Unica Asset Picker
V12.0 Developer's Guide

Contents

Chapter 1. Unica Asset Picker Developer Guide

This guide provides information on plugin development and troubleshooting of Unica Asset Picker.

Overview

Asset Picker facilitates easy integration with Content Management Systems and enables searching content from the Content Management Systems.

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

Plugins

Asset Picker integrates with different CMS using REST APIs. It addresses the challenge of programming interface disparity between different systems by leveraging the custom plugins or modules written specifically for the target system.

You can implement plugins using Java programming language. Asset Picker 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 Asset Picker to fetch desired content from external content store.

Integration support and plugin development approach

Asset Picker 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 Asset Picker. 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

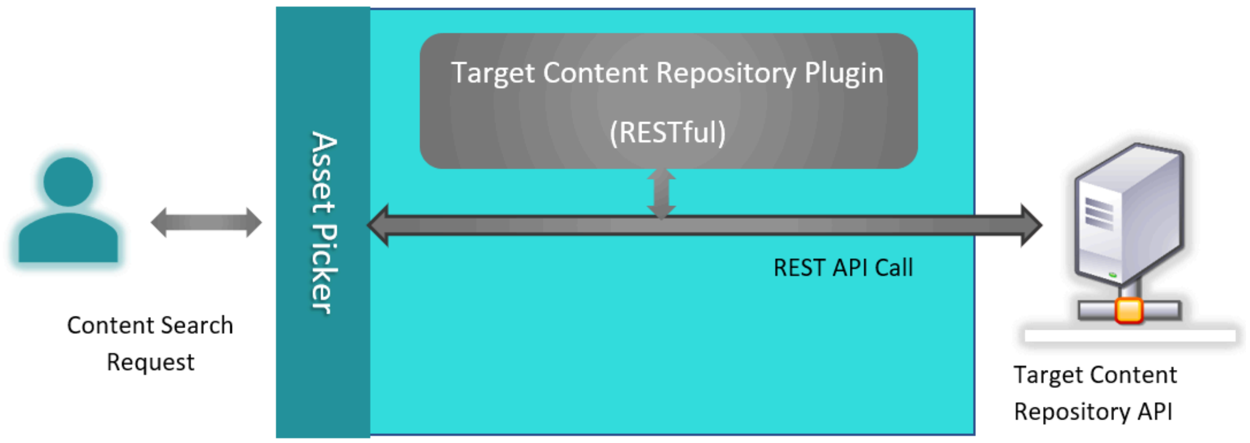
An alternate plugin development approach for non-RESTful integration involves thorough implementation. For example, a plugin written for fetching content from database needs to address everything involved in making DB connection, executing SQLs, closing connections, result set hydration, failure handling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

Plugins do not initiate the content search. Asset Picker first receives the search request, which is delegated to the respective plugin. In case of RESTful integrations, Asset Picker initiates the HTTP interaction and gathers the necessary information from the plugin, when required.

RESTful content search flow

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

Figure 1. RESTful content search flow

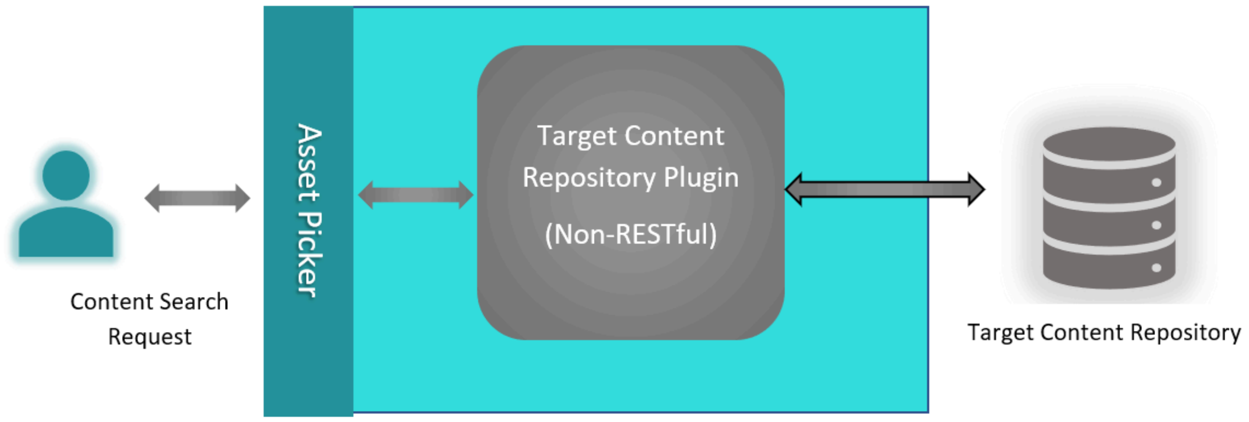


When Asset Picker receives content search request from user for the target system, it consults with the respective plugin to gather request specific logical information and makes an API call to the target system. It consults with the plugin once again to transform the API response into an expected format 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 Asset Picker. Unlike RESTful repositories, Asset Picker will not know the type, architecture, protocol and the authentication mechanism used for communicating with the target repository.

