Version 4 Release 2

IBM i2 Analyze Connector Developer Guide

IBM

Note

Before you use this information and the product that it supports, read the information in "Notices" on page 21.

This edition applies to version 4, release 2, modification 1 of IBM® i2® Analyze and to all subsequent releases and modifications until otherwise indicated in new editions. Ensure that you are reading the appropriate document for the version of the product that you are using. To find a specific version of this document, access the Configuring section of the IBM Knowledge Center, and ensure that you select the correct version.

Contents

Connecting to external data sources

A deployment of IBM i2 Analyze that includes the Opal services can use i2 Connect to query and retrieve data from external data sources. By implementing a connector between i2 Connect and an external data source, you enable i2 Analyze to create and display records that represent the data in that source.

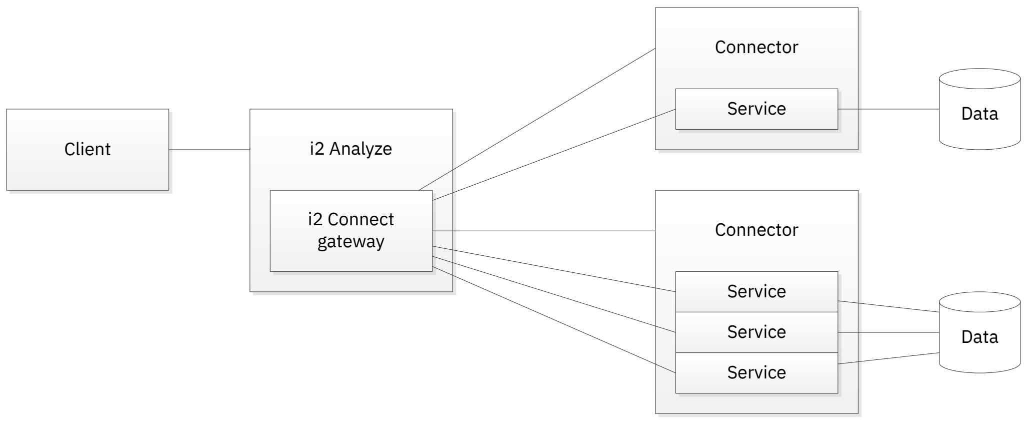
If the deployment and their permissions support it, users who create records from external sources through i2 Connect can upload them to the Information Store for storage, sharing, and subsequent analysis.

i2 Connect solutions depend on *connectors* that you create to make the bridge between i2 Analyze and external data sources. Conventionally, you write one connector for each source that you want to connect to. Each connector provides one or more *services* that present a querying interface to users and perform the actual queries on the data source.

IBM i2 Analyze and i2 Connect

At version 4.2.1 of IBM i2 Analyze, you can provide users with access to an Information Store, or enable them to query data in one or more external sources, or both. i2 Analyze deployments that are targeted at external data access include the i2 Connect gateway alongside or instead of the Information Store.

In a complete i2 Connect solution, the gateway provides i2 Analyze with information about what connectors and services are available, and how clients are to use them. i2 Analyze passes client requests for data to the gateway, which receives and responds to the requests on behalf of those services.



Each connector contains code that interacts with an external data source, and the definitions of the services that it supports, and descriptions of how clients might present services to users. For example, clients need to know the following information:

- The name and description of each service
- Whether a service supports parameters that users can specify when they run it
- The input controls to display, and the validation to perform, for each parameter

- Whether a service behaves differently as a result of the current chart selection (any selected records are then *seeds* for the operation that the service performs)

The role of the gateway is to retrieve this information from all the connectors that i2 Analyze knows about. It can then help to ensure that requests to and responses from services are formatted correctly.

