unica Platform. Thus, if the same target system is configured for two different partitions in Unica Centralized Offer Management, then two different objects of your service class will be created for each partition. Likewise, if the same target system is configured for any other Unica product, a separate object for that configuration will exist. The `com.hcl.unica.system.integration.config.SystemConfig` object encapsulates all the system configurations made in Unica Platform Configuration section, whereas `com.hcl.unica.system.integration.config.ServiceConfig` object holds all the configurations made for the corresponding service in `<ASSET_PICKER_HOME>/conf/plugin-services.yml` and `<ASSET_PICKER_HOME>/conf/custom-plugin-services.yml` files. These objects are also accessible using `ExecutionContext` in all the methods discussed earlier.

  > **Note:** Content Integration Framework does not provide any special end-of-lifecycle method for services to clean up the things initialized inside the init method. We recommend you to use the standard Java approach by implementing the finalize method, if necessary.

Best approach selection

Although, it is possible to implement a service using either approaches, each approach has some advantages and limitations when it comes to the capabilities.

1. **RESTful approach**
   a. Advantages
      - Less verbose & reads closer to the typical HTTP interaction
      - Out of the box transport level error handling
      - Out of the box support for retrial in case of temporary outages
      - Out of the box support for proxied connectivity
      - Out of the box support for future enhancements in Content Integration Framework
   b. Limitations
      - Cannot be used for non-RESTful or non-HTTP integrations, such as database or file system interactions
2. **Functional approach**

      a. <u>Advantages</u>
- Can be used for non-RESTful or non-HTTP integrations, such as database or file system interactions

      b. <u>Limitations</u>
- No out-of-the-box support available for transport level error handling, retrials, proxied connectivity, and any future enhancements from Content Integration Framework.
- If required, the explicit implementation of missing out-of-the-box capabilities can make service implementations very verbose.

You can see that the Functional approach is well suited for non-RESTful or non-HTTP based integrations. Any service implemented using RESTful approach can also be implemented using Functional approach by taking care of all the necessary out-of-the-box capabilities provided by Content Integration Framework. While Functional approach gives flexibility in terms of implementation design, it takes away a few useful capabilities.

Chapter 3. Plugin Development SDK

This topic provides information about the various classes, interfaces, and enums from the Content Integration SDK, with the help of corresponding logical units in `asset-integration-starter`, `aem-integration`, and `wcm-integration` reference projects that are included as a part of development kit along with the Content Integration feature.

Content Integration SDK for plugin development can be found under `<ASSET_PICKER_HOME>/dev-kits/sdk/` directory on your application server. The following jars can be found inside the `sdk` directory:

- `integration-api.jar`
- `entity-mapper-api.jar`
- `standard-integrations.jar`

These jars contain all the SDK classes, interfaces & enums discussed in this section. Check out the relevant classes from these jars whenever you come across the respective topic in this guide.

Generic type parameters

Generic type parameters are used for implementing service interfaces.

A service that resides in a plugin is just a programming unit, which takes some input and returns the expected output. Similarly, the REST API, wrapped by our service, asks for the required input (request body, headers, cookies, and query parameters) and produces the desired response (response body, headers, and cookies). It requires certain generic notations for the inputs and outputs exchanged during end-to-end logical flow.

Content Integration Framework uses RQ type parameter to denote the type of input supplied to the service on its invocation. Here, the RS type parameter is used to denote either the type of object returned by the Functional service or the type of response body returned by the remote REST API invoked using RESTful approach. The purpose of RS might change based on where it is used, but it always indicates the return value of something.

RestService<RQ, RS>

Refer the `com.example.service.rest.ExampleCustomService` class from the `asset-integration-starter` project to understand the type parameters used in the `RestService` inteface. `RestService` is just a marker interface extended from `HttpService`. The definition of these type parameters is similar for the `HttpService` too.

- **RQ**

  A service requires an input to perform its operation. RQ corresponds to the type of input, or request, the service requires when invoked. The `com.example.service.rest.ExampleCustomService` takes an input of type `ServiceInput`. The same type parameter is used in the `ExecutionContext` object passed to all methods in the `RestService` or the `HttpService` interface. The input, or the request, object passed to the service, when invoked, is obtained by calling the `getRequest` method in the `ExecutionContext` object.

  ```
  @Override
   public HttpRequest buildRequest(ExecutionContext<ServiceInput> executionContext) {
    ServiceInput input = executionContext.getRequest();
    // Remaining implementation omitted for brevity
    }
  ```

- **RS**

  This parameter type corresponds to the type of response (post deserialization) received from the remote REST API. Service implementation chooses this parameter based on the kind of object it wants to work with in `transformResponse` method. If you look at the signature of the `transformResponse` method in the `com.example.service.rest.ExampleCustomService` class, you will see that the `ApiResponse` is supplied as the type argument to the `HttpResponse` class, which corresponds to the RS type parameter of the `RestService` interface.

  > ✏️ **Note:** Deserialization occurs according to the `Content-Type` header present in HTTP response received from REST API. The type used as the second generic argument to `RestService`, or the `HttpService`, must be appropriately annotated if Jackson or JAXB deserialization is expected.

FunctionalService<RQ, RS>

`FunctionalService` interface is analogous to the `java.util.function.Function` interface from the Standard Java Library. The type parameters of `FunctionalService` have similar semantics as the type parameters of `java.util.function.Function` interface.

- **RQ**

  Represents the type of input given to the service upon invocation.

- **RS**

  Represents the type of value returned by the service upon completion.

ServiceGateway<RQ, RS>

`ServiceGateway` is used as programmatic interface to services. `ServiceGateway` indicates the types of input & output/return values of the given service. Implementation of `getServiceInterface` method is mandatory for all services (except inbound HTTP service & Kafka consumer/listener services) & is supposed to return class object of subtype of `ServiceGateway` parameterized with appropriate input & output types. `ServiceGateways` & implementation for `getServiceInterface` already exists for all the standard services. All custom services implemented by the plugin must adhere to this contract. The definition of `com.hcl.unica.system.integration.service.gateway.ServiceGateway` is as follows:

```
public interface ServiceGateway<RQ, RS> {
  public RS execute(RQ request) throws ServiceExecutionException;
}
```

Semantics for the type parameter RQ is the same as mentioned earlier. The other type parameter, RS represents the output of the service that resides in the plugin. It does not represent the response received from remote REST API or any other target systems. For the `com.example.service.rest.ExampleCustomService` class, the `CustomServiceGateway` is defined as the child interface of `ServiceGateway` by using `ServiceInput` and `ServiceOutput` type arguments because the service receives an input of type `ServiceInput` and returns the value of type `ServiceOutput` on completion. For RESTful implementations, execution of `transformResponse` method marks the service completion in case of successful response from respective system.

> ✏️ **Note:**

> - `getServiceInterface` method in `com.example.service.rest.ExampleCustomService` class returns the class object of `CustomServiceGateway`. `ServiceGateway` interface (or its child interface) provides information about the input and the output of service implementation. `ServiceGateway` interface is further used to contain the reference of service instance and invoke its execution.
> - By obtaining reference to the `ServiceGateway` instance of any service thus implemented, execute(RQ request) method can be invoked to execute the service. Note that the execute method may throw the `ServiceExcecutionException` if anything goes wrong during service execution. Details on service invocation and exception handling will be provided in topics that follow.

ObjectEventInterpreterService<T>

`ObjectEventInterpreterService` is used for implementing Webhooks to listen to the events coming over HTTP from external systems. The `ObjectEventInterpreterService<T>` is subtype of `InboundHttpService<RQ, RS>`. The RQ & RS type parameters in `InboundHttpService` represents the type of deserialized request expected by the inbound HTTP service & the type of response (prior to serialization) returned by this service. `ObjectEventInterpreterService` is a specialized version of `InboundHttpService` wherein type parameter T is analogous to RQ (the type of deserialized request). Services implementing `ObjectEventInterpreterService` can only specify the request type since response is managed by the Content Integration Framework itself. For example, refer to the `com.example.service.rest.events.ExampleEventInterpreterService` service from asset-integration-starter project.

KafkaConsumerService<K, V, RS>

`KafkaConsumerService` is used for implementing Kafka event listener/interpreter services. The type parameters K & V represents the Key & Value of incoming Kafka message. For example, refer to the `com.example.service.kafka.events.consumer.ExampleKafkaEventInterpreterService` service from `asset-integration-starter` project.

AbstractKafkaProducerService<K, V>

`AbstractKafkaProducerService` is used for implementing Kafka message publisher services. The type parameter K & V represents the Key & Value of outgoing Kafka message. For example, refer to the `com.example.service.kafka.events.producer.ExampleKafkaProducerService` service from `asset-integration-starter` project.

I18n support

Content Integration Framework makes use of localized messages wherever applicable & possible. It implicitly looks for localized strings in certain cases. It expects certain standard keys to be present inside translation files. Likewise, it provides an API to explicitly obtain localized messages inside service implementation should there be such need.

I18n support is provided to the plugins by means of translation (properties) files. Plugins can maintain the translation files inside **src/main/resource/i18n** directory. Alternatively, **i18n** directory containing translation files can be kept under classpath on the application server. Translation files must be named in the following format – {SystemId}_Messages_{locale}.properties. For example, for a system with **Foo** as an identifier, following sort of translation files can be placed inside **i18n** directory:

- Foo_Messages.properties (default)
- Foo_Messages_de.properties
- Foo_Messages_en.properties
- Foo_Messages_es.properties
- Foo_Messages_fr.properties
- Foo_Messages_it.properties
- Foo_Messages_ja.properties
- Foo_Messages_ko.properties
- Foo_Messages_pt.properties
- Foo_Messages_ru.properties
- Foo_Messages_zh_CN.properties
- Foo_Messages_zh_TW.properties

Not all translation files are mandatory. Content Integration Framework looks for the properties file corresponding to the user locale. If it does not find the translation in that file, or the file itself is not present, then it falls back to the file corresponding to the system's locale. If no translation is found in this case as well (or the file itself is not found), then it falls back to the default translation file, Foo_Messages.properties. If the translation is found nowhere (either because of missing entry or missing file), then it uses the text appropriate for the given context. `asset-integration-starter` project is shipped with these sample translation files inside **src/main/resources/i18n** directory.

Service invocation

The `asset-integration-starter` project contains a `com.example.service.client.ExampleServiceClient` class to illustrate the service invocation.

The `ExampleServiceClient` class obtains reference to the `SystemGateway` object for the system represented by an identifier *Foo* by calling `SystemGatewayFactory.getSystemGateway` method with `Foo` as an argument. `SystemGatewayFactory.getSystemGateway` method thus gives a handle to any target system by specifying its *systemId*. Once the handle is obtained in terms of `SystemGateway` object, it can be used to invoke any service on the respective target system. The following is the corresponding code snippet from `ExampleServiceClient` class:

```
private SystemGateway systemGateway = SystemGatewayFactory.getSystemGateway("Foo");
```

Note: System identifier has been hard coded in this example only for the sake of clarity. For real implementations, avoid hard coding system identifiers for invoking the services from the same or any other plugin. Content Integration Framework supports aliased configurations of same system. Aliased configurations are made by specifying the desired system identifier, followed by a separator (space) and desired alias for the system. For example, Foo can be configured in Platform Configuration using any of the following identifiers – Foo, Foo Staging, Foo Prod, Foo Demo etc. More than one configuration can coexist for the same system by making use of aliased system identifiers. Aliased identifiers refer the same plugin at runtime but execute within separate context. Hence, it is important to invoke services within the right context by using the identifiers used in Platform configurations. Plugin can use

> following API to obtain the identifier thus used in Platform configuration. Using identifiers obtained from this API makes the plugin compatible for aliased configurations.

```
String systemId = executionContext.getSystemConfig().getIdentifier();
```

SystemGateway

The `com.hcl.unica.system.integration.service.gateway.SystemGateway` provides an overloaded method `executeService`, for executing any service on the target system. One version of this method offers a way to execute any service declared in service declaration files (`<ASSET_PICKER_HOME>/conf/custom-plugin-services.yml` and `<ASSET_PICKER_HOME>/conf/plugin-services.yml`) for the respective system. And the other version offers a way to execute an ad hoc HTTP call on the target system without declaring any explicit service for it in the service declaration file. The following are the two versions of the `executeService` method with their signatures:

- **<RQ, RS> RS executeService(String serviceName, RQ serviceInput, Class<? extends ServiceGateway<RQ, RS>> gatewayClass) throws ServiceExecutionException**

   This is a generic method and works with the type parameters RQ & RS. The significance of RQ & RS is same as mentioned earlier. This method helps to execute an already declared service. The `invocationDemo` method in `ExampleServiceClient` class demonstrates the use of this method. It accepts the following arguments:

   ◦ **String serviceName**

   This must be the name of service to be executed. Name of the service must exactly match with its corresponding declaration in service declaration file.

   ◦ **RQ serviceInput**

   This is an input to the service to be executed. The type parameter RQ represents the type of input required for the service being invoked.

   ◦ **Class<? extends ServiceGateway<RQ, RS>> gatewayClass**

   It must be same as the return value of `getServiceInterface` method in corresponding service implementation. It helps the Content Integration Framework to identify the right input for the service being executed and returns the output of desired type. The RQ and RS type parameters used for `gatewayClass` argument represents the type of input supplied on service invocation and the type of response returned by the service on completion, respectively.

   On successful completion, this method returns the object of type represented by the type parameter RS. Thus, the third argument to the `executeService` method, `gatewayClass`, governs the type of input that goes into the service and the type of value that service returns.

- **<T> HttpResponse<T> executeService(HttpRequest request, Class<T> expectedResponse) throws ServiceExecutionException**

   This is also a generic method, where the type parameter T represents the type of response expected out of the remote HTTP call. It helps to make an ad-hoc HTTP call to the target system without declaring an explicit service for it in service declaration file. The `adHocInvocationDemo` method in the `ExampleServiceClient` class demonstrates the use of this method. It accepts the following listed arguments:

- ◦ **HttpRequest request**

  This must be an object of `com.hcl.unica.system.model.request.HttpRequest` class. `HttpRequest` provides a builder interface for constructing the object with required details. This object essentially encapsulates the details required for making an HTTP call, such as absolute URL, HTTP request method, HTTP request headers & HTTP request body or HTTP request payload.

- ◦ **Class<T> expectedResponse**

  This must tell the type of response expected from remote URL. Jackson and JAXB types can also be used. Deserialization of JSON/XML will happen automatically in such case.

On successful completion, this method returns the object `com.hcl.unica.system.model.response.HttpResponse`, encapsulating the response object from the remote call. The type of response encapsulated by the `HttpResponse` will be the same as the `expectedResponse` argument to the `executeService` method. The `HttpResponse` object gives access to the HTTP response status code, response headers, and response cookies, in addition to the response payload.

Both versions of the `executeService` method can throw the `com.hcl.unica.system.integration.exception.ServiceExecutionException` or one of its subtypes if anything goes wrong during service execution. The object of this exception can be consulted for the immediate cause of service execution failure. Likewise, if the invoked service represents a REST/HTTP service (ad-hoc service invocations are always HTTP calls), and the failure occurs out of HTTP interaction, an optional `HttpResponse` object can also be obtained from the exception. In such cases, the `HttpServiceExecutionException` is thrown by the `executeService` methods. The presence of `HttpResponse` depends on whether the HTTP interaction happened or not. The `HttpServiceExecutionException` might be received because of an exception in any logic executed prior to the actual HTTP call, such as `buildRequest` method in a declared service.

The `executeService` method can also throw a `SystemNotFoundException` if the plugin for the specified target system is not present, or the corresponding system is not onboarded in Unica Platform. Similarly, it can throw a ServiceNotFoundException if the specified service is either not declared in service declaration file or not implemented by the plugin.

> 📝 **Note:**
>
> - You will observe that the type of the input to the `custom-service` is same as the type used for service implementation in the `com.example.service.rest.ExampleCustomService` class or the `com.example.service.functional.CustomService` class. The type of output is same as the one used for defining `CustomServiceGateway` interface whose class object is returned from `getServiceInterface` method in both versions of `CustomService` implementations.
> - The `com.example.service.rest.ExampleCustomService`class and the `com.example.service.functional.ExampleCustomService` class represents the same service implemented with two different approaches. The service declaration files in `asset-integration-starter` project namely the `META-INF/rest-content-services.yml` and the `META-INF/functional-content-services.yml` have an entry for `custom-service` pointing to the respective versions of the `factoryClass`. These two

versions are provided only for illustration purpose. For all practical purposes, only one version of the service implementation is expected by the Content Integration Framework. Irrespective of the approach used for service implementation, the method for service invocation remains the same.