Plugin development overview

Asset Picker facilitates easy integration with new content repositories without having to alter the core Asset Picker framework.

Asset Picker seamlessly integrates with system-specific, independent plugins. Once the plugin is developed and included in the classpath of the application server hosting Asset Picker, the corresponding system can be onboarded in the Unica product suite by updating a few configurations in Unica Platform. For more information, see Unica Asset Picker Administrator's Guide

Asset Picker 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 `AssetPicker/dev-kits` directory within Platform home. Two reference projects, named `aem-integration` and `wcm-integration`, are available for Adobe Experience Manager (AEM) and IBM Web Content Manager (WCM) respectively. To write a plugin for new system, we recommend you use the starter project to save writing boilerplate code.

Components of plugin

A typical plugin contains the following components:

- Service meta information file *(on page 5)*
- Service implementations *(on page 11)*

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.

Service meta information file is an YML configuration file containing the list of services included in the plugin. A service can either be a standard service or a custom service.

Standard services carry special semantics and purpose in Asset Picker. Implementation of certain standard services is mandatory for Asset Picker to work with the content repository.

Service meta information file

The following are the pre-requisites for meta information file:

- Service meta information file is expected inside the `META-INF` directory on project class path.
- Name of the meta information file must end with `content-services.yml` suffix. Examples are:
    - `wcm-content-services.yml`
    - `aem-content-services.yml`
    - `example-content-services.yml`

Reference files can be found inside `aem-integration`, `wcm-integration`, and `asset-integration-starter` projects under the following listed locations:

- `dev-kits\aem-integration\src\main\resources\META-INF`
- `dev-kits\wcm-integration\src\main\resources\META-INF`
- `dev-kits\asset-integration-starter\src\main\resources\META-INF`

The following is the example content of a file from asset-integration-starter project:

```
services:
  -
    systemId: Foo
    serviceName: simple-search
    factoryClass: com.example.service.rest.SimpleSearchService

  -
    systemId: Foo
    serviceName: resource-loader
    factoryClass: com.example.service.rest.ResourceLoaderService
```

```
-
  systemId: Foo
  serviceName: asset-selection-callback
  factoryClass: com.example.service.rest.ContentSelectionCallbackService



-
  systemId: Foo
  serviceName: custom-service
  factoryClass: com.example.service.rest.CustomService
```

Service declarations

Meta information document begins with services key, which is an array of dictionaries containing three elements named systemId, serviceName and factoryClass. Details of the elements are as follows:

- `systemId`

  This string value uniquely identifies a target content repository. This identifier should preferably contain only alphanumeric characters. Dots, dashes, and underscores can be used to add readability. 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
  ```

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 different 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. Asset Picker does not impose any restrictions. Likewise, entries for one plugin can be split into multiple meta information files as long as file names end with the `content-services.yml` suffix.

- `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. The following is the list of standard service names:

  - `simple-search`
  - `resource-loader`
  - `asset-selection-callback`
- `factoryClass`

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

Standard services

The following table provides and introduction to the standard services of Asset Picker:

Table 1. Standard services and their description

Standard service name	Description
`simple-search`	Simple search service responds to the content search requests received by Asset Picker. 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.
`resource-loader`	This service is useful whenever indirect access to the search item is required. This service is not mandatory and should be implemented only when following challenges are encountered: 1. The return value of the `simple-search` service includes a URL (either relative or absolute) to the respective search item so that Asset Picker client can load its contents over the web. If there is no direct web link to the search item, compose a URL that concerns the search item, which will come back to the Asset Picker whenever the contents of that item need to be loaded by the client. Such URL

Standard service name	Description
	should contain a unique identifier for the search item, which is used by the `resource-loader` service to read the resource content. For example, if the target repository is a database, then the `simple-search` service will fetch records from the database that matches the given search query. Since the items are loaded from the database, there may not be any URL directly pointing to each record. In such cases, the `simple-search` service composes a URL relative to the Asset Picker context root that includes an identifier of the item. Subsequent requests to load individual search item will go through Asset Picker, which will delegate the resource loading task to `resource-loader` service. Additionally, the resource-loader service identifies the resource, or item, based on the identifier, the `simple-search` service had supplied in the URL. This URL allows the `resource-loader` service to read the contents of search items and respond. 2. If the target system does not provide anonymous access to search items, the `resource-loader` service is used as a makeshift provision to fetch the contents of individual items. The

Standard service name	Description
	`resource-loader` service authenticates itself before fetching the resource contents from the target repository. If the `resource-loader` service is implemented using RESTful approach, Asset Picker will take care of the authentication out-of-the-box, subject to the configurations in Unica Platform. (For more information on repository onboarding configurations, see *Unica Asset Picker Administrator's Guide*). **Note:** On production, routing resource loading requests using Asset Picker, in either way, is not recommended. The usage of `resource-loader` service should be restricted to demonstrations or development phase only. Content is expected to be anonymously accessible in the target system for seamless content consumption.
`asset-selection-callback`	The service is useful when the plugin performs any action in response to the selection of any search item by the client. When the `simple-search` service returns search results to the client, one of the search items is chosen by the client for further use cases. The Plugin might check which item was chosen by the client to perform the corresponding action (for example, locking that item in underlying

Standard service name	Description
	repository so that it cannot be updated or deleted etc.). This service is not mandatory and should be implemented only if callback operation is necessary.

Service implementations

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

The Asset Picker provides an SDK to streamline the service implementation and facilitates rapid development of plugins. The Asset Picker 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 Asset Picker SDK. For additional details, see SDK Plugin Development (on page 16).

RESTful approach

The `com.example.service.rest.CustomService` 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 Asset Picker 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 the input given to the service being implemented.

- **String getEndpointUrl(ExecutionContext<RQ> executionContext)**

  This method returns an absolute endpoint URL of the service running on target system. Base URL of target system is configured in Unica Platform. Hence, the plugin need not make any provision to configure that in any way. `ExecutionContext` object supplied to this method provides a way to read the base URL so that the absolute URL of the service can be composed. Also look at how the `getEndpointUrl` method is defined in `com.aem.service.AemSimpleSearchService` class inside `aem-integration` project. As it can be noted, the base URL is obtained from `ExecutionContext` by navigating through `InstanceConfig` object. The `InstanceConfig` holds all the configurations made in Unica Platform for the very target system instance your service will communicate with. This is a mandatory method for the service to implement.
- **HttpMethod getHttpMethod()**

  This method should return one of the values from the `HttpMethod` enum supplied with the Asset Picker SDK. As the name goes, this method tells which HTTP request method should be used during HTTP interaction with target system. This is a mandatory method for service to implement.
- **Map<String, Object> getHeaders(ExecutionContext<RQ> executionContext)**

  This optional method can be overridden by the service if it wants to include any HTTP request headers in the outgoing HTTP call. Return value must be a Map instance, wherein HTTP 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 implementation, no custom headers will be sent along with the outgoing HTTP request.

  Note: Although the Map returned by this method accepts values of type Object (or its subtypes), only String objects are supported as of current implementation of Asset Picker. 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.
  ```

    - `Content-Type` HTTP header must be populated as `contentType` key due to special considerations in underlying framework.
    - `application/json` is the default contentType for RESTful services, if none is supplied by the `getHeaders` method.
- **Object buildRequest(ExecutionContext<RQ> executionContext)**

  This is also an optional method. If the target service expects any request body, then this method can be overridden to build the desired HTTP request body. Return type of this method is Object, and hence any type of valid request body can be supplied so long as relevant `Content-Type` header is populated using the `getHeaders` method.

  Note: Jackson and JAXB Support - Object serialization using Jackson and JAXB is completely supported by Asset Picker. Thus, appropriately decorated object with Jackson or JAXB annotations can be returned from this method. In such case, no `Content-Type` header is required to be populated explicitly. Asset Picker takes care of supplying appropriate header during HTTP invocation. Serialization of supplied object into the request body is also handled by Asset Picker itself, hence no explicit serialization is required.

  In the absence of this implementation, empty request body will be sent along with the outgoing HTTP request.
- **Object transformResponse(RS response, ExecutionContext<RQ> executionContext)**

  This optional method transforms the HTTP response into a desired format. The additional, first argument to this method is the HTTP response body received from the target service. This argument is a generic type and is decided based on the actual type parameter used while implementing the service. This response can be any object, either a String containing the text as received from the service, a byte array containing the response contents or a deserialized object representing the response JSON/XML.