A connector for i2 Connect is an implementation of a REST interface that enables the gateway to perform its role. More specifically, a connector must support the following tasks:

- Respond to requests from the gateway for information about its services
- Validate and run queries against an external data source, including any queries that require parameters or seeds
- Convert query results to a shape that the gateway can process into i2 Analyze records

Writing a connector for i2 Connect requires you to develop an implementation of the REST interface that the gateway expects, and to write queries that retrieve data from an external source. You must also understand the i2 Analyze data model, and be able to convert the retrieved data so that it matches the schema that a particular deployment of i2 Analyze supports.

The example that IBM provides in the deployment toolkit demonstrates several of these requirements in a fully functional connector. The remainder of this guide explains how to meet them in more detail.

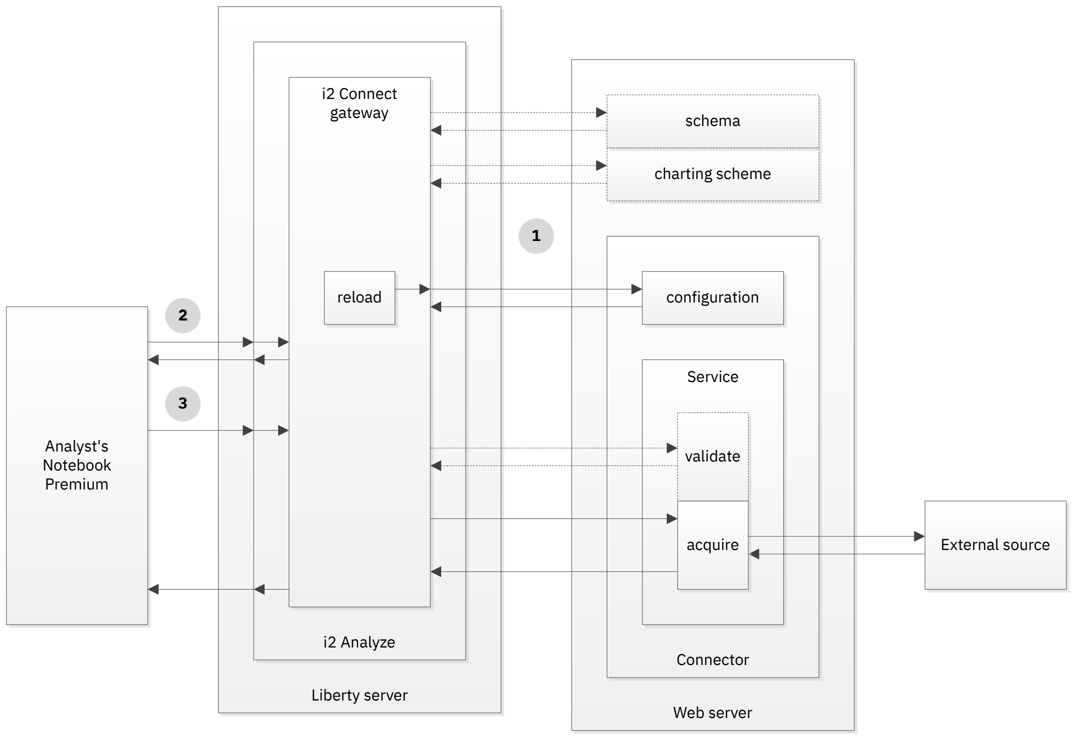
Note: For instructions on deploying i2 Connect with the example connector, see Deploying i2 Analyze with the Information Store and i2 Connect. For information on securing the communication between the gateway and connectors for i2 Connect, see Client authenticated Secure Sockets Layer with IBM i2 Connect.

Architecture of i2 Connect solutions

The simplest possible deployment of IBM i2 Analyze that uses i2 Connect to access an external data source has four parts. The client, Analyst's Notebook Premium, communicates with a deployment of i2 Analyze that includes the i2 Connect gateway. The gateway manages at least one connector, which in turn exchanges data with an external source.

i2 Connect interactions

The following diagram represents the deployment that is described in the introduction, and the interactions that take place between each part.