Multi-partitioned clients

From the perspective of service implementation, the `ExecutionContext` and `SystemConfig` objects, passed to various callback methods, contain client application and partition specific information. And from the perspective of service invocation, services executed using `executeService` method, from the `SystemGateway` class, runs against the system configured for the right client application and the partition of the user accessing Unica Content Integration. Hence, neither the implementation nor the invoker need to work with partitioning and other contextual details, explicitly. Content Integration Framework handles it automatically.

Execution context

Almost every method in service implementation contract receives an instance of `com.hcl.unica.system.model.request.ExecutionContext` class.

This object contains all the contextual information that is necessary for a service to perform its operation. The following are the methods in ExecutionContext class, which can be used to obtain various types of information during service execution:

- **T getRequest()**

  This method can be used to obtain the input, or request, object passed to the service when it is executed using `executeService` method discussed in Service invocation on page 23 (The T return type is the type parameter corresponding to the type parameter RQ used for defining the service).

- **Map<String, Object> getAttributes()**

  Returns a Map which can be used to store and retrieve custom attributes during service execution. It is useful for carrying execution specific temporary information across multiple callbacks. For example, if the implementation of `buildRequest` method from the `RestService` interface or `HttpService` interface needs to share some information with `transformResponse` method, it can share it using this attribute Map.

  It is important to note that Content Integration Framework creates a separate instance of `ExecutionContext` for each individual service invocation. Hence, context attributes cannot be shared across multiple service executions. Their scope is limited to individual service execution.

- **ServiceConfig getServiceConfig()**

  This method returns an instance of `com.hcl.unica.system.integration.config.ServiceConfig` class. `ServiceConfig` object holds the configurations made in service declaration file for the respective service.

- **SystemConfig getSystemConfig()**

  This method returns an instance of `com.hcl.unica.system.integration.config.SystemConfig` class. `SystemConfig` object contains all the configurations made in Unica Platform for the target system. In case of multi-partitioned configurations, this object will be appropriately populated by Content Integration Framework to hold partition-specific

configuration for the concerned client application. To know the various system configuration settings in Unica Platform, see *Unica Content Integration Administrator's Guide.*

- **void setAttributes(Map<String, Object>)**

  This method can be used to set attributes in `ExecutionContext`, which can be then be obtained in other areas of the service implementation. This is useful for sharing custom contextual information during service execution. Scope of the attributes stored in execution context is limited to the current execution flow only. Attributes cannot be shared across multiple execution flows of the same service.

- **Locale getUserLocale ()**

  This method can be used to obtain signed in user's locale.

- **String localized(String key, String defaultText)**

  Returns the localized text for the given key. If translation does not exist for the given key in plugin's I18n resource bundles, then returns the default text.

InboundHttpRequestContext<T>

This class inherits from ExecutionContext<T> and holds the context information relevant to the inbound RESTful service. In addition to the methods inherited from ExecutionContext, it provides following methods for obtaining HTTP request details:

- **Map<String, Object> getPathVariables()**

  Inbound RESTful services can use path variables while defining the endpoint signature. Path variables allow designing dynamic URLs, wherein certain portions of the URL carry value specific to the given request context. This method can be used to obtain path variables thus defined & their values.

- **Map<String, List<Object>> getRequestParams()**

  This method can be used to obtain request parameters & their values.

- **Map<String, Object> getRequestHeaders()**

  This method can be used to obtain request headers & their values.

User data source

Unica Platform uses user data sources to store sensitive information, such as API credentials, security tokens, database user credentials, etc. Plugins often need to store such configuration details. Content Integration provides the relevant configuration to specify the name of user data source and the associated Unica user while onboarding systems using Unica Platform configuration.

Use the `ExecutionContext` to obtain applicable user data source (credentials) by navigating through `SystemConfig` object:

```
executionContext.getSystemConfig().getDataSourceCredentials()
```

The `DataSourceCredentials` object returned by the `getDataSourceCredentials` method contains the selected data source based on the strategy set up for **User credentials** in Platform configuration. Hence, plugins need not make any logical decision pertaining to the right selection of the user data source.

Likewise, `SystemConfig` object also provides an overloaded version of `getDataSourceCredentials`, `getDataSourceCredentials`(String username, String dataSourceName), which accepts the username & the name of data

source to be looked up in the given user's account. This method can be used for obtaining user data sources in other similar logical contexts should there be any need.

Standard services and specialized types

The plugin developer needs to implement one of the following interfaces to create an individual service - `RestService/ HttpService`, `FunctionalService`, `KafkaConsumerService`, `KafkaProducerService`.

The Content Integration Framework leverages this design and defines service classes for all the standard services. The standard-integrations.jar provided as part of Content Integration SDK provides specialized versions of `RestService` and `FunctionalService` for each standard service to facilitate their implementation using RESTful as well as Functional approach. These specialized classes are discussed further in this guide under Specialized types on page 31 section.

Invocation of standard services

Once implemented & declared in service declaration file, Content Integration Framework invokes the standard services in following scenarios provided that respective system is configured appropriately in Unica Platform configuration:

- **Simple Search (`simple-search`)**

  Whenever Content Integration Framework receives content or asset search request from its client application against target system, it invokes the `simple-search` service implemented for respective system. Content Integration Framework provides necessary input to the `simple-search` service upon invocation. Search items received from `simple-search` service are then returned to the client application. Identification of the target system happens based on the `systemId` property used in the service declaration file and the corresponding **System Identifier** setting in Unica Platform that is populated during the target system onboarding. This service must be implemented by the plugin, else the content search request ends up in 404 response to the client application.

  The search result produced by this service can be either paginated or unpaginated. Presence or absence of support for paginated result should be clearly indicated using paginatedSearch property under systems section in service declaration file as explained in the Service declaration file on page 5 topic.

- **Resource Loader (`resource-loader`)**

  The `resource-loader` service is executed by the Content Integration Framework only when indirect (or authenticated) access needs to be made to the search item on the target system. Configuration can be made in Unica Platform to indicate whether contents can be accessed directly (anonymously) from the target system or not. For more information about system configurations, see *Unica Content Integration Administration Guide*. Content Integration Framework provides a default `resource-loader` service to each system. The default `resource-loader` service simply loads the web resources from the target system by supplying necessary authorization details, if applicable. Plugins may choose to override the default `resource-loader` service and include their own implementation by extending the out-of-the-box implementation. Content download and content rendition might fail if the required overridden `resource-loader` implementation is missing

- **List Category Folders (list-category-folders)**

  If this service is implemented, system assumes that content categories are organized in folders. It is expected to provide top/root level folders as well as sub-folders of a particular parent folder as and when requested for category navigation. Only one level of folder list is expected in single execution, entire folder hierarchy need not be provided.

If this service is implemented, it is imperative to implement the `list-content-categories` service accordingly so that categories belonging to a specific folder can be retrieved at a time.

- **List Content Categories (`list-content-categories`)**

If implemented, this service is invoked for fetching the list of content categories supported by the respective system. Content categories are used to narrow down the content search/selection within a particular category. There may be other use cases pertaining to these categories in future releases of Unica Content Integration.

If content categories are organized in folders, then it is imperative to implement `list-category-folders` service to return the folder hierarchy. In such case, `list-content-categories` service receives identifier of the folder for which category list needs to be returned in single execution.

This is an optional service and absence of its implementation does not impact content searchability in Content Picker. Other alternatives are used instead to generate the list of supported content categories in the absence of this service, that is `supportedContentTypes` standard parameter for `simple-search` service in service declaration file or `getSupportedContentTypes()` method in `simple-search` service implementation. These alternatives provide way for static configuration of categories, whereas `list-content-categories` service provides opportunity to fetch categories dynamically on each user request.

- **List Folders (`list-folders`)**

This service is used to facilitate content navigation along with the `list-contents` service. In addition to the content search, content can also be located by navigating through the hierarchy of folders (or any other similar concept in respective system). If this service is implemented, it is expected to provide top/root level folders as well as sub-folders of a particular parent folder as and when requested during content navigation. Only one level of folder list is expected in single execution. Entire folder hierarchy need not be provided. If this service is implemented, it is imperative to implement the list-contents service as well to turn the content navigation feature on.

This is an optional service and absence of its implementation does not impact content searchability in Content Picker. However, content navigation is disabled in Content Picker UI if this service is not implemented.

- **List Contents (`list-contents`)**

This service is used to facilitate content navigation along with the `list-folders` service. If implemented, this service is expected to provide the list of contents belonging to a particular folder. List can be either paginated or unpaginated. Presence or absence of support for paginated list should be clearly indicated using `paginatedList` property under systems section in service declaration file as explained in the topic.

If this service is implemented, it is imperative to implement the `list-folders` service as well to turn the content navigation feature on.

This is an optional service and absence of its implementation does not impact content searchability in Content Picker. However, content navigation is disabled in Content Picker UI if this service is not implemented.

- **Get Content Details (`get-content-details`)**

Any content searched using `simple-search` service or listed using `list-contents` service can be selected and used for various use cases in Unica applications. Such use cases might demand the details of already chosen content at later point of time. One such example is Content Preview feature in Centralized Offer Management, wherein details of already linked content with offer attribute are shown. Whenever Unica applications need details of any individual content, the get-content-details service is invoked by supplying the unique identifier of the required content.

This is an optional service and absence of its implementation does not impact content searchability in Content Picker. However, subsequent user requests for fetching details of a content will not be served if this service is not implemented.

- **Get Object Schema (`get-object-schema`)**

This service is invoked by Unica applications to fetch the meta information about various attributes present in the content. Based on the strictness of content categorization as discussed in Service declaration file on page 5, this service may choose to generate individual schema for each category of contents or generate a master schema comprising the attributes of all category of contents. In case of strict categorization of content, separate schema is preferable. Whereas for leniently categorized contents, a master schema is advisable. The meta information about each content attribute must include the details such as the type and format of the value it holds and a unique identifier to uniquely identify that attribute in the given system. Syntax & semantics associated with this service are discussed in detail under Specialized types on page 31 section.

The object schema thus generated by this service is used by Unica applications for mapping individual attributes of contents/domain objects with individual attributes of Unica objects (for example, Offers in Centralized Offer Management, Data Definitions in Journey). Such object mappings allow Content Integration Framework to derive attribute values for one object from the other. Furthermore, Content Integration Framework leverages this capability to synchronize objects automatically whenever object events are received.

- **Get Cognitive Analysis (`get-cognitive-analysis`)**

This service is invoked to attempt cognitive analysis of an image and fetch the cognitive details accordingly. It is invoked only if respective system is configured as the Preferred cognitive service provider in Platform Configuration. For more information, see *Unica Content Integration Installation and Configuration Guide*.

This is an optional service and absence of its implementation does not impact content searchability or any other feature in Content Picker. However, cognitive tagging feature is disabled in Centralized Offer Management if this service is not available.

- **Object extension service**

Object extension services are presently used by Centralized Offer Management to add custom actions to the Offers. Content Integration Framework does not mandate any standard service name for Object extension service. Plugin can choose any suitable name for it and define more than one such services, if required. The core responsibility of this service is to accept desired request attributes on invocation and use those attributes to enhance target object's capability (Offer in case of Centralized Offer Management) by either performing some supporting operation or providing values for other attributes of the target object. As mentioned in Standard services on page 8 section, implementation of get-object-schema service must be enhanced to address the schema generation requirement for the expected request object as well as for the response object returned by this service. Request object's schema is used for mapping source object's attributes to custom action's request object attributes. Whereas response object's schema is used for mapping its attributes with target object's attributes. For more information on setting up custom actions in Centralized Offer Management, please refer to the Centralized Offer Management Administrator Guide. Once the custom action is defined, its execution triggers the invocation of respective extension service. Syntax & semantics of this service are further discussed in Specialized types section.

This is an optional service and absence of its implementation does not impact content search ability in Content Picker. However, the **Custom Actions** feature in Centralized Offer Management becomes unavailable for the respective system if this service is not implemented.

- **Object event interpreter service (Webhook)**

Event interpreter service is invoked whenever there is an incoming HTTP request on the webhook exposed by the respective event interpreter service. As of release 12.1.2, **Unica Centralized Offer Management** can listen to the events received via such webhooks and synchronize Offers using the associated content details. For more information on auto synchronization of Offers, see Unica Centralized Offer Management administrator guide. Likewise, Unica Journey can also receive events via Webhooks using its REST Entry Sources. Events received via Webhooks are processed based on the object mappings set up using object schemas returned by `get-object-schema` service.

This is an optional service and absence of its implementation does not impact content searchability or any other feature in Content Picker. However, if respective external system is interested in publishing content lifecycle events to Unica products and/or Unica features associated with content events are desired, then implementation of this service is necessary to aid event processing.

- **Object event interpreter service (Kafka listener)**

Kafka event listeners are another way of receiving & interpreting events from external systems. All dependencies, characteristics & semantics of Kafka event listeners are same as Webhooks.

Specialized types

The following are the specialized derivatives of `RestService`, `HttpService`, `FunctionalService`, `InboundHttpService`, `KafkaConsumerService` & `KafkaProducerService` interfaces, and their related types for all the standard services. Use the `asset-integration-starter` project to implement the details mentioned in the following topics:

Derivatives of RestService

Derivatives of RestService interface facilitates creation of RESTful implementation of standard services.

Simple search (`simple-search`)

The following are the specialized interfaces and classes available for the `simple-search` service:

- `com.hcl.unica.system.integration.service.search.RestSearchService`

  The `com.example.service.rest.SimpleSearchService` class in `asset-integration-starter` project is a quick starter implementation for RESTful `simple-search` service. Its parent is `com.hcl.unica.system.integration.service.search.RestSearchService` class.

The `RestSearchService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case it is `SimpleSearchResponse` class defined inside the `asset-integration-starter` project.

`RestSearchService` class implements `RestService` interface and defines the `SearchRequest` class as the type argument RQ for `RestService`. Thus, the object of `SearchRequest` becomes input to all the `simple-search` services (same input is used for Functional counterpart of simple-search as well). `SearchRequest` class is part of the Content Integration SDK.

In addition to defining the input type for the `simple-search` service, `RestSearchService` class also overrides the `transformResponse` method and defines return value of this method to be of `ContentPage` type. `ContentPage` is also part of the Content Integration SDK and encapsulates the search result and associated pagination details.

The plugin must extend its `simple-search` implementation from the service `com.hcl.unica.system.integration.service.search.RestSearchService` to be recognized as a `simple-search` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from, for the `simple-search` services implemented using the Functional approach).

`RestSearchService` extends from `com.hcl.unica.system.integration.service.search .AbstractSearchService` abstract class.

We recommend looking at `com.aem.service.AemSimpleSearchService` class from the `aem-integration` project to know more about how the `SearchRequest` class and the `ContentPage` class are used during service implementation.

Adhering to the contract of `Presentable` interface while populating list of contents in `ContentPage` is a crucial part of this service implementation. `Presentable` interface is covered in more detail in subsequent section.

- `com.hcl.unica.system.integration.service.search.AbstractSearchService`

This is a common base class for RESTful as well as Functional `simple-search` implementations. So, the details of this class also apply to the Functional implementation of `simple-search`.

This class defines the `com.hcl.unica.system.integration.service.gateway.SimpleSearchServiceGateway` interface as the service gateway for the `simple-search` service. A closer look at this interface tells us that the `simple-search` takes the `SearchRequest` object and returns the `ContentPage` object.

In addition to defining the service interface for `simple-search`, it introduces one more method for the `simple-search` service, named `getSupportedContentTypes`. Every `simple-search` implementation can optionally override and implement this method. Please note that this method is very `simple-search` specific and has nothing to do with other standard and custom services. The signature of this method is as follows:

```
public Map<String, String> getSupportedContentTypes();
```

Implementation of this method returns a `Map<String, String>` representing the supported categories of contents that can be searched in the target system. There is no specific semantic associated with the entries in this Map. It can be any meaningful key-value pair. It acts as a filter for client application during the search operation. As of current implementation of Unica Content Integration, this Map is used to populate entries in a drop down, wherein keys of the Map become values of the options, and values of the Map become display labels for the options. Thus, keys can carry internal names, or identifiers, and values should be readable and meaningful texts. If the user needs to search any specific type of content, he can choose one or more options from the supported types. In such case, `simple-search` service receives a set of keys corresponding to the values chosen by the user. Set of keys received from

the client application can be obtained from `ExecutionContext` object by navigating through the `getRequest` method and then calling `getTypes()` on it. The `simple-search` implementation deals with these set of keys, as per the target system's programming interface, and filters the search items accordingly.