  Note: Jackson and JAXB Support - Object deserialization using Jackson and JAXB is completely supported by Asset Picker. 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 Asset Picker, 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 Asset Picker.

In addition to these methods, there is one more method the `getServiceInterface` inherited from `com.hcl.unica.cms.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.

Asset Picker 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 Asset Picker. 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.CustomService 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 Asset Picker. 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 the input given to the service being implemented.

- **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.

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 Asset Picker in this regard

    b. Limitations
        - Cannot be used for non-RESTful or non-HTTP integrations, such as database or file system interactions
2. **Functional approach**
    a. Advantages
        - Can be used for non-RESTful or non-HTTP integrations, such as database or file system interactions

    b. Limitations
        - No out-of-the-box support available for transport level error handling, retrials, proxied connectivity, and any future enhancements from Asset Picker

- Incorporation of logic for all the missing out of the box supports can make functional service very verbose

You can see that the Function 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 Asset Picker. While Functional approach gives flexibility in terms of implementation design, it takes away a few useful capabilities.

SDK Plugin Development

SDK plugin development provides information about the various classes, interfaces, and enums from the Asset Picker SDK, with the help of corresponding logical units in `asset-integration-starter`, `aem-integration`, and `wcm-integration` reference projects that is embedded as a part of development kit along with the Asset Picker application.

Generic type parameters

Generic type parameters are used for implementing service interfaces. For more information on service interfaces, see Service implementations (on page 11).

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, takes the requested content and produces the desired response. It requires certain generic notations for the inputs and outputs exchanged during end-to-end logical flow.

Asset Picker uses RQ to denote certain inputs to the service, and RS to denote either output of the service or response of the remote REST API. The definition of RS might change based on where it is used.

RestService<RQ, RS>

Refer the `com.example.service.rest.CustomService` 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.CustomService` 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 String getEndpointUrl(ExecutionContext<ServiceInput>
   executionContext) {
  ServiceInput input = executionContext.getRequest();
  // Remaining implementation omitted for brevity
  }
  ```
- **RS**

  This type parameter 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 `com.example.service.rest.CustomService` class, you will see that the object of `ApiResponse` type is supplied as the first argument, which corresponds to the RS type parameter of `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 `FunctionalInterface` 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>

This interface is used for implementing the `getServiceInterface` method from `AbstractService<RQ, RS>` interface. `AbstractService` is an important interface of `RestService`, or `HttpService`, and the `FunctionalService`. Semantics for RQ and RS for AbstractService are same as `RestService`, or `HttpService`. It declares the `getServiceInterface` method, which must be implemented by a service. This is the only additional method a RESTful service needs to implement and it returns the class object of the derivative (child interface) of `ServiceGateway`. The definition of `com.hcl.unica.cms.integration.service.gateway.ServiceGateway` is as follows:

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

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.CustomService` 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.

Note: `getServiceInterface` method in `com.example.service.rest.CustomService` 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.

Service invocation

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

The `invocationDemo` method in this class obtains the reference to `custom-service` by using `getServiceGateway` static method from `ServiceGatewayFactory` class. The `getServiceGateway` method takes three arguments to return the service instance. The arguments are as follows:

- `String systemId`

  This system identifier is same as the one used in service meta information file to declare the service whose instance needs to be obtained.
- `String serviceName`

  This is name of service whose instance needs to be obtained. It must be same as the one declared in service meta information file.
- `Class<T> gatewayClass`

  This must be the Class object of ServiceGateway interface (or its child interface). It must match the return value of getServiceInterface method in corresponding service implementation.

The invocationDemo method in the com.example.service.client.CustomServiceClient class uses CustomServiceGateway (gatewayClass) to obtain the service instance of custom-service (serviceName) for the system Foo (systemId). The following code snippet is for your reference:

```
public void invocationDemo() {
  String systemId = "Foo";

  CustomServiceGateway customService =
ServiceGatewayFactory.getServiceGateway(
    systemId,
    "custom-service",
    CustomServiceGateway.class
   );