The diagram also identifies five REST endpoints that connectors for i2 Connect can implement and use:

Configuration endpoint

The gateway sends a request to the mandatory configuration endpoint to gather information about all the services that the connector supports. All connectors have at least one service, and it is a service that implements the validate and acquire endpoints.

Schema endpoint

The schema endpoint is optional, and so is where you implement it. If one or more of your connectors can return results whose types are not in the i2 Analyze schema, you might be able to write schema fragments that define those types. The configuration specifies whether and where a schema endpoint is available.

Charting scheme endpoint

The charting scheme endpoint is optional, and closely associated with the schema endpoint. A connector configuration that specifies a schema endpoint also specifies a charting scheme endpoint.

Validate endpoint

The validate endpoint is optional. If the configuration endpoint says that a particular service supports it, then the gateway sends a request to the validation endpoint immediately before a request to the acquire endpoint. The purpose of the validate endpoint is to test whether a request is reasonable before it is run.

Acquire endpoint

The acquire endpoint is mandatory for all services. The gateway sends a request to the acquire endpoint that tells the service to query the external source. The service receives data back from the source and places it into the response that it returns to the gateway.

Interaction sequence

The interaction between the gateway and the other parts of an i2 Analyze deployment that includes i2 Connect takes place in three distinct phases, labeled 1, 2, and 3 in the diagram.

1. When i2 Analyze starts up, the i2 Connect gateway sends a request to the configuration endpoint of every connector that is listed in the topology. It caches the information about the connectors and services that it receives in response.

   If the configuration for any of the connectors specifies schema and charting scheme endpoints, and if the deployment supports it, the gateway sends requests to retrieve fragments from those endpoints.

   Note: You can also force the gateway to repeat this process while i2 Analyze is running. The gateway implements the reload endpoint, whose POST method you can call after you change the configuration of a connector.
2. When a client connects to i2 Analyze, the latter gets the cached information from the gateway and returns it to the client. The client can then present the queries that the services implement to users, and help users to provide valid parameters and seeds to those queries where appropriate.
3. When a user runs a query, i2 Analyze passes it to the gateway for processing. The gateway packages the query into a request to the acquire endpoint on the service in question, preceded by a request to the validate endpoint if the service supports it.

   On receiving the response from the acquire endpoint, the gateway converts the data that it contains into results that i2 Analyze can ultimately return to the client.

Creating a connector for i2 Connect

To create a connector for i2 Connect, you need a data source to query, and a server that supports REST endpoints to query it from. On that foundation, you can build the functionality that takes requests from the i2 Connect gateway, retrieves data from the source, and returns it to the gateway in a shape that is ready for conversion to i2 Analyze records.

About this task

The easiest approach to writing a connector is to start by getting data from the external source to the i2 Analyze server - and then to the client - with as few complications as possible. When you have that mechanism, you can use it as a base from which to develop the more complex services that you want to implement.

Note: The example connector that IBM provides with i2 Analyze contains six simple services that between them demonstrate several aspects of connector development. As you read this documentation, it is helpful also to read the source code for that example.

Understanding the external source

Data is only valuable in i2 Analyze if users can examine it alongside, and with the same tools as, all the other data in the system. Just like ingesting or importing data, connecting to an external source requires you to align its contents with the i2 Analyze schema.

About this task

Before you can write a connector for i2 Connect, you need to understand what data is available from the external source, and how that data maps to the types in the schema. If no straightforward mapping exists between the data in the source and one or more of the entity or link types, then you might need to extend the i2 Analyze schema.

Procedure

1. Consider the type of the source and how you can query it.

   For spreadsheets and text files, it is likely that you have complete access to the data they contain. For databases and web services, you might be restricted to a set of predefined queries.
2. Identify the types of data that you can retrieve from the source, and compare them with the types in the i2 Analyze schema for your deployment.