Standard service parameter - supportedContentTypes

Overriding `getSupportedContentTypes` method is recommended only if the Map needs to be generated dynamically. Content Integration Framework provides an alternate approach to statically define this Map using a standard service parameter called `supportedContentTypes`, configured under `params` element in the service declaration file. For example, refer the `simple-search` service declaration for AEM and WCM inside `<ASSET_PICKER_HOME>/conf/plugin-services.yml` file.

List content categories (`list-content-categories`)

The following are the specialized interfaces and classes available for the `list-content-categories` service:

- `com.hcl.unica.system.integration.service.content.categories.list. RestContentCategoriesListService`

The `com.example.service.rest.` ExampleContentCategoryListingService class in asset-integration-starter project is a quick starter for RESTful list-content-categories service. ExampleContentCategoryListingService class extends from RestContentCategoriesListService class.

The `RestContentCategoriesListService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case it is specified as `List<ContentCategoryDetails>` for the sake of example.

`RestContentCategoriesListService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.content.categories.`ContentCategoryListRequest class as the type argument RQ for RestService. Thus, the object of ContentCategoryListRequest becomes input to all the `list-content-categories` services (same input is used for Functional counterpart of list-content-categories as well). `ContentCategoryListRequest` object should be consulted for parent category ID as well as for parent folder ID if categories are hierarchical and/or maintained in folder hierarchy.

In addition to defining the input type for the list-content-categories service, RestContentCategoriesListService class also overrides the transformResponse method and mandates the return value of this method to be an object of `List<ContentCategory>` type. `ContentCategory` class is part of Content Integration SDK.

The plugin must extend the implementation of list-content-categories service from com.hcl.unica.system.integration.service.content.categories.list. RestContentCategoriesListService class to be recognized as a valid list-content-categories service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

`RestContentCategoriesListService` extends from

```
com.hcl.unica.system.integration.service.content.categories.list.AbstractContentCategoriesListService
  class
```

.

- `com.hcl.unica.system.integration.service.content.categories.list.AbstractContentCategoriesListService`

This is a common base class for `RESTful` as well as Functional implementations of `list-content-categories` service. So, the details covered herein applies to Functional version of `list-content-categories` as well.

This class defines the `com.hcl.unica.system.integration.service.gateway.ContentCategoriesListServiceGateway` interface as the service gateway for the `list-content-categories` service. This interface extends from `com.hcl.unica.system.integration.service.gateway.ServiceGateway` interface and mandates the `ContentCategoryListRequest & List<ContentCategory>` objects to be the input and output types for the `list-content-categories` service.

List folders (`list-folders`)

The following are the specialized interfaces and classes available for the `list-folders` service:

- `com.hcl.unica.system.integration.service.folder.list.RestFolderListService`

  The `com.aem.service.AemFolderListService` class in `aem-integration` project is a reference implementation for RESTful list-folders service. `AemFolderListService` class extends from `RestFolderListService` class.

  The `RestFolderListService` class has a type parameter **RS**, which represents the type of response (post deserialization) received from the remote REST API. In this case it is `SimpleSearchResponse` class defined inside the `aem-integration` project.

  `RestFolderListService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.folder.list.FolderListRequest` class as the type argument RQ for `RestService`. Thus, the object of `FolderListRequest` becomes input to all the `list-folders` services (same input is used for Functional counterpart of `list-folders` as well).

  In addition to defining the input type for the `list-folders` service, `RestFolderListService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of `List<Folder>` type. Folder is a standard type defined in Content Integration SDK.

  The plugin must extend the implementation of `list-folders` service from `com.hcl.unica.system.integration.service.folder.list.RestFolderListService` class to be recognized as a valid list-folders service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

  `RestFolderListService` extends from `com.hcl.unica.system.integration.service.folder.list.AbstractFolderListService` class.

- `com.hcl.unica.system.integration.service.folder.list. AbstractFolderListService`

  This is a common base class for RESTful as well as Functional implementations of `list-folders` service. So, the details covered herein applies to Functional version of `list-folders` as well.

  This class defines the `com.hcl.unica.system.integration.service.gateway.FolderListServiceGateway` interface as the service gateway for the `list-folders` service. This interface extends from `com.hcl.unica.system.integration.service.gateway.ServiceGateway` interface and mandates the `FolderListRequest` and `List<Folder>` objects to be the input and output types for the `list-folders` service.

List category folders (`list-category-folders`)

The specialized types used for implementing list-category-folders are exactly same as the ones used for list-folders.

List contents (`list-contents`)

The following are the specialized interfaces and classes available for the list-contents service:

- `com.hcl.unica.system.integration.service.content.list.RestContentListService`

  The `com.aem.service.AemContentListServiceclass` in `aem-integration` project is a reference implementation for RESTful `list-contents` service. `AemContentListServiceclass` class extends from `RestContentListService` class.

  The `RestContentListService` class has a type parameter **RS**, which represents the type of response (post deserialization) received from the remote REST API. In this case it is `SimpleSearchResponse` class defined inside the `aem-integration` project.

  `RestContentListService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.content.list.ContentListRequest` class as the type argument **RQ** for `RestService`. Thus, the object of `ContentListRequest` becomes input to all the `list-contents` services (same input is used for Functional counterpart of `list-contents` as well).

  In addition to defining the input type for the `list-contents` service, `RestContentListService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of `ContentPage` type. This return type is same as the one used for `simple-search` service. `ContentPage` is a standard type defined in Content Integration SDK.

  The plugin must extend the implementation of `list-contents` service from `com.hcl.unica.system.integration.service.content.list.RestContentListService` class to be recognized as a valid `list-contents` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

  `RestContentListService` extends from `com.hcl.unica.system.integration.service.content.list.AbstractContentListService` class.

- `com.hcl.unica.system.integration.service.content.list.AbstractContentListService`

  This is a common base class for RESTful as well as Functional implementations of `list-contents` service. So, the details covered herein applies to Functional version of `list-contents` as well.

  This class defines the `com.hcl.unica.system.integration.service.gateway.ContentListServiceGateway` interface as the service gateway for the `list-contents` service. This interface extends from `com.hcl.unica.system.integration.service.gateway.ServiceGateway` interface and mandates the `ContentListRequest` and `ContentPage` objects to be the input and output types for the `list-contents` service.

Get content details (`get-content-details`)

The following are the specialized interfaces and classes available for the get-content-details service:

- `com.hcl.unica.system.integration.service.content.details.RestContentDetailsService`

  The `com.aem.service.AemObjectDetailsService` class in `aem-integration` project is a reference implementation for RESTful `get-content-details` service. `AemObjectDetailsService` class extends from `RestContentDetailsService` class.

  The `RestContentDetailsService` class has a type parameter **RS**, which represents the type of response (post deserialization) received from the remote REST API. In this case it is `SimpleSearchResponse` class defined inside the `aem-integration` project.

  `RestContentDetailsService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.content.details.ContentDetailsRequest` class as the type argument **RQ** for `RestService`. Thus, the object of `ContentDetailsRequest` becomes input to all the `get-content-details` services (same input is used for Functional counterpart of `get-content-details` as well).

  In addition to defining the input type for the `get-content-details` service, `RestContentDetailsService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of Presentable type.

  The plugin must extend the implementation of `get-content-details` service from `com.hcl.unica.system.integration.service.content.details.RestContentDetailsService` class to be recognized as a valid `get-content-details` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

  `RestContentDetailsService` extends from the `com.hcl.unica.system.integration.service.content.details.AbstractContentDetailsService` class.

- `com.hcl.unica.system.integration.service.content.details.AbstractContentDetailsService`

  This is a common base class for RESTful as well as Functional implementations of `get-content-details` service. So, the details covered herein applies to Functional version of `get-content-details` as well.

  This class defines the `com.hcl.unica.system.integration.service.gateway.ContentDetailsServiceGateway` interface as the service gateway for the `get-content-details` service. `ServiceGateways` are the means to programmatically define input and output types of the service and facilitate invocation of the services. A closer look at this interface tells us that the `get-content-details` service accepts the `ContentDetailsRequest` object and returns a Presentable object.

Get cognitive analysis (get-cognitive-analysis)

The following are the specialized interfaces and classes available for the get-cognitive-analysis service:

- `com.hcl.unica.system.integration.service.cognitive.analysis.RestCognitiveAnalysisService`

  The `com.example.service.rest.ExampleCognitiveAnalysisService` in `asset-integration-starter` project is a quick starter implementation for RESTful `get-cognitive-analysis` service. `ExampleCognitiveAnalysisService` in class extends from `RestCognitiveAnalysisService` class.

  The `RestCognitiveAnalysisService` class has a type parameter **RS**, which represents the type of response (post deserialization) received from the remote REST API. In this case it is `CognitiveDetails` class defined inside the `asset-integration-starter` project.

`RestCognitiveAnalysisService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.cognitive.analysis.CognitiveAnalysisRequest` class as the type argument **RQ** for `RestService`. Thus, the object of `CognitiveAnalysisRequest` becomes input to all the `get-cognitive-analysis` services (same input is used for Functional counterpart as well).

In addition to defining the input type for the `get-cognitive-analysis` service, `RestCognitiveAnalysisService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of `com.hcl.unica.system.model.response.cognitive.analysis.CognitiveAnalysis` type. `CognitiveAnalysis` is a standard type defined in Content Integration SDK.

The plugin must extend the implementation of `get-cognitive-analysis` service from `com.hcl.unica.system.integration.service.cognitive.analysis.RestCognitiveAnalysisService` class to be recognized as a valid `get-cognitive-analysis` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

`RestCognitiveAnalysisService` extends from `com.hcl.unica.system.integration.service.cognitive.analysis.AbstractCognitiveAnalysisService` class.

- `com.hcl.unica.system.integration.service.cognitive.analysis.AbstractCognitiveAnalysisService`

  This is a common base class for RESTful as well as Functional implementations of `get-cognitive-analysis` service. So, the details covered herein applies to Functional version of `get-cognitive-analysis` as well.

  This class defines the `com.hcl.unica.system.integration.service.gateway.CognitiveAnalysisServiceGateway` interface as the service gateway for the `get-cognitive-analysis` service. This interface extends from `com.hcl.unica.system.integration.service.gateway.ServiceGateway` interface and mandates the `CognitiveAnalysisRequest` and `CognitiveAnalysis` objects to be the input and output types for the `get-cognitive-analysis` service.

Object extension service

The following are the specialized interfaces and classes available for the object extension service:

- `com.hcl.unica.system.integration.service.object.extension.RestObjectExtensionService`

  The `com.example.service.rest.ExampleObjectExtensionService` in asset-integration-starter project is a quick starter implementation for RESTful object extension service. `ExampleObjectExtensionService` class extends from `RestObjectExtensionService` class.

  The `RestObjectExtensionService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case, it is CognitiveDetails class defined inside the `asset-integration-starter` project.

  `RestObjectExtensionService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.object.extension.ObjectExtensionRequest` class as the type argument RQ for `RestService`. Thus, the object of `ObjectExtensionRequest` becomes input to object extension services (same input is used for Functional counterpart as well). `ObjectExtensionRequest` is discussed in subsequent section.

  In addition to defining the input type for object extension service, `RestObjectExtensionService` class also overrides the `transformResponse` method and allows returning any type of object.

The plugin must extend the implementation of object extension service from `com.hcl.unica.system.integration.service.object.extension.RestObjectExtensionService` class to be recognized as a valid object extension service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

`RestObjectExtensionService` extends from `com.hcl.unica.system.integration.service.object.extension.AbstractObjectExtensionService` class.

- `com.hcl.unica.system.model.request.object.extension.ObjectExtensionRequest`

The `ObjectExtensionRequest` provides and API to retrieve values of request attributes. The core principle of object extension service is to accept desired request attributes, perform designated operation and optionally return with a response object. Request attribute values are obtained from the object whose capabilities need to be extended by using the object extension service being implemented. Implementation of object extension service must be request object agnostic. It is the responsibility of Content Integration Framework to extract request attributes from the object to be extended and supply it to the object extension service by means of `ObjectExtensionRequest`. Thus, object extension service must be able to extend the capability of any object if the object being extended contains the required request attributes for the object extension service. `ObjectExtensionRequest` provides following methods for obtaining request attribute values –

- `Collection<AttributeValue> getAttributeValues()` – This method returns a collection of all request attribute values.

  Attribute values are wrapped inside `AttributeValue` object, which provides API for reading the value of attribute. It provides a convenience method, as(Class<T> type), for obtaining type cast value of the attribute. Likewise, `getValue()` method can be used to read the uncast value.

  Furthermore, `AttributeValue` class provides `getAttributeSchema()` method to obtain the meta information about respective attribute. `getAttributeSchema()` returns an object of `AttributeSchema` containing the required meta information about attribute. The most important methods in `AttributeSchema` are `getType()` & `getFormat()`. These methods can be consulted to check if the attribute is of desired type. As of 12.1.1 release of Content Integration Framework, object extension service may receive values for the attributes which are not of required type. Hence, extension service must consult the type & format of the attribute before using its value. Service implementation must filter the required attribute values by itself. Refer to the `buildRequest` method implementation in `com.example.service.rest.ExampleObjectExtensionService` to learn how this can be achieved. This example implementation tries to extract first URL & first Text attribute from the request.

- `Map<String, AttributeValue> getAttributes()` - From release 12.1.2 onward, request attributes of object extension service can also be mapped to source object's attributes. get-object-schema service must be enhanced in such case to generate schema for request object. Following standard service parameter must be set to true if request attribute mapping is desired for the given object extension service. If this parameter is missing, it is assumed to be false by default. Likewise, requirement of response attribute mapping can also be indicated using attributeMappingSupport | response parameter. By default, Content Integration Framework assumes value for response to be true.

```
params:
  attributeMappingSupport:
    request: true
```

```
        response: true
```

`getAttributes()` method returns map of attribute values, wherein values are organized by respective attribute identifiers. The Map object returned by this method is useful for retrieving attribute value using its identifier (as used in object schema).

Object extension service can override void `validateRequest(ExecutionContext<ObjectExtensionRequest> executionContext`) method to probe request attributes before performing core operation. `InvalidServiceRequestException` can be raised If desired attributes are not found in the `ObjectExtensionRequest`. Refer to the Validation support on page 63

- `boolean isStronglyBound()` — Object extension services written prior to 12.1.2 may still receive unmapped requests if user hasn't changed the request mapping for such services. This method can be consulted to check if request object attribute mapping exists for the given execution request. This method returns true if object extension service does have request object attribute mapping, false otherwise. If this method returns true, service can look for its request object attributes by their identifiers in the Map returned by `getAttributes()` method. Else, it must find out required type of attributes the way `ExampleObjectExtensionService` does in `asset-integration-starter` project.

- `com.hcl.unica.system.integration.service.object.extension.AbstractObjectExtensionService`

This is a common base class for RESTful as well as Functional implementations of object extension service. So, the details covered herein applies to Functional version as well.

This class defines the `com.hcl.unica.system.integration.service.gateway.ObjectExtensionServiceGateway` interface as the service gateway for object extension service. This interface extends from `com.hcl.unica.system.integration.service.gateway.ServiceGateway` interface and mandates the `ObjectExtensionRequest` to be the input type for object extension service. Object extension service can return any type of object.

Service title & summary

Object extension services are listed as **Custom actions** in **Centralized Offer Management**. The title/label for object extension service is derived from I18n resource bundles using the key same as service name declared for respective object extension service. Likewise, a brief summary is also shown for the service to help users working with it. Summary is maintained in I18n resource bundles using `{service-name}.summary` key. For example, check `example-object-extension-service` & `example-object-extension-service.summary` keys in `asset-integration-starter\src\main\resources\i18n\Foo_Messages.properties` file.

Collaboration services

- **Channel creation service (`collaboration.create-channel`)**
  - The following are the specialized interfaces and classes available for the collaboration.create-channel service:
    - `com.hcl.unica.system.integration.service.collab.channel.create.RestChannelCreationService` —

      The `com.example.service.rest.collaboration.ExampleChannelCreationService` in `asset-integration-starter` project is a quick starter implementation for RESTful `collaboration.create-channel` service. `ExampleChannelCreationService` class extends from `RestChannelCreationService` class.

The `RestChannelCreationService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case it is JsonNode class defined inside the `ExampleChannelCreationService` class.

`RestChannelCreationService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.collab.channel.CollaborationChannelCreationRequest` class as the type argument RQ for `RestService`. Thus, the object of `CollaborationChannelCreationRequest` becomes input to all the `collaboration.create-channel` services (same input is used for Functional counterpart as well). `CollaborationChannelCreationRequest` carries the name of channel to be created.

In addition to defining the input type for the `collaboration.create-channel` service, `RestChannelCreationService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of `com.hcl.unica.system.model.response.collab.channel.CollaborationChannelDetails` type. `CollaborationChannelDetails` must carry the identifier of the newly created channel in addition to its name. It can optionally contain URL to the channel page.