  ServiceInput input = new ServiceInput();
  ServiceOutput output = customService.execute(input);
 }
```

The return type of getServiceGateway is also CustomServiceGateway. Service instance obtained can be used to execute the respective service by supplying the required input object.

Note:

- The return type of `getServiceGateway` is also `CustomServiceGateway`. Service instance obtained can be used to execute the respective service by supplying the required input object. Execute method is used on `ServiceGateway` interface to execute the service. You will observe that the type of the input to `custom-service` is same as the type used for service implementation in the `com.example.service.rest.CustomService` class or the `com.example.service.functional.CustomService class`. The type of output is the same as the one used for defining `CustomServiceGateway` interface whose Class object is returned from `getServiceInterface` method in both versions of `CustomService` class.
- The `com.example.service.rest.CustomService` class and the `com.example.service.functional.CustomService` class represents the same service implemented with two different approaches. The service meta information files in `asset-integration-starter` project using 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 Asset Picker. Irrespective of the approach used for service implementation, the method for service invocation remains the same.

Multi-partitioned clients

In case of multi-partitioned client application of Asset Picker, the earlier mentioned method of obtaining the service instance will appropriately return reference partition specific to the service gateway. The `ExecutionContext` object passed to various callback methods will contain the necessary partition specific information to work with.

Execution context

Almost every method in service implementation contract receives an instance of `com.hcl.unica.cms.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 execute method on the `ServiceGateway` interface. (The T return type is the type parameter corresponding to the generic argument used for defining the service.)
- **Map<String, Object> getAttributes()**

  Returns additional attributes pertaining to the current service execution, such as HTTP response status & headers for current HTTP call.

  Note: `Content-Type` HTTP header is populated as `contentType` key due to special considerations in underlying framework.
- **ServiceConfig getServiceConfig()**

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

- **InstanceConfig getInstanceConfig()**

  This method returns an instance of `com.hcl.unica.cms.integration.config.InstanceConfig` class. This object contains all the configurations made in Unica Platform for the target system (instance in this method name refers to the target system instance, and not the service instance). In case of multi-partitioned configurations, this object will be appropriately populated by Asset Picker to hold partition specific configuration. To know the various instance configuration settings in Unica Platform, see Unica Asset Picker Admin Guide.
- **void setAttributes(Map<String, Object>)**

  Use of this method is limited to Asset Picker only. Avoid using this method with the Plugins.

User data source

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

```
executionContext.getInstanceConfig().getDataSourceCredentials()
```

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

Similarly, the `getUnicaToken` method called on `InstanceConfig` object returns a UnicaToken object containing the Unica Token required for invoking APIs of Unica applications.

Standard services and specialized types

The plugin developer needs to implement `RestService/HttpService` or `FunctionalService` interface to create an individual service.

Also, the implemented service can be executed by obtaining reference to it using `ServiceGatewayFactory.getServiceGateway` method. The Asset Picker leverages this design and defines certain standard services. These are Simple Search (`simple-search`), Resource Loader (`resource-loader`), and Asset Selection Callback (`asset-selection-callback`) services. The Asset Picker provides specialized interfaces extended from `RestService` and `FunctionalService` for each of these standard services to facilitate their implementation using RESTful or Functional approach.

Invocation of standard services

Once declared in service meta information file and implemented using either RESTful or Functional approach, Asset Picker invokes it in the following scenarios:

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

  Whenever Asset Picker receives content or asset search request from its client application against target system, it invokes the `simple-search` service implemented for respective system. Asset Picker 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 meta information file and the corresponding **System Identifier** setting in Unica Platform 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.
- **Resource Loader (`resource-loader`)**

  The `resource-loader` service is executed by Asset Picker only when indirect (or authenticated) access needs to be made to the search item on target system. With the absence of a direct URL to an individual search item on the target system, the `resource-loader` service can be implemented by the plugin. Being an optional service, Asset Picker automatically recognizes if the `resource-loader` implementation exists for the target system and invokes it accordingly.

- **Asset Selection Callback (`asset-selection-callback`)**

  Asset Picker executes this service for the target system when the client application selects one of the search items returned by the `simple-search` service. Because the service is optional, Asset Picker can automatically recognize the existence of this service and invoke accordingly.

Specialized types

The following are the specialized derivatives of `RestService`, `HttpService`, and `FunctionalService` 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 *(on page 24)*
- Derivatives of HttpService *(on page 28)*
- Derivatives of FunctionalService *(on page 30)*

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.cms.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.cms.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 Asset Picker 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 Asset Picker SDK and encapsulates the search result and associated pagination details.

The plugin must extend its `simple-search` implementation from `RestSearchService` service in order to be recognized as a `simple-search` service by Asset Picker.

`RestSearchService` extends from `com.hcl.unica.cms.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.

- `com.hcl.unica.cms.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.cms.integration.service.gateway.SimpleSearchServiceGateway` interface as the service gateway for the `simple-search` service. ServiceGateways are the means to programmatically define input and output types of the service and the work with the 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 abstract method for the `simple-search` service. Every `simple-search` implementation must override and implement this new 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 new method is as follows:

```
abstract public List<String> getSupportedContentTypes();
```

Implementation of this method returns a list of strings representing the categories of contents or assets to search against in target system. There is no specific semantic associated with the values in this list. It can be any meaningful text. It acts as a filter for client application during search operation. Client application can send values from this list to filter the search items. Values received from the client application can be obtained from the `ExecutionContext` object by navigating through the `getRequest` method and then calling `getTypes()` on it. `getRequest()` returns the `SearchRequest` object which contains the set of supported types the client application has sent to filter the search result. Simple-search implementation deals with these set of values as per the target system's programming interface and filters the search items accordingly. Look at the `getSupportedContentTypes` method in `com.aem.service.AemSimpleSearchService` class in `aem-integration` project, and how the `restrictContentTypes` method in `com.aem.service.simplesearch.SimpleSearchRequestBuilder` class restricts the search result to the selected types.

Asset selection callback (`asset-selection callback`)

The following are the specialized interfaces and classes available for `asset-selection-callback` service:

- `com.hcl.unica.cms.integration.service.assetselectioncallback`

```
.RestAssetSelectionCallbackService
```

The `com.example.service.rest.ContentSelectionCallbackService` class in the `asset-integration-starter` project is a quick starter implementation for RESTful `asset-selection-callback` service. Its parent is the following class:

```
com.hcl.unica.cms.integration.service.assetselectioncallback
.RestAssetSelectionCallbackService
```

The `RestAssetSelectionCallbackService` class has a type parameter RS, which represents the type of response (post deserialization) received from remote REST API. In this case it is String class defined in Standard Java Library.

The `RestAssetSelectionCallbackService` class implements the `RestService` interface and defines the `com.hcl.unica.cms.model.request.assetselectioncallback.AssetSelectionDetails` class to be the type argument RQ for RestService. Thus, the object of `AssetSelectionDetails` becomes the input to all the `asset-selection-callback` services (Same input is used for Functional counterpart of asset-selection-callback as well). The `AssetSelectionDetails` class is a part of the Asset Picker SDK. The `AssetSelectionDetails` class encapsulates the details of an Asset (search item) selected by the client application and the contextual information such as the search query that lead to the search result containing the selected item.

Plugin must extend its `asset-selection-callback` implementation from the `RestAssetSelectionCallbackService` service in order to be recognized as an `asset-selection-callback` service by the Asset Picker (functional counterpart is also a valid choice to extend from).

`RestAssetSelectionCallbackService` extends from the following abstract class:

```
com.hcl.unica.cms.integration.service.assetselectioncallback
.AbstractAssetSelectionCallbackService
```

-
  ```
  com.hcl.unica.cms.integration.service.assetselectioncallback
  .AbstractAssetSelectionCallbackService
  ```

This is a common base class for RESTful as well as Functional `asset-selection-callback` implementations. So, the details of this class mentioned here also applies to the Functional implementation of `asset-selection-callback`.

The following class defines interface as the service gateway for the `asset-selection-callback` service:

```
com.hcl.unica.cms.integration.service.gateway
.AssetSelectionCallbackServiceGateway
```

ServiceGateways are the means to define input and output types of the service and programmatically work with the service. A closer look at this interface tells us that the `asset-selection-callback` takes the `AssetSelectionDetails` object and returns any Object. Currently, return value from `asset-selection-callback` is ignored by the Asset Picker.

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.cms.integration.service.resourceloader.DefaultWebResourceLoaderService`
  The `com.example.service.rest.ResourceLoaderService` 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.cms.integration.service.resourceloader
.DefaultWebResourceLoaderService
```

`DefaultWebResourceLoaderService` class is the default implementation of `resource-loader` service provided by the Asset Picker SDK. If the plugin does not implement its own `resource-loader` service, Asset Picker falls back on this default implementation. Default implementation of `resource-loader` provided by Asset Picker 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 from the `DefaultWebResourceLoaderService` class.