   For each type in the source, you can reach one of three conclusions:

   - The type and its data are a match for (or a subset of) one of the entity or link types in the schema
   - The type is a match for one of the types in the schema, but its data implies new property types
   - The type is not currently represented in the schema
3. If the i2 Analyze schema already describes all the data in the external source, then you can start the process of writing queries that retrieve data from it.
4. If the data in the external source contains property types that the i2 Analyze schema does not describe, you can either ignore that data or edit the schema to add those property types.

   Additive changes to the schema are always permitted.
5. If the data in the external source contains item types that the i2 Analyze schema does not describe, then you must use one of the techniques for adding them to the schema.

Extending the i2 Analyze schema

The data that you retrieve from an external source through an i2 Connect service must correspond with entity types and link types that the i2 Analyze schema defines. If the types in the schema do not correspond with the external data, then you must extend the schema as part of connector development.

Before you begin

If you can align the data that you can retrieve from the external source with existing types in the i2 Analyze schema, then you can skip this task. If you cannot align the data, then you need to add types to the schema.

About this task

i2 Connect permits two approaches to extending an i2 Analyze schema (and its accompanying charting scheme):

- Add the new entity, link, and property types directly to the deployed schema in the usual way. Additive changes to an i2 Analyze schema are always allowed.

- Create schema and charting scheme *fragments* that you deploy alongside a connector. When you add the connector to your deployment, i2 Connect merges the fragments into the deployed schema.

  Important: Schema fragments are not compatible with i2 Analyze deployments that include an Information Store. In these deployments, you can extend the schema only by editing it directly.

In most cases, updating the deployed schema is the better approach. Any types that you create are immediately and permanently available throughout the i2 Analyze deployment. When you use IBM i2 Schema Designer to edit the schema, you also gain protection against creating duplicate identifiers.

Creating fragments might be appropriate if *both* of the following conditions are true:

- You are developing a connector for use in multiple i2 Analyze deployments. In that case, the new types travel with the connector that retrieves data for them.
- Your i2 Analyze deployments involve only i2 Connect and do not include Information Stores.

Schema fragments can add new entity and link types to the schema for a deployment. Subject to some conditions, they can also add property types to the entity and link types that are already in the schema:

- To add new entity and link types, you must ensure not only that their identifiers do not clash with existing identifiers, but also that the types themselves are distinct from existing ones. Having two different types that are both named Person is likely to confuse users!
- To add property types to an existing entity or link type, the fragment must contain a replica of the type in question. The display name, description, icon, and semantic type of the entity or link type in the fragment must match the one in the deployed schema exactly. Furthermore, any property type definition in the fragment that has the same identifier as a property type in the deployed schema must match that definition exactly.

  If you follow these rules, any new property types in the replica entity or link type are merged into the deployed schema when i2 Analyze processes the schema fragment.

Procedure

- If you decide to extend the deployed schema directly, you can do so in the same way that you would add types for any other reason:
  a) Open the XML file that contains the deployed i2 Analyze schema in Schema Designer.
  b) Add entity and link types (or property types) to the schema that correspond to data from the external source.
  c) Add configuration for new and modified entity and link types to the charting scheme.
  d) Update the i2 Analyze deployment with the modified schema and charting scheme.

  For more information, see the Schema Designer user guide, and the documentation on modifying the i2 Analyze schema.
- If you decide to create fragments, then you can still use Schema Designer, but after that the process changes:
  a) In Schema Designer, select **File** > **New Schema Fragment**.

     Schema fragments have slightly different syntax from full schemas. Choosing this option ensures that your schema fragment follows the rules.
  b) Edit the schema and charting scheme fragments to add new types for the data from the external source, and then save and close the XML files.
  c) Use a text editor to verify that none of the identifiers in the fragments duplicates an identifier in the deployed schema, unless you are adding property types to it.

Note: If you plan to use the fragments in multiple deployments, you must perform this check for each deployment.

d) Arrange for the XML in the fragments to be available from endpoints on the server that hosts the connector.

The example connector that IBM provides with i2 Analyze includes a simple demonstration.

To complete this approach, you must tell the i2 Connect gateway where to find the endpoints that you created. You provide that information when you implement the configuration endpoint.