Plugin must extend the implementation of `collaboration.create-channel` service from `RestChannelCreationService` class to be recognized as a valid collaboration.create-channel service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

- **Channel modification service (`collaboration.update-channel`)**

  The following are the specialized interfaces and classes available for the `collaboration.update-channel` service:

  ◦ `com.hcl.unica.system.integration.service.collab.channel.update.RestChannelModificationService` –

  The `com.example.service.rest.collaboration.ExampleChannelModificationService` in `asset-integration-starter` project is a quick starter implementation for RESTful `collaboration.update-channel` service. `ExampleChannelModificationService` class extends from `RestChannelModificationService` class.

  The `RestChannelModificationService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case it is JsonNode class defined inside the `ExampleChannelModificationService` class.

  `RestChannelModificationService` class implements RestService interface and defines the `com.hcl.unica.system.model.request.collab.channel.CollaborationChannelModificationRequest` class as the type argument RQ for RestService. Thus, the object of `CollaborationChannelModificationRequest` becomes input to all the collaboration.update-channel services (same input is used for Functional counterpart as well). `CollaborationChannelModificationRequest` carries the identifier of the channel to be updated along with the new channel name.

  In addition to defining the input type for the `collaboration.update-channel` service, `RestChannelModificationService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of `com.hcl.unica.system.model.response.collab.channel.CollaborationChannelDetails` type. `CollaborationChannelDetails` must carry the identifier & name of the updated channel. It can optionally contain URL to the channel page.

Plugin must extend the implementation of `collaboration.update-channel` service from `RestChannelModificationService` class to be recognized as a valid `collaboration.update-channel` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

- **Channel details service (`collaboration.get-channel-details`)**

  The following are the specialized interfaces and classes available for the collaboration.get-channel-details service:

    ◦ `com.hcl.unica.system.integration.service.collab.channel.details.RestChannelDetailsService` –

      The `com.example.service.rest.collaboration.ExampleChannelDetailsService` in `asset-integration-starter` project is a quick starter implementation for RESTful `collaboration.get-channel-details` service. `ExampleChannelDetailsService` class extends from `RestChannelDetailsService` class.

      The `RestChannelDetailsService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case it is JsonNode class defined inside the `ExampleChannelDetailsService` class.

      `RestChannelDetailsService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.collab.channel.CollaborationChannelDetailsRequest` class as the type argument RQ for `RestService`. Thus, the object of `CollaborationChannelDetailsRequest` becomes input to all the `collaboration.get-channel-details` services (same input is used for Functional counterpart as well). `CollaborationChannelDetailsRequest` carries the identifier of the channel to be retrieved.

      In addition to defining the input type for the `collaboration.get-channel-details` service, `RestChannelDetailsService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of `java.util.Optional` of `com.hcl.unica.system.model.response.collab.channel.CollaborationChannelDetails` type. `CollaborationChannelDetails` must carry the identifier & name of the requested channel. It can optionally contain URL to the channel page. If requested channel is not found, this method can return `Optional.empty()`.

      Plugin must extend the implementation of `collaboration.get-channel-details` service from `RestChannelDetailsService` class to be recognized as a valid `collaboration.get-channel-details` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

- **Member addition service (`collaboration.add-members-to-channel`)**

  The following are the specialized interfaces and classes available for the `collaboration.add-members-to-channel` service:

    ◦ `com.hcl.unica.system.integration.service.collab.channel.member.RestChannelMemberAdditionService` –

      The `com.example.service.rest.collaboration.ExampleMemberAdditionService` in `asset-integration-starter` project is a quick starter implementation for RESTful `collaboration.add-members-to-channel` service. `ExampleMemberAdditionService` class extends from `RestChannelMemberAdditionService` class.

      The `RestChannelMemberAdditionService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case it is JsonNode class defined inside the `ExampleMemberAdditionService` class.

`RestChannelMemberAdditionService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.collab.channel.CollaborationChannelMembers` class as the type argument RQ for RestService. Thus, the object of `CollaborationChannelMembers` becomes input to all the `collaboration.add-members-to-channel` services (same input is used for Functional counterpart as well). `CollaborationChannelMembers` carries the channel identifier & list of `com.hcl.unica.system.model.request.collab.channel.CollaborationChannelMember` objects. Each `CollaborationChannelMember` object carries identifier of the user/member to be added to the channel corresponding to the supplied channel identifier.

In addition to defining the input type for the `collaboration.add-members-to-channel` service, `RestChannelMemberAdditionService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of `com.hcl.unica.system.model.response.collab.channel.ChannelMembersAdditionStatus` type. `ChannelMembersAdditionStatus` must carry the list of `com.hcl.unica.system.model.response.collab.channel.ChannelMemberAdditionStatus` objects containing addition status for each supplied member. The `memberId` property in `ChannelMemberAdditionStatus` object should carry the identifier of member/user the status is being reported for. The error property can be used to convey error message associated with specified member's addition operation. Plugin can choose to send localized error using an API provided by `ExecutionContext`. The ok property must be set to true if member is successfully added to the channel, or to false otherwise. Reason property should be used to convey appropriate failure reason. The entire list of member addition status should be set on the status property of `ChannelMembersAdditionStatus` object to be returned from `transformResponse` method.

Plugin must extend the implementation of `collaboration.add-members-to-channel` service from `RestChannelMemberAdditionService` class to be recognized as a valid `collaboration.add-members-to-channel` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

- **Member removal service (`collaboration.remove-members-from-channel`)**

  The following are the specialized interfaces and classes available for the `collaboration.remove-members-from-channel` service:

  ◦ `com.hcl.unica.system.integration.service.collab.channel.member.RestChannelMemberRemovalService` –

  The `com.example.service.rest.collaboration.ExampleMemberRemovalService` in `asset-integration-starter` project is a quick starter implementation for RESTful `collaboration.remove-members-from-channel` service. `ExampleMemberRemovalService` class extends from `RestChannelMemberRemovalService` class.

  The `RestChannelMemberRemovalService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case it is `JsonNode` class defined inside the `ExampleMemberRemovalService` class.

  `RestChannelMemberRemovalService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.collab.channel.CollaborationChannelMembers` class as the type argument RQ for `RestService`. Thus, the object of `CollaborationChannelMembers` becomes input to all the `collaboration.remove-members-from-channel` services (same input is used for Functional counterpart as well). `CollaborationChannelMembers` carries the channel identifier & list of

`com.hcl.unica.system.model.request.collab.channel.CollaborationChannelMember` objects. Each `CollaborationChannelMember` object carries identifier of the user/member to be removed from the channel corresponding to the supplied channel identifier.

In addition to defining the input type for the `collaboration.remove-members-from-channel` service, `RestChannelMemberRemovalService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of `com.hcl.unica.system.model.response.collab.channel.ChannelMembersRemovalStatus` type. `ChannelMembersRemovalStatus` must carry the list of `com.hcl.unica.system.model.response.collab.channel.ChannelMemberRemovalStatus` objects containing removal status for each supplied member. The memberId property in `ChannelMemberRemovalStatus` object should carry the identifier of member/user the status is being reported for. The error property can be used to convey error message associated with specified member's removal operation. Plugin can choose to send localized error using an API provided by `ExecutionContext`. The ok property must be set to true if member is successfully removed from the channel, or to false otherwise. reason property should be used to convey appropriate failure reason. The entire list of member addition status should be set on the status property of `ChannelMembersRemovalStatus` object to be returned from `transformResponse` method.

Plugin must extend the implementation of `collaboration.remove-members-from-channel` service from `RestChannelMemberRemovalService` class to be recognized as a valid `collaboration.remove-members-from-channel` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

- **Message posting service (`collaboration.post-message-to-channel`)**
  The following are the specialized interfaces and classes available for the `collaboration.post-message-to-channel` service:

  ○ `com.hcl.unica.system.integration.service.collab.channel.message.RestMessagePostingService` –

  The `com.example.service.rest.collaboration.ExampleMessagePostingService` in `asset-integration-starter` project is a quick starter implementation for RESTful `collaboration.post-message-to-channel` service. `ExampleMessagePostingService` class extends from `RestMessagePostingService` class.

  The `RestMessagePostingService` class has a type parameter RS, which represents the type of response (post deserialization) received from the remote REST API. In this case it is `JsonNode` class defined inside the `ExampleMessagePostingService` class.

  `RestMessagePostingService` class implements `RestService` interface and defines the `com.hcl.unica.system.model.request.collab.channel.GroupMessage` class as the type argument RQ for `RestService`. Thus, the object of `GroupMessage` becomes input to all the `collaboration.post-message-to-channel` services (same input is used for Functional counterpart as well). `GroupMessage` carries the identifier of target channel & the message itself.

  In addition to defining the input type for the `collaboration.post-message-to-channel` service, `RestMessagePostingService` class also overrides the `transformResponse` method and mandates the return value of this method to be an object of same `GroupMessage` type. The response `GroupMessage` object should carry the identifier of newly posted message.

Plugin must extend the implementation of `collaboration.post-message-to-channel` service from `RestMessagePostingService` class to be recognized as a valid `collaboration.post-message-to-channel` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from).

Derivatives of HttpService

Only `resource-loader` standard service is implemented as an `HttpService` as it relates to the standard HTTP GET operation. You can also use `RestService` without losing any capability.

Resource loader (`resource-loader`)

The following are the specialized interfaces and classes available for resource-loader service:

- `com.hcl.unica.system.integration.service.resourceloader.DefaultWebResourceLoaderService`

  The `com.example.service.rest.ExampleResourceLoaderService` class in `asset-integration-starter` project is a quick starter implementation for the `resource-loader` service and extends from the following class:

  ```
  com.hcl.unica.system.integration.service.resourceloader
  .DefaultWebResourceLoaderService
  ```

  `DefaultWebResourceLoaderService` class is the default implementation of `resource-loader` service provided by the Content Integration SDK. If the plugin does not implement its own `resource-loader` service, Content Integration Framework falls back on this default implementation. Default implementation of `resource-loader` provided by Content Integration SDK simply follows the given resource URL and retrieves the web resource from target system. It encapsulates the standard HTTP GET operation.

  If the plugin needs to have its own `resource-loader` implementation which slightly modifies the standard HTTP GET, we recommend extending it from the `DefaultWebResourceLoaderService` class. It is not necessary to extend `resource-loader` implementation from the `DefaultWebResourceLoaderService` if the plugin must use a completely different approach for loading contents, such as reading from file system, database, FTP server etc. In such a case, it must extend from either `HttpWebResourceLoaderService` for HTTP-based approach or `WebResourceLoaderService` for functional approach.

- `com.hcl.unica.system.integration.service.resourceloader.HttpWebResourceLoaderService`

  The `DefaultWebResourceLoaderService` class discussed earlier extends from the `HttpWebResourceLoaderService` abstract class. This class defines the input type and the type of HTTP response received from target URL for `resource-loader` service as `com.hcl.unica.system.model.request.resourceloader.ResourceRequest` and `byte[]` respectively. `ResourceRequest` class encapsulates the resource URL and system identifier. Similarly, `resource-loader` works with a byte array when the content from remote HTTP URL is successfully read.

  If the plugin does not extend its `resource-loader` implementation from the `DefaultWebResourceLoaderService` class, it must at least extend from `com.hcl.unica.system.integration.service.resourceloader.HttpWebResourceLoaderService` class to be recognized as a `resource-loader` service by the Content Integration Framework (Functional counterpart, discussed later, is also a valid choice to extend from for the `resource-loader` services implemented using the Functional approach).

- `com.hcl.unica.system.integration.service.resourceloader.AbstractWebResourceLoaderService`

The `HttpWebResourceLoaderService` class discussed in previous point extends from `AbstractWebResourceLoaderService` abstract class. This class defines the following service gateway interface for the `resource-loader` service:

```
com.hcl.unica.system.integration.service.gateway
 .ResourceLoaderServiceGateway
```

To know the role of service gateways in service invocation, see . `ResourceLoaderServiceGateway` interface defines `ResourceRequest` and `HttpResponse<?>` as input and output types for the `resource-loader` service. `HttpResponse` is an interface, implemented by the `WebResource` class. It encapsulates the HTTP response headers, body, or payload, and cookies received from the remote URL. Even if the customized `resource-loader` service does not fetch the content over web, it must return the object of the `WebResource` (or any other implementation of `HttpResponse`) populated with the appropriate details. Failing to populate the `WebResource` appropriately may lead to content loading issues for client applications. The `WebResource` provides a builder API to create an object with necessary details. The most important thing is to populate the `Content-Type` header so that client application can deal with the payload accordingly. Similarly, `Content-Disposition` header must also be populated appropriately containing the filename associated with the content.

Derivatives of FunctionalService

Derivatives of FunctionalService interface facilitates creation of functional implementation of standard services. Functional service is just an object with a public method which takes a certain input and generates the desired output.

Simple search (`simple-search`)

The following are the specialized interfaces and classes available for simple-search service:

- `com.hcl.unica.system.integration.service.search.SearchService`

  The `com.example.service.functional.ExampleSimpleSearchService` class in the `asset-integration-starter` project is a quick starter implementation for the Functional `simple-search service`. It extends from the `com.hcl.unica.system.integration.service.search.SearchService` class.

  The `SearchService` class implements the `FunctionalService` interface and defines the `SearchRequest` class and the `ContentPage` class to be the type arguments RQ & RS respectively for the FunctionalService. Thus, the object of the `SearchRequest` becomes an input to all the `simple-search` services and the `ContentPage` is expected as an output on completion of the service.

  The plugin must extend its `simple-search` implementation from the `com.hcl.unica.system.integration.service.search.SearchService` class to be recognized as a `simple-search` service by the Content Integration Framework (RESTful counterpart discussed in earlier section is also a valid choice to extend from for the `simple-search` services implemented using RESTful approach).

  The `SearchService` extends from the `com.hcl.unica.system.integration.service.search.AbstractSearchService` abstract class. It introduces one more method, named `getSupportedContentTypes`. For more information on the method, see .

Resource loader (`resource-loader`)

The following are the specialized interfaces and classes available for the resource-loader service:

- `com.hcl.unica.system.integration.service.resourceloader.WebResourceLoaderService`

  The `com.example.service.functional.ExampleResourceLoaderService` class in `asset-integration-starter` project is a quick starter implementation for Functional `resource-loader` service. It extends from the following class:

  ```
  com.hcl.unica.system.integration.service.resourceloader.WebResourceLoaderService
  ```

  The `WebResourceLoaderService` class implements the `FunctionalService` interface and defines the `ResourceRequest` and the `HttpResponse` types to be the type arguments RQ & RS, respectively, for the `FunctionalService`. Thus, the object of the `ResourceRequest` becomes an input to all the `resource-loader` services and the `HttpResponse` is expected as an output on completion of the service (the same input and output types are used for RESTful counterpart of the resource-loader). For more information on `ResourceRequest` & `HttpResponse` types, see Derivatives of RestService on page 31.

  The plugin must extend its `resource-loader` implementation from the `com.hcl.unica.system.integration.service.resourceloader.WebResourceLoaderService` service to be recognized as a `resource-loader` service by the Content Integration Framework (HTTP counterpart discussed in the earlier section is also a valid choice to extend from for the `resource-loader` services implemented using the HTTP approach).

  The WebResourceLoaderService extends from the following class:

  ```
  com.hcl.unica.system.integration.service.resourceloader.
  AbstractWebResourceLoaderService
  ```

  For more information about this class, see Derivatives of RestService on page 31.

List content categories (`list-content-categories`)

The following are the specialized interfaces and classes available for list-content-categories service:

- `com.hcl.unica.system.integration.service.content.categories.list.ContentCategoriesListService`

  Plugin can alternatively choose Functional approach to implement `list-content-categories` service by extending the implementation from `ContentCategoriesListService` class. The `ContentCategoriesListService` class implements the `FunctionalService` interface and mandates the `ContentCategoryListRequest` and the `List<ContentCategory>` classes to be the type arguments **RQ** and **RS** respectively for the `FunctionalService`. Thus, the object of the `ContentCategoryListRequest` becomes an input to the `list-content-categories` service and the object of List<ContentCategory> type is expected as an output on completion of the service. `ContentCategoryListRequest` should be consulted for parent category ID & folder ID to fetch categories belonging to specific parent category and/ or to parent folder. As of current implementation of Content Integration Framework, parent (category) ID is always null since only one level of category list is supported. However, if categories are organized in folders & `list-content-categories` service considers folder ID while listing categories, then Content Integration Framework can deal with navigable folder list if `list-category-folders` service is also implemented.