- `com.hcl.unica.cms.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.cms.model.request.resourceloader.ResourceRequest` and byte[] respectively. `ResourceRequest` class encapsulates the relative resource URL and instance identifier (Instance identifier is exactly same as the `systemId` used everywhere in Asset Picker). 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 `HttpWebResourceLoaderService` class in order to be recognized as a `resource-loader` service by Asset Picker. (Functional counterpart is also a valid choice to extend from.)
- `com.hcl.unica.cms.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.cms.integration.service.gateway
  .ResourceLoaderServiceGateway
  ```

  To know the role of service gateways in service invocation, please see Service invocation (on page 19). `ResourceLoaderServiceGateway` interface defines

`ResourceRequest` and `WebResource<?>` as input and output types for resource-loader service. The `com.hcl.unica.cms.model.response.resourceloader.WebResource` class is just a wrapper for HTTP response headers, body & cookies received from remote URL.

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.cms.integration.service.search.SearchService`

  The `com.example.service.functional.SimpleSearchService` class in the `asset-integration-starter` project is a quick starter implementation for the Functional `simple-search service`. Its parent is the `com.hcl.unica.cms.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 `SearchService` service in order to be recognized as a `simple-search` service by the Asset Picker (RESTful counterpart is also a valid choice to extend from).

  The `SearchService` extends from the `com.hcl.unica.cms.integration.service.search .AbstractSearchService abstract` class. It introduces one more abstract method, named `getSupportedContentTypes` to implement the `simple-search` service.

Asset selection callback (`asset-selection-callback`)

The following are the specialized interfaces and classes available for the `asset-selection-callback` service:

- `com.hcl.unica.cms.integration.service.assetselectioncallback.AssetSelectionCallbackService`

  The `com.example.service.functional.ContentSelectionCallbackService` class in the `asset-integration-starter` project is a quick starter implementation for Functional `asset-selection-callback` service. Its parent is the following class:

  ```
  com.hcl.unica.cms.integration.service.assetselectioncallback
  .AssetSelectionCallbackService
  ```

  The `AssetSelectionCallbackService` class implements the `FunctionalService` interface and defines the `AssetSelectionDetails` class and the `Object` classes to be the type arguments RQ & RS respectively for the `FunctionalService`. Thus, the object of the `AssetSelectionDetails` becomes an input to all the `asset-selection-callback` services and the `Object` or its subtype is expected as an output on completion of the service (the same input & output types are used for RESTful counterpart of asset-selection-callback). `AssetSelectionDetails` class is part of the Asset Picker SDK.

  Plugin must extend its `asset-selection-callback` implementation from the `AssetSelectionCallbackService` service in order to be recognized as an `asset-selection-callback` service by the Asset Picker (RESTful counterpart is also a valid choice to extend from).

  The `AssetSelectionCallbackService` extends from the following abstract class:

  ```
  com.hcl.unica.cms.integration.service.assetselectioncallback
  .AbstractAssetSelectionCallbackService
  ```

Resource loader (`resource-loader`)

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

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

  The `com.example.service.functional.ResourceLoaderService` class in `asset-integration-starter` project is a quick starter implementation for Functional `resource-loader` service. Its parent is the `com.hcl.unica.cms.integration.service.resourceloader.WebResourceLoaderService` class.

  The `WebResourceLoaderService` class implements the `FunctionalService` interface and defines the `ResourceRequest` and the `WebResource` classes 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 `WebResource` 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`).

  The plugin must extend its `resource-loader` implementation from the `WebResourceLoaderService` service to be recognized as a `resource-loader` service by the Asset Picker (RESTful counterpart is also a valid choice to extend from).

  The `WebResourceLoaderService` extends from the following abstract class:

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

Standard exceptions

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

RESTful approach

Asset Picker handles error conditions, arising from services implemented using RESTful approach.

Additionally, Asset Picker 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 Asset Picker 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 target content repository and are present within the com.hcl.unica.cms.integration.exception package.

- **RepositoryNotFoundException**

  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 also 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 also conveyed as 404 HTTP response to the client.

- **InaccessibleRepositoryException**

  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 also conveyed as 503 HTTP response to the client.
- **SluggishRepositoryException**

  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 also conveyed as 504 HTTP response to the client.
- **InternalRepositoryErrorException**

  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 also 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.

Loggers

Asset Picker 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>1.7.26</version>
```

```
</dependency>
```

Use 1.7.26 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.

Verification and troubleshooting

You should verify end-to-end integration after the plugin has been developed. Place the JAR file, containing the plugin implementation, in the class path of the application server where the Asset Picker is deployed. Additionally, configure the corresponding content repository in Platform configuration under supported application.

Note: Currently, only Unica Centralized Offer Management can access Asset Picker.

For more information on configuration details, see Unica Asset Picker Administration Guide.

After the plugin is deployed and the configurations are customized, restart the Asset Picker application. The changes in user data source does not require a restart.

Verification of integration

Although, you can verify Asset Picker using REST endpoints, we recommended you to check end-to-end integration by running through the relevant user interface in Unica.