- The plugin must extend its list-content-categories implementation from the `com.hcl.unica.system.integration.service.content.categories.list.ContentCategoriesListService` class to be

recognized as a valid `list-content-categories` service by the Content Integration Framework (RESTful counterpart discussed in earlier section is also a valid choice to extend from).

`ContentCategoriesListService` extends from `AbstractContentCategoriesListService` class. Details of `AbstractContentCategoriesListService` class are covered in the Derivatives of RestService on page 31 topic.

List folders (`list-folders`)

The following are the specialized interfaces and classes available for list-folders service:

- `com.hcl.unica.system.integration.service.folder.list.FolderListService`

  Plugin can alternatively choose Functional approach to implement `list-folders` service by extending the implementation from `FolderListService` class. The `FolderListService` class implements the `FunctionalService` interface and mandates the `FolderListRequest` and the `List<Folder>` classes to be the type arguments **RQ** and **RS** respectively for the `FunctionalService`. Thus, the object of the `FolderListRequest` becomes an input to the `list-folders` service and the object of `List<Folder>` type is expected as an output on completion of the service.

- The plugin must extend its list-folders implementation from the `com.hcl.unica.system.integration.service.folder.list.FolderListService` class to be recognized as a valid `list-folders` service by the Content Integration Framework (RESTful counterpart discussed in earlier section is also a valid choice to extend from).

  `FolderListService` extends from `AbstractFolderListService` class. Details of `AbstractFolderListService` class are covered in the Derivatives of RestService on page 31 topic.

List category folders (`list-category-folders`)

The specialized types used for implementing `list-category-folders` are exactly same as the ones used for list-folders.

List contents (`list-contents`)

The following are the specialized interfaces and classes available for list-contents service:

- `com.hcl.unica.system.integration.service.content.list.ContentListService`

  Plugin can alternatively choose Functional approach to implement `list-contents` service by extending the implementation from `ContentListService` class. The `ContentListService` class implements the `FunctionalService` interface and mandates the `ContentListRequest` and the `ContentPage` classes to be the type arguments **RQ** and **RS** respectively for the `FunctionalService`. Thus, the object of the `ContentListRequest` becomes an input to the `list-contents` service and the object of `ContentPage` type is expected as an output on completion of the service.

- The plugin must extend its list-contents implementation from the `com.hcl.unica.system.integration.service.content.list.ContentListService` class to be recognized as a valid list-contents service by the Content Integration Framework (RESTful counterpart discussed in earlier section is also a valid choice to extend from).

  `ContentListService` extends from `AbstractContentListService` class. Details of `AbstractContentListService` class are covered in the Derivatives of RestService on page 31 topic.

Get content details (`get-content-details`)

The following are the specialized interfaces and classes available for get-content-details service:

- `com.hcl.unica.system.integration.service.content.details.ContentDetailsService`

  Plugin can alternatively choose Functional approach to implement `get-content-details` service by extending the implementation from `ContentDetailsService` class.

  The `ContentDetailsService` class implements the `FunctionalService` interface and mandates the `ContentDetailsRequest` and the Presentable classes to be the type arguments **RQ** and **RS** respectively for the `FunctionalService`. Thus, the object of the `ContentDetailsRequest` becomes an input to the `get-content-details` service and the object of Presentable type is expected as an output on completion of the service.

  The plugin must extend its `get-content-details` implementation from the `com.hcl.unica.system.integration.service.content.details.ContentDetailsService` class to be recognized as a valid `get-content-details` service by the Content Integration Framework (RESTful counterpart discussed in earlier section is also a valid choice to extend from).

  `ContentDetailsService` extends from `AbstractContentDetailsService` class. Details of `AbstractContentDetailsService` class are covered in the topic.

Get object schema (`get-object-schema`)

`get-object-schema` service is used to generate the schema of domain object or entity used by the respective system to represent the content. Object schema in simplest form is just a hierarchical metadata of each mappable content/object attribute. Attribute hierarchy and metadata is expected to match the JSON representation of the domain object. Attribute metadata mainly includes the data type of the attribute, format of the value held in the attribute, unique identifier of the attribute and display title or label for the attribute.

The following are the specialized interfaces and classes available for `get-object-schema` service:

- `com.hcl.unica.system.integration.service.object.schema.ObjectSchemaProviderService`

  The `ObjectSchemaProviderService` class implements the `FunctionalService` interface and mandates the `com.hcl.unica.system.model.ObjectSchemaRequest` and the `com.hcl.unica.system.model.json.schema.ObjectSchema` classes to be the type arguments **RQ** and **RS** respectively for the `FunctionalService`. Thus, the object of the `ObjectSchemaRequest` becomes an input to the `get-object-schema` service and the object of `ObjectSchema` type is expected as an output on completion of the service. Plugin however need not build the `ObjectSchema` by itself. It should just override and implement following abstract method from `ObjectSchemaProviderService` class.

  ObjectProfile getObjectProfile(ObjectSchemaRequest objectSchemaRequest)

  The `getObjectProfile()` method accepts `ObjectSchemaRequest` and returns `ObjectProfile`. (These types are discussed in subsequent section.)

  The plugin must extend `get-object-schema` implementation from the `com.hcl.unica.system.integration.service.object.schema.ObjectSchemaProviderService` class to be recognized as a valid `get-object-schema` service by the Content Integration Framework. There is no RESTful counterpart of this

standard super class since object schema generation does not include any HTTP interaction. Plugins can implement custom RESTful service and invoke it internally from within `get-object-schema` service if required.

- `com.hcl.unica.system.model.ObjectSchemaRequest`

Object of this class is supplied as an input to the `get-object-schema` service. The most important method of this class is `getObjectIdentity()` which returns an object of type `com.hcl.unica.system.model.ObjectIdentity` encapsulating the details of the content chosen by the user to request the master schema. It includes `applicationId` (the system identifier), `objectType` (content type/category identifier) and `objectId` (unique identifier of the selected content).

Content categorization can be strict or lenient. (Refer to `additionalFeatures | content | categorization` section in Service declaration file on page 5 for the overview of content categorization & the standard system property to declare the nature of categorization.) If `additionalFeatures | content | categorization` is set to strict, then object schema should be generated strictly specific to the given content type/category.

If categorization property is set to lenient, then regardless of the category and/or content chosen by the user at the time of setting up content mapping, the generated schema must include attributes of all kinds of contents supported by the respective system. In other words, only one master schema is used for mapping all types of contents provided by the given system.

Furthermore, `get-object-schema` service is also used for generating schemas for request & response objects of Object extension services (known as Custom actions in Centralized Offer Management). In such case, `{object-extension-service-name}.request` & `{object-extension-service-name}.response` are supplied in `objectType` property of `ObjectSchemaRequest` object. (`{object-extension-service-name}` denotes the actual name of service declared in `custom-plugin-services.yml` file).

The `getEnrichmentObjectJson()` method in `ObjectSchemaRequest` class can be ignored as of current release.

- `com.hcl.unica.system.integration.service.object.schema.ObjectProfile`

This is a return type of `getObjectProfile()` method in `get-object-schema` service. It carries the Java type corresponding to the domain entity/object representing the required type/category of content for which schema needs to be generated.. Content Integration Framework consults this Java type to generate the schema for public and non-public non static class properties (inclusive of Enums & Optionals). `@MappableAttribute` annotation can be used to configure each individual class property to control the schema generated by Content Integration Framework. Refer to the `com.aem.model.response.simplesearch.SimpleSearchItem` domain object in `aem-integration reference` project to get an idea about how this annotation is used. More details are provided on `@MappableAttribute` in next section. `ObjectProfile` can optionally include an instance of `com.hcl.unica.system.integration.service.object.schema.ObjectSchemaEnricher` to dynamically add/modify/remove attributes from the schema thus generated. Next section explains `ObjectSchemaEnricher` in detail.

- `com.hcl.unica.system.integration.service.object.schema.ObjectSchemaEnricher`

`ObjectSchemaEnricher` is an abstract class. Implementation of `ObjectSchemaEnricher` gives an opportunity to generate object schemas dynamically if the concerned domain object contains some dynamic properties (for example, dynamically added attributes to Offer Template. Each Offer Template can have different set of attributes which cannot be statically listed in corresponding Java class.). The `ObjectProfile` class discussed in previous section has a property (`schemaEnricher`) to hold reference to `ObjectSchemaEnricher`, if required. If dynamic enrichment is not desired, then `schemaEnricher` property of `ObjectProfile` can be left out (don't set it to null though). Plugin should

extend it to have desired implementation. The type parameter to `ObjectSchemaEnricher` class represents the Java type containing the additional details required for enriching the statically generated object schema. These additional details might be provided by the client applications of Unica Content Integration. As of current release, no additional details are provided, hence it should be set to Void while implementing the schema enricher. `ObjectSchemaEnricher` declares only one abstract method which should be implemented by the plugin:

```
abstract public ObjectSchema enrich(
  ObjectSchema objectSchema,
  ObjectSchemaEnrichmentRequest<T> objectSchemaEnrichmentRequest
)
```

The first argument to this method is an instance of `com.hcl.unica.system.model.json.schema.ObjectSchema` class. It contains the automatically generated domain object schema derived from the Java type supplied in `ObjectProfile`. At its core, `ObjectSchema` is just a `Map<String, AttributeSchema>`, wherein class property names forms the keys of this map and property metadata ends up as an object of `AttributeSchema`. If the class property in turn refers to another object, the corresponding `AttributeSchema` will have another `Map<String, AttributeSchema>` containing the attributes of that object type and so on.

> **Note:** It is important to note that attribute names used as the keys in attribute map correspond to the JSON properties which ends up in the JSON representation of the domain object. Hence, if `@JsonProperty` annotation is used to override the JSON property name for certain class attribute, then Content Integration Framework automatically detects it and uses the overridden property name while generating schema for the supplied Java type. Likewise, `ObjectSchemaEnricher` implementation must also ensure that names of dynamically added attributes must match JSON representation of concerned domain object.

`ObjectSchema` as well as `AttributeSchema` extend from `com.hcl.unica.system.model.json.schema.AttributeContainer` abstract class. `AttributeContainer` provides convenience methods to `ObjectSchema` and `AttributeSchema` classes for navigating through attribute hierarchy as well as for adding, modifying and removing attributes at any level in the hierarchy to ease the schema enrichment. Attributes at any level in the hierarchy can be accessed and manipulated using their names as appearing in JSON representation.

- `com.hcl.unica.system.model.json.schema.generator.annotations.MappableAttribute`

`@MappableAttribute` annotation provides a way to control how Content Integration Framework generates object schema from the respective Java type. Use of `@MappableAttribute` is not mandatory. If it is not used, Content Integration Framework automatically figures out property metadata. If required, this annotation should be placed on top of desired class properties. Following annotation attributes can be used to control the schema generation:

- **hidden** – Set this to true to explicitly exclude certain property from object schema (`@JsonIgnore` is presently not considered by Content Integration Framework. Hence, any property excluded from JSON representation using `@JsonIgnore` must be explicitly excluded from schema)
- **id** – Supply unique identifier for the property. Content Integration Framework needs unique identifier for each mappable class property. If `@MappableAttribute` is not used, or **id** is not specified, it generates one automatically based on the location of property inside the class.

Automatic generation of attribute identifier is subject to the name and the hierarchical location of class property inside the domain object graph. It implies that if the property name is changed and/or moved up

or down the object graph hierarchy, it will change the identifier associated with it. Such refactoring can mislead Content Integration Framework while reading the values of refactored attributes and may lead to undesired data in mapped contents (such as Offers in COM). Hence, to avoid such inadvertent changes in attribute identifiers, we recommend you to assign unique attribute identifiers manually, which remain constant regardless of the name and location of class properties.

- ◦ **title** – Display title/label for the property. If omitted, Content Integration Framework generates one using property name.
- ◦ **type** – One of the values from

  `com.hcl.unica.system.model.json.schema.generator.annotations.AttributeType`. If omitted, Content Integration Framework automatically figures out the appropriate type. For numeric properties holding dates as Unix timestamp (seconds since epoch), type must be set to `AttributeType.INTEGER`, and format must be set to `AttributeFormat.DATETIME`.

- ◦ **format** – One of the values from

  `com.hcl.unica.system.model.json.schema.generator.annotations.AttributeFormat`. Content Integration Framework can automatically identify standard java temporal types (`Date`, `LocalDateTime`, `Instant`) and set the attribute type to `DATETIME`. Other formats, such as URL, HTML, EMAIL should be explicitly declared.

- ◦ **implementation** – Should be used for polymorphic references to explicitly declare the Java type to be considered for automatic schema generation.

- ◦ **hiddenProperties** - `@MappableAttribute` annotation can be used at the class level to hide multiple properties at single place. `hiddenProperties` takes an array of Strings containing the names of properties (direct as well as inherited ones) to be excluded from automatically generated schema. It is particularly useful for hiding properties inherited from third party parent class.

- ◦ **pattern** – Content Integration Framework supports certain temporal types for representing date & timestamps, viz. `Date, LocalDateTime, Instant`. If certain date properties are maintained in String format, then pattern can be used to declare the pattern of date `strings.type` for such attributes can be String, the format however must be set to `AttributeFormat.DATETIME.pattern` must contain valid pattern string supported by Java's DateTimeFormatter.

- ◦ **maxItems** – Content Integration Framework supports schema generation & related use cases for arrays of scalers & arrays of objects from release 12.1.2. If certain attribute is an array or collection of scalers (primitives & their wrapper types, Strings) or other custom Java type, then framework automatically identifies it and sets the type of such property to `AttributeType.ARRAY`. By default, framework considers maximum 1 item in the given array. `maxItems` can be used to explicitly override maximum number of items such attribute can carry. `maxItems` expects and array of integers. For now, only one -dimensional arrays are supported by Content Integration Framework. Hence, the maximum items must be specified as first element of array supplied against `maxItems`. For example, maxItems = {5}. This example denotes that specified property can carry at most 5 items. It may not always carry those many items, but if it ever does, plugin doesn't expect more than specified number items in it. The `maxItems` value helps to limit the number of individual array element shown on UI at the time of Content/Object mappings.

- ◦ **serialization** – By default, properties declared as Collection<String> (List/Set) are serialized or resolved as comma separated values (CSV). This default behavior is intact even after the support for arrays in release 12.1.2. However, one can switch off this default consideration for List<String> by setting serialization to

`AttributeSerialization.AUTO`. Thus, `serialization` property lets the developer control whether List<String> should be considered as CSV or as an array of strings. Future releases may use `serialization` for controlling other aspects of attributes.

External management of attribute mappability

In addition to controlling `mappability` of attributes by placing `@MappableAttribute` annotation directly on top of class properties, Content Integration Framework also allows to control each attribute externally using `@MappableObjects` & `@MappableObject` annotations placed on top of your domain type/class. If you don't have access to the source of your domain class or it is an auto generated class, you can create a subclass extending from it, and place `@MappableObjects` annotation on top of your subclass & control each individual property inherited from parent class. Refer to `com.example.model.mapping.SubType` & `com.example.model.mapping.SuperType` classes from `asset-integration-starter` project to learn more about these annotations. If your super class uses some other nested types/classes, properties of those types can also be controlled using this approach. `@MappableObjects` is an array of `@MappableObject`, which allows to specify the Java type & associated array of properties (`@Property`). The `@Property` annotation in turn carries name of the property & `@MappabileAttribute` declaration for that property. All capabilities of `@MappabileAttribute` are available using this approach as well.

- **Java Type to AttributeType Mapping**

Following table summarizes the mapping between Java type and AttributeType/AttributeFormat used by the Content Integration Framework for automatic schema generation:

Java Type	AttributeType	AttributeFormat
◦ `String` ◦ `Character` ◦ `Char` ◦ `CharSequence` ◦ `LocalDate` ◦ `LocalTime` ◦ `ZonedDateTime` ◦ `OffsetDateTime` ◦ `OffsetTime` ◦ `ZoneId` ◦ `Calendar` ◦ `UUID`	`STRING`	
◦ `Boolean` ◦ `boolean`	`BOOLEAN`	
◦ `BigInteger` ◦ `Integer` ◦ `Int` ◦ `Long` ◦ `Long`	`INTEGER`	

Java Type	AttributeType	AttributeFormat
◦ `Short` ◦ `Short` ◦ `Byte` ◦ `byte`		
◦ `BigDecimal` ◦ `Number` ◦ `Double` ◦ `Double` ◦ `Float` ◦ `float`	`NUMBER`	
◦ `Date` ◦ `LocalDateTime` ◦ `Instant` Content Integration Framework expects date values be expressed in UTC standard time. Temporal values expressed in any other timezone can lead to inaccurate temporal calculations in further use cases.	`INTEGER`	`DATETIME`

Collection<String> is resolved as comma separated list of strings. If any string element in the collection contains comma, it is enclosed in double quotes. If the enclosed string contains any double quote of its own, then its double quote is escaped by another double quote.