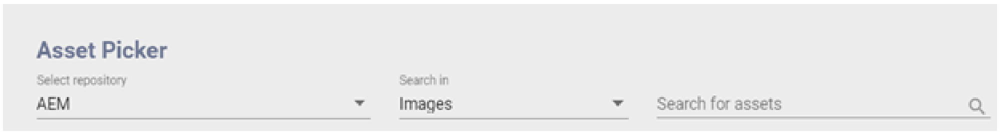
In this release, the custom attribute, for an Asset URL in Offers, allows working with Asset Picker. By navigating to the relevant screen in Offers management, you can launch Asset Picker for a URL picker custom attribute. Once the Asset Picker popup is launched, use the following guidelines to verify different services included in your plugin:

- Verify the system registration
- Verify the `simple-search` service
- Verify the `resource-loader` service
- Verify the `asset-selection-callback` service

Verify the system registration

Your content repository must be listed in the first dropdown as shown in the following image. Additionally, all the supported content types should appear in the next dropdown box.

Figure 3. Verifying system registration



Use the developer tools of your browser to troubleshoot the response received from the Asset Picker backend. The following endpoint URL is accessed when Asset Picker launches to retrieve the list of content repositories available for the logged in user:

`/api/AssetPicker/instances`

If your content repository is not listed, ensure that the system identifier used in the plugin and the system identifier used in platform configuration match. Additionally, you can refer to the application logs to check for any possible error or exception.

Verify the simple-search service

After you perform a valid search against your content repository, the expected search result with relevant details appear.

Use the developer tools on your browser to troubleshoot the response received from the Asset Picker backend. The following endpoint URL will be accessed when performing the search operation:

`/api/AssetPicker/WCM/assets?query=none&types=Images&page=0&size=10 10`

In the URL mentioned earlier:

- `WCM` is the system identifier for your content repository.
- `query` contains the search keyword.
- `types` contain the list of supported content types for filtering the search result.
- `page` is the page number of the search result.
- `size` is the maximum number of items expected on single page.

Verify the resource-loader service

In Platform configuration, if the **Anonymous Content** is configured to `No` for your content repository, the search items will be accessed through Asset Picker instead of directly connecting to the target repository. To ensure accuracy of the `resource-loader` service, verify the accessibility, or visibility, of every search item.

Use the developer tools of your browser to troubleshoot the response received from the Asset Picker backend. The following URL is accessed to retrieve individual search item:

`/api/AssetPicker/WCM/?resource /wps/wcm/connect/9350fd83-cd83-465a-a847-967f59048c0c/unnamed.jpg?MOD=AJPERES&CVID=n2B8-11`

In the earlier URL, WCM is the system identifier for your content repository and the resource contains the relative URL to corresponding search item. This URL is similar to the one the simple-search service populates in the resource URL attribute of corresponding asset, when transforming the search result.

In Platform configuration, if the **Anonymous Content** is configured to `Yes` for your content repository, all search items will be fetched directly from your content repository instead of going through the Asset Picker. In such a case, the `resource-loader` service will not be invoked.

Verify the asset-selection-callback service

The 12.0 release of Asset Picker does not support the `asset-selection-callback` service.

Overview of loggers

As mentioned in Verification of integration (on page 36), the logging configuration for Asset Picker is available in the `log4j.xml` and `log4j2.xml` files, placed in the `AssetPicker/conf/logging` folder within Platform home.

The `log4j.xml` file is used for the loggers originating from Platform’s `unica_common.jar` and `unica_helper.jar`. Whereas, `log4j2.xml` is used for the loggers originating from everywhere else in Asset Picker.

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 Asset Picker 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.

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 Asset Picker. 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 Asset Picker 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

Loggers	Information
	up under QOS section for the respective system in Platform Configuration.

Other important loggers

Other important loggers are useful in troubleshooting Asset Picker. 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 Asset Picker 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 Asset Picker'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 Asset Picker.

```
PlatformConfigurationCategoryResolver: Platform configuration: Reading
 list of entries for path {Affinium|Offer|partitions|partition1|
assetPicker|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 Asset Picker:

```
YamlConfigReader: 2 service configuration file(s) found.
YamlConfigReader: Parsing service configuration file (YAML):
 {jar:file:/{DEPLOYMEN_LOCATION}/asset-viewer/WEB-INF/lib/aem-
integration-0.0.1-SNAPSHOT.jar!/META-INF/aem-content-services.yml}...
YamlConfigReader: Parsing service configuration file (YAML):
 {jar:file:/{DEPLOYMEN_LOCATION}/asset-viewer/WEB-INF/lib/wcm-
integration-0.0.1-SNAPSHOT.jar!/META-INF/wcm-content-services.yml}...
```

- **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...
```