For example, a list of strings like ["string 1","string 2","string, 3","string,\"4","string\"5"] is resolved as follows –

string 1,string 2,"string, 3","string,""4",string"5

Notice that string"5 is not enclosed in double quotes since it does not contain any comma.

- `com.hcl.unica.system.model.Mappable`

Contents from content repositories are linked to Unica objects using content URLs. Once any content URL is referenced inside Unica objects, Content Integration Framework allows showing preview of already linked content by making use of `get-content-details` service. To achieve such content previews at later point of time, Content Integration Framework & Unica products store the system identifier, category/type & identifier of the linked content. By default, any attribute declared as AttributeType.STRING & AttributeFormat.URL using @MappableAttribute annotation will be considered as an URL belonging to the main content. If such URL attribute value is referenced in Unica objects, then the preview for such reference will show the details of content it belongs to. However, the URL attribute may not necessarily represent the content object it is wrapped in. It may be a separate entity managed inside target content repository. For example, a Book object may include two different image URL attributes, called

bookCover & authorPhoto. While preview for bookCover should certainly show the details of the book it represents, authorPhoto however should show the brief bio about book's author.

With the default arrangement, Content Integration Framework continues to show the book details for authorPhoto's preview unless explicitly overridden by the plugin. The Mappable interface provides an opportunity to override this default behavior for desired attribute. To override the content preview for any attribute, domain class must implement the Mappable interface and override following method –

```
ObjectIdentity overrideObjectReference(String attributeId, ObjectIdentity defaultReference)
```

The `overrideObjectReference` method accepts the attribute identifier whose real content identity needs to be established. It also receives the default identity assumed by Content Integration Framework as second argument. Plugin can conditionally override the content identity by matching the `attributeId` with desired attribute's identifier. The `ObjectIdentity` returned from this method essentially tells the real system (applicationId), category (objectType) & identifier (objectId) of the respective attribute. Returning null from this method turns the content preview off. In other circumstances, plugin should respond with `defaultReference` value to let the Content Integration Framework function normally.

Get cognitive analysis (`get-cognitive-analysis`)

The following are the specialized interfaces and classes available for get-cognitive-analysis service:

- `com.hcl.unica.system.integration.service.cognitive.analysis.CognitiveAnalysisService`

  Plugin can alternatively choose Functional approach to implement `get-cognitive-analysis` service by extending the implementation from `CognitiveAnalysisService` class. The `CognitiveAnalysisService` class implements the `FunctionalService` interface and mandates the `CognitiveAnalysisRequest` and the `CognitiveAnalysis` classes to be the type arguments **RQ** and **RS** respectively for the `FunctionalService`. Thus, the object of the `CognitiveAnalysisRequest` becomes an input to the `get-cognitive-analysis` service and the object of `CognitiveAnalysis` type is expected as an output on completion of the service.
- The plugin must extend its `get-cognitive-analysis` implementation from the `com.hcl.unica.system.integration.service.cognitive.analysis.CognitiveAnalysisService` class to be recognized as a valid `get-cognitive-analysis` service by the Content Integration Framework (RESTful counterpart discussed in earlier section is also a valid choice to extend from).

  CognitiveAnalysisService extends from `AbstractCognitiveAnalysisService` class. Details of `AbstractCognitiveAnalysisService` class are covered in the topic.

Object extension service

The following are the specialized interfaces and classes available for object extension service:

- `com.hcl.unica.system.integration.service.object.extension.ObjectExtensionService`

  Plugin can alternatively choose Functional approach to implement object extension service by extending the implementation from ObjectExtensionService class. The ObjectExtensionService class implements the FunctionalService interface and mandates the ObjectExtensionRequest and the Object classes to be the type

arguments RQ and RS respectively for the FunctionalService. Thus, the object of the ObjectExtensionRequest becomes an input to the object extension service and expects any type of as an output on completion of the service.

`ObjectExtensionRequest` is covered in more detail under RESTful section of object extension service.

- The plugin must extend its object extension service implementation from the com.hcl.unica.system.integration.service.object.extension.ObjectExtensionService class to be recognized as a valid object extension service by the Content Integration Framework (RESTful counterpart discussed in earlier section is also a valid choice to extend from).

`ObjectExtensionService` extends from `AbstractObjectExtensionService` class. Details of `AbstractObjectExtensionService` class are covered in the Derivatives of `RestService` topic.

Collaboration services

Please refer the example services from `com.example.service.functional.collaboration` package in `asset-integration-starter` project. Like any other Functional counterpart of RESTful service implementation, the request & response contract for Functional collaboration services remain same as their RESTful counterparts. Please refer Derivatives of RestService on page 31 section for complete list of collaboration services.

Derivatives of InboundHttpService

Derivatives of InboundHttpService facilitates exposing RESTful endpoints coupled with certain predefined features in Content Integration Framework. As of release 12.1.2, Content Integration Framework allows exposing RESTful endpoints (webhooks) to receive & process content lifecycle events.

Object event interpreter service (Webhook)

The following are the specialized interfaces and classes available for the object event interpreter service:

- `com.hcl.unica.system.integration.service.object.event.ObjectEventInterpreterService`

   The `com.example.service.rest.ExampleEventInterpreterService` class in asset-integration-starter project is a quick starter implementation for an object event interpreter service. It extends from `ObjectEventInterpreterService` class. Implementing an object event interpreter service by extending standard ObjectEventInterpreterService exposes a webhook which can be invoked by the respective content repository to send content lifecycle event notifications.

   The `ObjectEventInterpreterService` class has a type parameter T, which represents the type of request (post deserialization) received from the client of webhook. In case of `ExampleEventInterpreterService`, request body is expected in String format. The type parameter T can be any valid Jackson or JAXB annotated class if JSON or XML is expected in the request body. Deserialization of JSON & JAXB happens based on the `Content-Type` request header. Having said that, Content Integration Framework supports only application/json & text/xml or application/xml type of request body if automatic deserialization is desired.

   Object event interpreter service does not directly respond to the client. Instead, it helps to interpret the incoming event information and shares the necessary details with Content Integration Framework for further processing. Content Integration Framework responds to the client accordingly. It responds with 200 HTTP response if interpretation succeeds. If event interpreter service cannot understand the event and fails to figure out necessary details about the event, Content Integration Framework responds with appropriate 4xx HTTP error response.

`ObjectEventInterpreterService` class mandates the event interpreter service to implement following methods:

- `ObjectEventListenerRequestSpec getRequestSpec(ServiceConfig serviceConfig)`

  This method accepts an object of `ServiceConfig` class and returns an object of type `ObjectEventListenerRequestSpec`. The `ObjectEventListenerRequestSpec` is essentially the partial specification of webhook's signature. The `ObjectEventListenerRequestSpec` wraps an object of `InboundHttpRequestSpec`, which comprises the relative endpoint URL of the webhook, and the HTTP methods supported by the webhook being exposed. The relative URL can contain path variables enclosed in curly braces. Refer to the `getRequestSpec` implementation in `ExampleEventInterpreterService`. Runtime values for these path variables can be obtained from `InboundHttpRequestContext`. See InoundHttpRequestContext<T> on page 26 section for more details.

  Avoiding webhook conflicts

  If plugin implements more than one event interpreter services, then it is important to note that no two interpreter services should expose webhooks with conflicting signatures. For example, if one interpreter service exposes a webhook with relative URL as /wh/{contentId}, supporting POST method, then another interpreter service must use different relative URL and/or different HTTP method to form a different signature. Thus, another webhook for supporting PUT method for the same relative URL /wh/{contentId} would be completely valid. (Note that single interpreter service can support multiple HTTP methods for same relative URL.)

- `Optional<ObjectEventDetails> interpret(InboundHttpRequestContext<T> executionContext)`

  This method accepts an object of InboundHttpRequestContext<T>, wherein T represents the type of request body. As explained in InoundHttpRequestContext<T> on page 26 section section, `InboundHttpRequestContext` can be used to obtain contextual information, including request payload, path variables, request parameters & request headers. In response, it is expected to return an object of `ObjectEventDetails` comprising the details of the event received via exposed webhook. Return value is wrapped in Optional since interpreter may not always be able to interpret the incoming event because of incorrect invocations. In such case, empty Optional should be returned. On successful interpretation, this method should return an Optional containing an instance of `ObjectEventDetails` comprising following details –

- `IdentifiableObject object`- The domain object for which event has been received. The class corresponding to domain object must implement IdentifiableObject interface and provide object ID & type accordingly.. In case of deletion events, fully populated object is not desired. Hence, an object of `com.hcl.unica.system.model.DeletedEntity` containing the identifier & type of deleted object can be returned. Refer to the implementation of `interpret` method in `ExampleEventInterpreterService`.

- `ObjectEventType eventType` – The type of event.

- `ObjectState transitionedState` – Current relevant state of the object. (As of release 12.1.2, transitionedState is ignored. It may be used in subsequent releases)

Plugin must extend its event interpreter service from the `com.hcl.unica.system.integration.service.object.event.ObjectEventInterpreterService` class for the successful exposure of webhook by the Content Integration Framework. On successful creation of event interpreter service,

Content Integration Framework exposes an endpoint URL in following format relative to the application context root. The HTTP methods supported by this URL are guided by the `getRequestSpec` method -

`/api/AssetPicker/webhook/{SYSTEM_ID}/events/{eventName}/{contentId}`

Wherein,

- **{SYSTEM_ID}** represents the identifier of the respective system.
- **/{eventName}/{contentId}** represents the relative URL configured in getRequestSpec method. **eventName** & **contentId** forms the path variables in this example.

Webhook security

All the webhooks exposed in Content Integration Framework are protected by default. See API Security Filter (link to See API Security Filter configuration in Unica Platform Admin Guide to learn more about API security. If none of the authentication mechanisms suites the need of Plugin, then event interpreter service can override `validate` method to perform custom authentication & authorization. Refer the implementation of `validate` method in `ExampleEventInterpreterService` class to get a brief idea about custom authentication. It demonstrates the use of Platform's user data source for maintaining API credentials. The USERNAME & PASSWORD constants represent the authentication information received in webhook request, either by means of request headers or embedded in the request body itself.

Derivatives of KafkaConsumerService

Derivatives of KafkaConsumerService facilitates receiving messages from Kafka topics. As of release 12.1.2, Content Integration Framework leverages Kafka consumer service to receive domain object events & subsequently process those events based on the object attribute mappings.

Object event interpreter service (Kafka listener)

Just like Webhook support can facilitate event receipt over HTTP, Kafka listeners can help to receive events via topics. It is important to note that, regardless of event channel (HTTP/Kafka), the subsequent event processing is always done based on the object attribute mapping. The following are specialized classes & interfaces available for implementation of Kafka based event receivers.

- `com.hcl.unica.system.integration.service.object.event.AbstractKafkaEventInterpreterService<K, V extends IdentifiableObject>`

  The K & V type parameters to `AbstractKafkaEventInterpreterService` represent the type of Key & Value components of the message received from Kafka topic. As of release 12.1.2, only String type of keys are supported, whereas value type must be subtype of `IdentifiableObject`. `IdentifiableObject`'s contract helps Content Integration Framework identify the type/category & identifier of the domain object corresponding to the event. If null is returned from `getObjectType()` implementation, then Content Integration Framework uses `{service-name}`.request as the category of corresponding domain object. Wherein, {service-name} is the actual name of the service declared in custom-plugin-services.yml file for respective event receiver/interpreter service. There is no standard service name for Kafka based event receiver/interpreter services. The service declaration should also include the `eventProfile` standard parameter –

```
{service-name}
  params:
```

```
        eventProfile:
        label: Credit Card Transaction
```

The label parameter works as the title for respective domain object whenever it is shown on the object mapping screen.

Refer the `com.example.service.kafka.events.consumer.ExampleKafkaEventInterpreterService` class from `asset-integration-starter` project for additional clarity.

`AbstractKafkaEventInterpreterService` declares following methods which service implementation must define –

1. `Optional<ObjectEventDetails> interpret(ExecutionContext<KafkaMessage<String, DomainEntity>> executionContext)` This method should interpret the event received via Kafka and possibly provide event details by means of `ObjectEventDetails`. The semantics of this method are exactly same as the semantics of `interpret` method required for Webhooks.

2. `boolean accept(ExecutionContext<KafkaMessage<String, DomainEntity>> executionContext)` This is an optional method. If implemented, this method can determine the acceptance or rejection of the received event. By default, Content Integration Framework processes all the events successfully interpreted by `interpret` method. `accept` method is called prior to the interpretation. If certain event is not suitable for further consideration, this method can return false, thereby skipping its interpretation as well as processing.

- `com.hcl.unica.system.integration.service.kafka.KafkaMessage<K, V>`

  KafkaMessage is supplied as request object to previously mentioned methods in `AbstractKafkaEventInterpreterService`. It encapsulates the key & value corresponding to the message received via Kafka topic.

- `com.hcl.unica.system.integration.service.object.event.GenericKafkaEventInterpreterService` Kafka based object event receipt & interpretation can also be done without creating any explicit service using `AbstractKafkaEventInterpreterService`. Content Integration Framework provides `GenericKafkaEventInterpreterService` for receiving and interpreting object events based on certain default assumptions. The service declaration in `custom-plugin-services.yml` file can be pointed to `GenericKafkaEventInterpreterService` and fully qualified Java type of associated event object can be declared as shown below

```
systemId: Foo
  serviceName: transaction-event-receiver
  factoryClass: c.h.u.s.i.s.object.event.GenericKafkaEventInterpreterService
  params:
    eventProfile:
      class: com.example.service.kafka.events.consumer.DomainEntity
      label: Credit Card Transaction
```

Wherein, class under the eventProfile service parameter should contain fully qualified name of the class representing domain object and/or event received via Kafka. The classes specified herein must implement IdentifiableObject. Presently there is no way to determine acceptance or rejection of events using this approach though.

Platform configuration

Each Kafka consumer service needs certain additional properties to be set in Platform configuration of corresponding system. Given below is the list of such additional properties. These properties must be listed in "Additional properties"

section for respective system configuration. Note that Kafka connectivity details are obtained from Kafka configuration settings made for the respective system –

Parameter	Description
{service-name}.kafka.topics	Comma separated list of source topic names
{service-name}.kafka.topics.{topic-name}.value.format	Expected message format. Supported formats are Avro & Json.
{service-name}.kafka.max-consumers	Number of required concurrent consumers for the given service (inclusive of all the topics service would listen to)
{service-name}.kafka.fetch.max.wait.ms	Same as standard fetch.max.wait.ms property used for Apache Kafka consumers
{service-name}.kafka.fetch.min.bytes	Same as standard fetch.min.bytes property used for Apache Kafka consumers
{service-name}.kafka.fetch.max.bytes	Same as standard fetch.max.bytes property used for Apache Kafka consumers
{service-name}.kafka.max.partition.fetch.bytes	Same as standard max.partition.fetch.bytes property used for Apache Kafka consumers
{service-name}.kafka.max.poll.records	Same as standard max.poll.records property used for Apache Kafka consumers
{service-name}.kafka.max.poll.interval.ms	Same as standard max.poll.interval.ms property used for Apache Kafka consumers
{service-name}.kafka.heartbeat.interval.ms	Same as standard heartbeat.interval.ms property used for Apache Kafka consumers
{service-name}.kafka.session.timeout.ms	Same as standard session.timeout.ms property used for Apache Kafka consumers

Note: Refer https://kafka.apache.org/documentation/#consumerconfigs for standard Kafka consumer configurations.

{service-name} represents the name of service declared in custom-plugin-services.yml file.

For example,

```
transaction-event-receiver.kafka.topics: topic1, topic2

transaction-event-receiver.kafka.topics.topic1.value.format: Avro

transaction-event-receiver.kafka.topics.topic2.value.format: Json

transaction-event-receiver.kafka.max-consumers: 20
```

```
transaction-event-receiver.kafka.fetch.max.wait.ms: 500

transaction-event-receiver.kafka.fetch.min.bytes: 26124

transaction-event-receiver.kafka.fetch.max.bytes: 31240

transaction-event-receiver.kafka.max.partition.fetch.bytes: 51124

transaction-event-receiver.kafka.max.poll.records: 200

transaction-event-receiver.kafka.heartbeat.interval.ms: 2000

transaction-event-receiver.kafka.session.timeout.ms: 9000
```

SASL prerequisites

For SASL connectivity with Kafka, provide appropriate authorization to consumer groups. Content Integration Framework creates consumer group for each configuration of respective system & assigns identifier to it as per below format -

`ci-consumer-{Unica-product-platform-config-node}-{partition-name}-{system-id}-{service-name}`

For example, for the `event-receiver` service in system `Foo`, following consumer groups will be created if `Foo` is configured in `partition1` of Unica Centralized Offer Management (`Offer`) as well as in Unica Journey (`Journey`).

`ci-consumer-offer-partition1-foo-event-receiver` (for system configured under Affinium|`Offer`|partitions|`partition1`|assetPicker|dataSources|`Foo` node in Platform configuration)

`ci-consumer-journey-foo-event-receiver` (for system configured under Affinium|`Journey`|assetPicker|dataSources|`Foo` node in Platform configuration)

> **Note:** Spaces in system identifier are replaced with dash(-) for composing Kafka consumer group ID.

Derivatives of KafkaProducerService

`KafkaProducerService` lets publishing messages/records onto Kafka topics. Following are the specialized types available for creating Kafka publisher service –

- `com.hcl.unica.system.integration.service.kafka.AbstractKafkaProducerService<K, V>`The type parameter K & V represents the key & value of the message to be sent onto target Kafka topic. Refer the `com.example.service.kafka.events.producer.ExampleKafkaProducerService from asset-integration-starter` project for a typical service definition. In addition to declaring type arguments, this service must implement the `getServiceInterface()` method to be able to invoke Kafka producer service programmatically. Apart from these declarations, `AbstractKafkaProducerService` requires no other business logic.

Platform configuration

Once Kafka producer service is defined & declared in custom-plugin-services.yml, certain additional parameters must be set up to be able to publish to any Kafka topic. Note that Kafka connectivity details are obtained from Kafka configuration settings made for the respective system. Following additional parameters must be configured in "Additional parameters" section of respective system configuration –

Parameter	Description
{service-name}.kafka.topics	Comma separated list of target topic names.
{service-name}.kafka.topics.{topic-name}.value.format	Expected message format. Supported formats are Avro & Json.
{service-name}.kafka.batch.size	Same as standard batch.size property used for Apache Kafka producers.
{service-name}. kafka.linger.ms	Same as standard linger.ms property used for Apache Kafka producers.

📝 **Note:** Refer https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html for standard Kafka producer configurations.

{service-name} represents the name of service declared in custom-plugin-services.yml file.

For example,

```
message-publisher-service.kafka.topics: topic1, topic2
```

```
message-publisher-service.kafka.topics.topic1.value.format: Json
```

```
message-publisher-service.kafka.topics.topic2.value.format: Avro
```

```
message-publisher-service.kafka.batch.size: 16900
```

```
message-publisher-service.kafka.linger.ms: 10
```

Invocation of KafkaProducerService

Content Integration Framework does not directly invoke any Kafka producer service. Plugin can define & use such service if required. Refer `com.example.service.client.ExampleKafkaProducerClient` from `asset-integration-starter` project to get clarity on how already defined Kafka producer service can be invoked from inside any other service. By default, Content Integration Framework publishes the message onto the topic configured in Platform configuration. As demonstrated in `ExampleKafkaProducerClient`, it is important to note that, caller of the service can completely override the target topic specifications if required.

AbstractEntity

The `com.hcl.unica.system.model.AbstractEntity` class represents a general domain entity. For the current release, this abstract class does not contain any implementation.

However, for the Content Integration Framework, plugins must extend their domain entities from the `com.hcl.unica.system.model.AbstractEntity` class. This ensures that `AbstractEntity` is the base for dealing with domain entities within Content Integration Framework.

As for the plugin implementations, the class used to represent an individual content returned by the `simple-search`, `list-contents`, and `get-content-details` services must extend from `AbstractEntity` class.

Presentable

To be able to render an individual content returned by the simple-search, list-contents & get-content-details services , the domain entity class used by these services must implement the `com.hcl.unica.system.model.presentation.Presentable` interface and override the `getPresentationDetails()` method. The `com.hcl.unica.system.model.presentation.Presentable $PresentationDetails` object returned by the `getPresentationDetails()` method must provide the `TextualPresentation` as well as `MultimediaPresentation` details.

`TextualPresentation` contains following particulars:

- 🖊️ **Note:** The highlighted fields are mandatory. For the other fields, provide details, if available.

- `heading` **– Title of the content**
- `subheadings` – List of subheadings for the content
- `summary` – Summary or description of the content
- `name` **– Should be used for filename associated with the content**
- `tags` – Tags associated with the content (out of the box plugins use this to convey MIME type or category of the content)

Whereas `MultimediaPresentation` contains following particulars:

- 🖊️ **Note:** The highlighted fields are mandatory. For the other fields, provide details, if available.

- `id` **- Unique identifier of the content**
- `folderId` **- Unique identifier of the folder respective content belongs to**
- `mimeType` **- MIME type of the original content**
- `size` - Size of original content in bytes
- `resourceUrl` **- Absolute URL to the original content**
- `thumbnailUrl` - Absolute URL to the content thumbnail, if available
- `fileName` **- File name associated with the original content**
- `type` **– Type/category identifier of the content (must be one of the values from supported content types set up using any of the applicable alternatives provided by Content Integration framework)**
- `list of variants` – Each variant supports almost same details as the primary MultimediaPresentation details except thumbnailUrl (it can only have its own resourceUrl), folderId and variants (variant cannot have any further variants)

Builder API

Almost all the standard types discussed in previous sections provide the builder API for the ease of constructing objects.

For example, `TextualPresentation` can be built using following syntax instead of splitting it into constructor and setter operations:

```
TextualPresentation.builder()
 .heading("Content title")
 .subheadings(Collections.emptyList())
 .name("photo.jpg")
```

```
    .tags(Collections.singletonList("Image"))
    .build();
```

It is not mandatory to use builder API for creating standard objects. However, it certainly keeps plugin implementations clean while dealing with complex objects.

Request Validation

The `AbstractService<RQ, RS>` interface declares a method to support request validation in service implementations. RESTful as well as Functional services implement `AbstractService`. Hence, services implemented using either of the approaches can override `void validateRequest(ExecutionContext<RQ> executionContext)` method to perform any validation on incoming request before serving it. Refer to the `com.example.service.rest.ExampleObjectExtensionService` class from `asset-integration-starter` project to learn how `validateRequest` method performs basic checks and throws `InvalidServiceRequestException` to convey validation failure. Validation failures can be conveyed using localized error messages. Please refer to the **I18n Support** section to learn about localization support.

Standard exceptions

Standard exceptions include exceptions provided by the Content Integration SDK, which can be used by the plugins to convey different failure conditions during service execution.

RESTful approach

Content Integration Framework handles error conditions, arising from services implemented using RESTful approach.

Additionally, Content Integration Framework initiates and handles the execution of remote API call for RESTful integrations, so that it can keep track of the success of all the HTTP operation. Thus, the plugins do not require any special exception to convey the failure of the REST call. If something goes wrong inside the service implementation; any appropriate unchecked exception is sufficient to convey the operation failure. Such exceptions are further conveyed as 502 HTTP response to the client.

Functional approach

Since Content Integration Framework does not initiate and manage the outgoing connections in case of Functional services, it cannot keep track of end to end success.

Hence, it provides certain standard exceptions, which the service implementations can throw to convey relevant failure conditions. These exceptions are related to communication with the target system and are present within the `com.hcl.unica.system.integration.exception` package.

- **SystemNotFoundException**

  This exception must be used when the target system or content repository cannot be located. Alternatively, `java.net.UnknownHostException` can also be used. This exception is conveyed as 404 HTTP response to the client.
- **ServiceNotFoundException**

  This exception must be used when the remote endpoint returns 404, or if the target service no longer exists. Absence of the target system and the absence of the required service are considered as different things. Hence, the

`ServiceNotFoundException` conveys presence of the target system and the absence of the required service, or feature, on the target system. For example, in case of content fetched from the database, the absence of the required table (or the absence of the permission to access it) can be conveyed using this exception. This exception is conveyed as 404 HTTP response to the client.

- **UnreachableSystemException**

This exception must be used to convey unreachable or inaccessible target systems, such as connection timeout. Alternatively, `java.net.ConnectException` can also be used. This exception is conveyed as 503 HTTP response to the client.

- **SluggishSystemException**

When the response from the target system is not received within expected time, this exception must be used to convey the slowness of the target system. Alternatively, `java.net.SocketTimeoutException` can also be used. This exception is conveyed as 504 HTTP response to the client.

- **InternalSystemError**

This exception must be used if the plugin receives a temporary, or unexpected, error from the target system to convey the problems in it. This exception is conveyed as 502 HTTP response to the client.

Any other exceptions are conveyed as 502 HTTP response to the client. In any case, the message in the exception is never returned to the client. Each HTTP response code carries a fixed, generic, and localized message.

Content Integration Framework wraps the exceptions received from service implementations into `com.hcl.unica.system.integration.exception.ServiceExecutionException` or its subtype. Exceptions received from REST-based services or HTTP-based services are wrapped in `com.hcl.unica.system.integration.exception.HttpServiceExecutionException`, whereas the ones received from Functional services are wrapped in `com.hcl.unica.system.integration.exception.ServiceExecutionException`.

As explained in Service invocation on page 23, `HttpServiceExecutionException` provides a method to obtain an `Optional<HttpResponse>` object. If the service execution fails before initiating an HTTP call, then this Optional object will not contain any `HttpResponse`.

Loggers

Content Integration Framework provides logging interface using the `slf4j` library. By adding dependency for the `slf4j` library, the plugins can use its API for adding loggers inside service implementations.

The starter as well as reference projects included in `dev-kits` manage their dependencies using Apache Maven. The following entry is found in the POM file:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.30</version>
</dependency>
```

Use 1.7.30 or higher version of slf4j-api to avoid conflict. Once the required dependency is added, the logger object can be obtained by directly accessing the `slf4j` API.

```
Logger log = LoggerFactory.getLogger(YOUR_CLASS.class);
```

Alternatively, project Lombok can also be used to get the logger object for your class. Lombok provides @Slf4j annotation, which can be used to inject the earlier mentioned property inside the annotated class. For more information on project Lombok, please visit its *official web page*.

Additionally, the application logs can be found in `AssetPicker/logs` directory under platform home. By default, all the loggers from your plugin will reside in the common log file configured in `AssetPicker/conf/logging/log4j2.xml` file. You can alter the `log4j2.xml` configuration file to route your loggers to a different file, for troubleshooting during development. Configuration of `log4j2` is not part of the scope of this guide. Please refer to the *official documentation of Apache Log4j2* for more information.

Chapter 4. Setting up the development environment

Set up the development environment in Eclipse IDE for writing your plugins. Use any Java EE IDE of your choice and make the required configurations mentioned in this topic. You need certain artifacts from `<ASSET_PICKER_HOME>` to complete the environment setup. This topic will provide information about project building and packaging using Apache Maven so ensure that you have Apache Maven installed.

About this task

To set up the development environment, complete the following steps:

1. From the `<ASSET_PICKER_HOME>/dev-kits/` location, copy the `asset-integration-starter` project and place it in your development workspace.
2. Open the Eclipse IDE.
3. Select **File > Import**.
   Result
   The **Select** dialog appears.
4. Select **WAR file** and click **Next**.
   Result
   The **WAR Import** dialog appears.
5. Click Browse, navigate to `<ASSET_PICKER_HOME>`, and `select asset-viewer.war` file.
6. Click **Finish**.
   Result
   The **WAR Import: Web libraries** dialog appears.
7. Click **Finish**.
8. Select **Window > Show View > Other**.
   Result
   The **Show View** dialog appears.
9. Select **Servers** and click **Open**.
   As an example, we will illustrate the use of Apache Tomcat 9.0 for running Content Integration. You can use any supported application server and make the required configurations.
   a. Open the `conf/server.xml` file from your Apache Tomcat 9.0 installation directory and add the following entry, with appropriate database details, inside the *<GlobalNamingResources>* element. Please replace *<DRIVER_CLASS_NAME>*, *<URL_TO_YOUR_PLATFORM_DATABASE>*, *<DATABASE_USERNAME>*, and *<DATABASE_PASSWORD>* with Platform database details:

```
<Resource auth="Container" driverClassName="{DRIVER_CLASS_NAME}"
                           maxActive="20"
                           maxIdle="0"
                           maxWait="10000"
                           name="UnicaPlatformDS"
                           password="{DATABASE_PASSWORD}"
                           username="{DATABASE_USERNAME}"
                           type="javax.sql.DataSource"
                           url="{URL_TO_YOUR_PLATFORM_DATABASE}"/>
```

    b. Open the `conf/context.xml` file from your Apache Tomcat 9.0 installation directory and add the following
       entry inside the *<Context>* element:

```
<ResourceLink auth="Container" global="UnicaPlatformDS" name="UnicaPlatformDS"
type="javax.sql.DataSource"/>
```

10. To add Apache Tomcat 9.0 as a new server in Eclipse, complete the following steps:

    a. In the **Servers** tab, click the link to create a new server.

      Result

      The **Define a New Server** dialog opens.

    b. Select **Tomcat v9.0 Server** and provide values for **Server host name** and **Server name**.

    c. Click **Next**.

      Result

      The server is successfully added.

    d. In the **Servers** tab, double-click the newly added server entry.

      Result

      The **Overview** dialog appears.

    e. Click the link **Open launch configuration**.

      Result

      The **Edit launch configuration properties** dialog appears.

    f. Edit the launch configurations to the add following JVM arguments

```
-DASSET_PICKER_HOME=<Point this to <ASSET_PICKER_HOME> directory>
-Dspring.profiles.active=platform-disintegrated
```

    g. Click **OK**.

11. To run the imported `asset-viewer.war` file on Apache Tomcat 9.0, right click the `asset-viewer.war` file and
    select **Run As > Run on Server**.

    Result

    The **Run on Server** dialog appears.

12. Click **Finish**.

    Result

    The `asset-viewer.war` will start executing on Apache Tomcat. After the setup is verified, stop the server and
    import the plugin development starter project.

13. To install Content Integration SDK, complete the following steps:

    a. In the following directories, delete the SDKs that are already installed:

          • `<LOCAL_M2_REPOSITORY>\com\hcl\unica\integration-api\0.0.1-SNAPSHOT`

          • `<LOCAL_M2_REPOSITORY>\com\hcl\unica\standard-integrations \0.0.1-SNAPSHOT`

          • `<LOCAL_M2_REPOSITORY>\com\hcl\unica\asset-integration-api\0.0.1-SNAPSHOT`

          • `<LOCAL_M2_REPOSITORY>\com\hcl\unica\entity-mapper-api\0.0.1-SNAPSHOT`

On UNIX or Mac OS X, `<LOCAL_M2_REPOSITORY>` refers to the `~/.m2/repository` directory.

On Microsoft Windows, `<LOCAL_M2_REPOSITORY>` refers to the `C:\Users\{your-username}\.m2\repository` directory.

b. Use the following commands to install Content Integration SDKs into your local Maven repository. Find `asset-integration-api.jar`, `integration-api.jar`, `standard-integrations.jar` and `entity-mapper-api.jar` inside the `<ASSET_PICKER_HOME>/dev-kits/sdk` directory.

```
mvn install:install-file –Dfile=<ASSET_PICKER_HOME>/dev-kits/sdk/asset-integration-api.jar
 -DgroupId=com.hcl.unica -DartifactId=asset-integration-api -Dversion=0.0.1-SNAPSHOT
 -Dpackaging=jar

mvn install:install-file –Dfile=<ASSET_PICKER_HOME>/dev-kits/sdk/integration-api.jar
 -DgroupId=com.hcl.unica -DartifactId=integration-api -Dversion=0.0.1-SNAPSHOT -Dpackaging=jar

mvn install:install-file –Dfile=<ASSET_PICKER_HOME>/dev-kits/sdk/standard-integrations.jar
 -DgroupId=com.hcl.unica -DartifactId=standard-integrations -Dversion=0.0.1-SNAPSHOT
 -Dpackaging=jar

mvn install:install-file –Dfile=<ASSET_PICKER_HOME>/dev-kits/sdk/entity-mapper-api.jar
 -DgroupId=com.hcl.unica -DartifactId=entity-mapper-api -Dversion=0.0.1-SNAPSHOT -Dpackaging=jar
```

14. To import the plugin development starter project, select **File > Import**.

    Result

    The **Select** dialog appears.

15. Select Existing Maven Projects and click **Next**.

    Result

    The **Maven Projects** dialog appears.

16. Click **Browse** to select the project and click **Finish**.

17. To update Maven dependencies of the `asset-integration-starter` project, right-click the `asset-integration-starter` project and select **Maven > Update Project**.

18. Ensure that newly imported project is using Java 8 to compile sources. Open project properties and complete the following steps to setup the compiler:

    a. Select **Java Compiler**.

    b. If Compiler compliance level is non-editable, select **Enable project specific settings**.

    c. Change the Compiler compliance level to `1.8`.

    d. Click **Apply and Close**.

19. To ensure that the right Java library is set up in the build path, complete the following steps:

    a. Select **Java Build Path > Libraries**.

    b. Select **JRE System Library (J2SE-1.5)**.

    c. Click **Remove**.

    d. Click **Add Library**.

       Result

       The **Add Library** dialog opens.

      e. Select **JRE System Library > Next**.

      Result

      The **JRE System Library** appears.

      f. Select an appropriate library and click **Finish**.

20. To enable annotation processing, complete the following steps:

      a. Select **Java Compiler > Annotation Processing**.

      b. Select **Enable project specific settings**.

      c. Select **Apply and Close**.

21. To install Lombok, complete the following steps:

      a. Double-click the `LOCAL_M2_REPOSITORY\org\projectlombok\lombok`
      `\1.18.16\lombok-1.18.16.jar`.

      Result

      The installer dialog appears.

      b. To specify the installation location of your IDE, click **Specify location**.

      c. To complete the installation, click **Install / Update**.

      d. Post installation of Lombok, restart the IDE.

22. To change the project name, complete the following steps:

      a. Open the file `pom.xml` and change its Maven project properties.

      b. Right-click the asset-integration-starter project and select **Refactor > Rename**.

23. In the `<ASSET_PICKER_HOME>/conf/custom-plugin-services.yml` file, declare the plugin services. You can access this file later to add declarations when you introduce services for your plugins.

24. To add plugin project to the deployment assembly of the `asset-viewer.war` project, complete the following steps:

      a. Right-click the `asset-viewer.war` project and select **Properties**.

      Result

      The **Properties for asset-viewer** dialog opens.

      b. Select **Deployment Assembly**.

      c. Select **Add**.

      Result

      The **Select Directive Type** dialog opens.

      d. Select **Project** and click **Next**.

      e. Select the `asset-integration-starter` plugin project you imported in previous steps and click **Finish**.

25. If necessary, clean the projects.

26. Make the appropriate configuration for your system in `<ASSET_PICKER_HOME>/conf/systems.properties` (refer `sample-systems.properties` file available in the `<ASSET_PICKER_HOME>/dev-kits/asset-`

`integration-starter` project). All the system onboarding configurations mentioned in *Unica Content Integration Administration Guide* are supported in `systems.properties` using relevant properties.

27. As you develop your plugin, check it by running the `asset-viewer.war` project on a previously configured application server. Since the project would already be added to the Deployment Assembly of `asset-viewer.war`, changes to your plugin project will be deployed whenever you run the `asset-viewer.war` project.

28. As you develop your plugin, by adding services to it, use a tool of your choice to hit the following REST endpoints (change the context root to match your setup) to verify the accuracy of your implementation:

   a. **Ensure system onboarding**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPicker/instances`
Request Method	`GET`

   b. **Verify simple-search service**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPicker/mysystem/assets?query=mountain&page=0&size=10&types=Phto` where, • `mysystem` represents the system identifier chosen by plugin implementation. • `query` contains the search keyword to lookup the content for. • `page & size` contains pagination details, where page is the serial number of pages to be retrieved and size is the total search items on a single page. • `types` is one of the supported content categories (types) to filter the search items against.
Request Method	`GET`

When you hit the URL, ensure that the response JSON contains the expected result. Only presentation details are included for every search items. Other content properties are excluded for the sake of brevity and performance.

   c. **Verify resource-loader service**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPicker/mysystem/download?resource=`

	`http://repository_base_url/contents/sample_ima`
	`ge.jpg%26resourceId=12345"`
	
	where
	
	    • `mysystem` represents the system identifier chosen by plugin implementation.
	    • `resource` contains the absolute URL content to be downloaded.
	    • `resourceId` contains the identifier of the content to be downloaded.
	
	(Plugin can choose to utilize either resource or resourceId or both to load the content.)

Request Method	`GET`

d. **Verify list-folders service**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPi`
	`cker/mysystem/folders?parentFolderId=1234`
	where:
	
	    • `mysystem` represents the system identifier chosen by plugin implementation.
	    • `parentFolderId` contains the identifier of the parent folder whose immediate subfolders are expected in response. This query parameter is optional & not supplied while listing the top/root level folders.

Request Method	`GET`

e. **Verify list-contents service**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPi`
	`cker/mysystem/folders/1234/contents`
	where:
	
	    • `mysystem` represents the system identifier chosen by plugin implementation.
	    • `1234` represents the identifier of the folder whose immediate contents are expected in response.

Request Method	`GET`

Only presentation details are included for every content listed by the `list-contents` service. Other content properties are excluded for the sake of brevity and performance.

f. **Verify get-content-details service**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPi cker/mysystem/assets/Images/1234` where: • `mysystem` represents the system identifier chosen by plugin implementation. • `Images` represents the category ID of the content whose details are expected in response. • `1234` represents the identifier of the content whose details are expected in response.
Request Method	`GET`

The JSON response produced by `get-content-details` service includes all the content properties, in addition to the presentation details.

g. **Verify get-object-schema service**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPi cker/object-mapping/application/mysystem/objec t/Images/1234/schema` where: • mysystem represents the system identifier chosen by plugin implementation. • Images represents the category of the reference content being used for schema generation. • 1234 represents the identifier of the reference content being used for schema generation. As of 12.1.0.4, content identifier and category are not much relevant since the schema is expected to include attributes for all the supported content categories.
Request Method	`GET`

The JSON response must contain the flattened list of all mappable attributes and their metadata.

h. **Verify list-content-categories service**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPicker/mysystem/categories` where: • `mysystem` represents the system identifier chosen by plugin implementation.
Request Method	`GET`

i. **Verify get-cognitive-analysis service**

Endpoint URL	`http://localhost:8888/asset-viewer/api/AssetPicker/actions/cognize?url=absolute_image_url` where: • `url` contains absolute URL of the image to fetch the cognitive analysis for
Request Method	`GET`

Chapter 5. Verification and troubleshooting

To verify end-to-end integration, place the `JAR` file, containing the plugin implementation, in the class path of the application server where the Content Integration is deployed. Additionally, configure the corresponding content repository in `<ASSET_PICKER_HOME>/conf/systems.properties` file (you can refer to the `sample-systems.properties` file within the `<ASSET_PICKER_HOME>/dev-kits/asset-integration-starter` project).

All the system onboarding configurations, mentioned in *Unica Content Integration Administration Guide*, are supported in the `systems.properties` using relevant properties. You must provide `-Dspring.profiles.active=platform-disintegrated` JVM argument for `systems.properties` to come into effect (you can always use Platform's configurations instead of systems.properties by removing `-Dspring.profiles.active=platform-disintegrated` JVM argument).

> ✏️ **Note:** Currently, only Unica Centralized Offer Management and Unica Plan can access Content Integration.

After the plugin is deployed, and the system configurations are made, restart the Content Integration application.

Although, you can verify Content Integration using REST endpoints mentioned in previous section, we recommend you to check end-to-end integration by running through the relevant user interface in Unica Centralized Offer Management and Unica Plan. Please refer to the corresponding user guides to learn how to access Content Integration features in respective products.

Use developer tools provided by the supported browsers to troubleshoot the API calls, if required.

Overview of loggers

As mentioned in Verification of integration, the logging configuration for Content Integration is available in the `log4j2.xml` file, placed in the `AssetPicker/conf/logging` folder within Platform home.

Content Integration uses Apache `Log4j2` for log management. The `RandomAccessFilePlatform` appender along with `com.unica` logger configured in `log4j2.xml` controls the logs produced by Platform's `unica-common.jar` and `unica-helper.jar` used in Content Integration. The remaining settings control logging for other core activities of Content Integration.

The default log level is set to `WARN` in both cases, which should be sufficient for the troubleshooting needs for plugin development. Most of the loggers, produced by the Content Integration at INFO & DEBUG level, are not extremely relevant for plugin development & integration. The following topics elaborate only the relevant loggers. These loggers are already present in `log4j2.xml` file and need to be uncommented, if required. Please ensure that log level is never set to `DEBUG` or `TRACE` for these loggers in production since they can generate sensitive information.

The `log4j2.xml` file also contains necessary configurations to route all the loggers for a specific user to a dedicated log file. By default, these configurations are commented. Appropriate description is added in `log4j2.xml` at the top of each configuration element to help activate the dedicated log file.

Useful loggers in log4j2.xml file

The following table lists the useful loggers in the `log4j2.xml` file:

Table 2. Useful loggers in log4j2.xml file

Loggers	Information
`org.springframework.web`	Setting this logger to TRACE level produces HTTP request and response details for all the incoming HTTP requests to Content Integration. This logger can be useful if you want to see what is being exchanged between frontend and backend.
com.hcl.unica.cms.integration .flow.interceptor.logger	This logger is most useful for plugin development. It logs the HTTP interaction between Content Integration Framework and the target repository. For any service implemented using RESTful approach (by implementing RestService, HTTPService or their specialized derivatives), this logger will write HTTP request and response details for all the outbound HTTP interactions with target system. To prevent security vulnerability, values of confidential headers are masked before logging. Only the last four characters are left unmasked for troubleshooting. Such headers include standard header Authorization, or any non-standard custom headers set in request or received in response.
`org.springframework.retry`	Setting this logger to `TRACE` level adds information related to retrial attempts while making HTTP calls to the target repository. This is useful to verify Retry Policy set up under QOS section for the respective system in Platform Configuration.
`com.hcl.unica.system.integration.flow.object.event &` `com.hcl.unica.system.integration.service.object.event`	Setting this logger to INFO level produces logs pertaining to the content event processing. Each content event received via webhook is assigned a unique identifier. Every logger thus produced contains the unique identifier to help tracing end to end processing of each event. Every log line begins with "Event#aa09af83-0f52-4d0c-82de-370e96a0f4c2", wherein the part after hash sign is unique event identifier. These event processing logs are recorded in a separate file named `asset-picker-webhook.log` under `<ASSET_PICKER_HOME>/logs` directory.

Other important loggers

Other important loggers are useful in troubleshooting Content Integration. Along with spotting warnings and errors, these loggers provide information that is useful from a functional point of view.

The following table lists the other important loggers:

- **Client applications** - If root logger level is set to INFO level, the following lines tells you the number of client applications, and which client applications Content Integration can identify:

```
SupportedClientApplications: Found {1} supported client applications.
SupportedClientApplications: Registered {Offer} as supported client application.
```

- **CORS** - If root logger is set to INFO level, the following lines can provide information about Content Integration's support for Cross Origin Resource Sharing:

```
RegexCorsConfig: CORS: Enabling CORS for {hcl.com} & its subdomains. Allowed HTTP methods - {[GET,
 POST]}, allowed headers - {[*]}
RegexCorsConfig: CORS: Allowed origins set to {[http(s)?://([^\.]+\.)*hcl.com(:[0-9]+)?]}
```

- **Platform configuration - Content repositories** - Setting the root logger level to INFO tells us about the content repositories that are identified by Content Integration Framework.

```
PlatformConfigurationCategoryResolver: Platform configuration: Reading list of entries for path
 {Affinium|Offer|partitions|partition1|Content Integration|dataSources}...
PlatformCmsConfigurationReader: Platform configuration: Imported settings for {AEM#119[partition1]}
PlatformCmsConfigurationReader: Platform configuration: Imported settings for {WCM#119[partition1]}
PlatformCmsConfigurationReader: Platform configuration: Imported settings for {Bing#119[partition1]}
```

- **Service meta information files** - The following lines are also logged at INFO level to tell how many service meta information files have been identified by Content Integration Framework:

```
c.h.u.s.c.s.PluginServicesYamlConfigReader: Scanning & parsing service configuration files.
c.h.u.s.c.s.PluginServicesYamlConfigReader: Seeking file at
 {<ASSET_PICKER_HOME>\conf\plugin-services.yml}.
c.h.u.s.c.s.PluginServicesYamlConfigReader: Found service config file at
 {<ASSET_PICKER_HOME>/conf/plugin-services.yml}
c.h.u.s.c.s.PluginServicesYamlConfigReader: Parsing service configuration file (YAML):
 {<ASSET_PICKER_HOME>/conf/plugin-services.yml}...
c.h.u.s.c.s.PluginServicesYamlConfigReader: Seeking file at
 {<ASSET_PICKER_HOME>\conf\custom-plugin-services.yml}.
c.h.u.s.c.s.PluginServicesYamlConfigReader: {1} service declaration(s) found for {COM} -
 {[COM:get-object-schema]}
c.h.u.s.c.s.PluginServicesYamlConfigReader: {12} service declaration(s) found for {WCM} -
 {[WCM:item-details, WCM:simple-search, WCM:content-list, WCM:logon-service, WCM:list-contents,
 WCM:library-list, WCM:get-content-details, WCM:folder-list, WCM:get-object-schema, WCM:list-folders,
 WCM:library-by-id, WCM:resource-loader]}
c.h.u.s.c.s.PluginServicesYamlConfigReader: {31} service declaration(s) found for {Deliver} -
 {[Deliver:update-folder, Deliver:simple-search, Deliver:list-by-ids, Deliver:zip-file-upload,
 Deliver:delete-content, Deliver:move-folder, Deliver:create-content, Deliver:list-folders,
 Deliver:zip-upload-template-unknown, Deliver:move-content, Deliver:list-sub-folders,
 Deliver:download-content-variant, Deliver:download-file-attachment, Deliver:get-user-entitlements,
 Deliver:list-top-folders, Deliver:update-dynamic-content, Deliver:create-folder,
 Deliver:find-libraries-by-name, Deliver:resource-loader, Deliver:zip-upload-content,
 Deliver:adopt-dynamic-content, Deliver:get-folder, Deliver:create-dynamic-content,
 Deliver:list-contents, Deliver:get-content-details, Deliver:patch-content, Deliver:delete-folder,
 Deliver:get-library, Deliver:update-content, Deliver:get-library-file, Deliver:adopt-content]}
```

```
c.h.u.s.c.s.PluginServicesYamlConfigReader: {1} service declaration(s) found for {Azure} -
 {[Azure:get-cognitive-analysis]}
c.h.u.s.c.s.PluginServicesYamlConfigReader: {1} service declaration(s) found for {DX-CORE} -
 {[DX-CORE:logon-service]}
c.h.u.s.c.s.PluginServicesYamlConfigReader: {7} service declaration(s) found for {DX}
 - {[DX:simple-search, DX:list-contents, DX:get-content-details, DX:rendition-details,
 DX:get-object-schema, DX:list-folders, DX:resource-loader]}
c.h.u.s.c.s.PluginServicesYamlConfigReader: {7} service declaration(s) found for {Commerce}
 - {[Commerce:simple-search, Commerce:list-contents, Commerce:get-content-details,
 Commerce:get-search-query-suggestions, Commerce:list-content-categories, Commerce:get-object-schema,
 Commerce:list-folders]}
c.h.u.s.c.s.PluginServicesYamlConfigReader: {7} service declaration(s) found for {AEM} -
 {[AEM:simple-search, AEM:list-contents, AEM:get-content-details, AEM:get-object-schema,
 AEM:get-content-fragment-model, AEM:list-folders, AEM:sample-inbound-service]}
c.h.u.s.c.s.PluginServicesYamlConfigReader: {2} service declaration(s) found for {Bing} -
 {[Bing:simple-search, Bing:get-content-details]}
```

- **Authentication protocols** - The following lines, logged at INFO level, confirms the authentication protocol is identified for the given content repository:

```
AssetPickerRestTemplate: Setting up {BASIC} authentication for {Offer[partition1].WCM:simple-search}
 service...
```

- **Platform configuration cache invalidation and service re-initializations** - All the Platform configurations for Content Integration are cached during application startup. These configurations are refreshed after certain interval (every 30 mins by default unless configured to use some other interval). The following logger is produced at INFO level, whenever configuration refresh begins:

```
INFO  [scheduling-1] c.h.u.s.c.s.ServiceBootstrapper: Re-initializing services...
```

Similarly, the following lines are generated at INFO level whenever it is over:

```
INFO  [scheduling-1] c.h.u.s.c.s.ServiceBootstrapper: Finished service initializations.
INFO  [scheduling-1] c.h.u.s.c.s.ServiceBootstrapper: Re-initialization completed in 3692 milliseconds.
 YAML read time: 15 milliseconds, DB Read Time: 3608 milliseconds, Service initialization time: 68
 milliseconds
```