Writing a query with no parameters

All connectors for i2 Connect contain at least one *service* that retrieves information from an external source. By starting with a service that runs a simple query, you can focus on aspects such as the REST endpoint implementations and data structures that are common to all services.

About this task

As you create a connector for i2 Connect, writing a query that does not require parameters makes early testing and development easier. Not only does it simplify the connector, but also it simplifies running the query from the client.

Procedure

No fixed set of steps for retrieving data from external data sources exists, but some of your choices can make later tasks easier. Whatever the source, your initial aim is to generate a small set of representative data.

- If the source allows it, begin by writing a tightly bound query. Or, if your source comprises one or more text files, try to abbreviate them.

  At the start of the process, large volumes of data can be distracting.
- If you can, make the query retrieve data for entities *and* links.

  Creating the data structures that represent linked entities is an important part of connector development.
- Try to generate a data set that has most of the types that the external source contains.

  No matter how complex your queries eventually become, the code to process data into the right shape for i2 Analyze is unlikely to change.

What to do next

When you implement an acquire endpoint, you can call the query that you developed here directly from that code. Alternatively, during development, you might decide to run the query now, and save the results in a file. Again, the second approach simplifies early development, and mirrors the behavior of the example connector that IBM provides.

Implementing a configuration endpoint

The configuration endpoint of a connector for i2 Connect must respond to requests with a JSON structure that describes the services in the connector. The more sophisticated the connector, the more information you need to provide.

Before you begin

Navigate to the source code for the example connector in the `connectors\example_connector` directory of your i2 Analyze distribution. In a text editor, open the `app.js` and `config.json` files that implement the configuration endpoint and demonstrate an approach that you might take.

About this task

When the i2 Analyze server starts, the i2 Connect gateway sends a request to the configuration endpoint. The response that your connector sends back tells the gateway what services are available. i2 Analyze caches this information and provides it to clients when they connect.

The simplest possible service does not support seeds and does not require users to provide parameters. As a result, it does not require the client to display user input controls. If you also decide not to create schema fragments, then the structure that you return from the configuration endpoint must contain only two objects:

- The `defaultValues` object is mandatory.

  Within the `defaultValues` object, you must specify the timezone for i2 Connect to apply to any retrieved value that does not specify its own timezone. If you want to, you can also specify the direction for any retrieved link that does not do so.

  Additionally, you can choose to specify default entity and link types that i2 Connect uses if an individual service does not specify the types of data that it can retrieve.
- The `services` array is also mandatory.

  The array must contain information for at least one service. The information must include a unique identifier and a name for the service. It must also specify whether the service requires a client UI, and the URL of the acquire endpoint.

Procedure

First, construct the `defaultValues` object:

1. Determine the timezone that is most likely to apply to any temporal data in the source that the services in this connector query.
2. Find the identifier of the timezone in the IANA Time Zone Database, and arrange for the response from the endpoint to include the identifier in its `defaultValues` object.

   For example:

   ```
   {
     "defaultValues": {
       "timeZoneId": "Europe/London",
       ...
     },
     "services": [
       ...
     ]
   }
   ```

   Note: You can also retrieve a list of supported timezones that includes their identifiers from the GET method on the i2 Analyze server's `/api/v1/core/temporal/timezones` REST endpoint.
3. If the source contains only a few types, and if you intend the connector eventually to have many services that run different queries, then consider adding an `entityTypeId` and a `linkTypeId` to the `defaultValues` object.

   It is more common to specify what types of record a query might retrieve on a per-service than on a connector-wide basis. However, if you do not supply default types here, then every service must supply types individually.

Then, add an object to the `services` array:

4. To the `services` array, add a `service` object that has an `id` and a `name`. It is common to include a `description` and populate the `resultItemTypeIds` array as well, although neither is mandatory.

   For example:

   ```
   {
     "defaultValues": {
       "timeZoneId": "Europe/London"
     },
     "services": [
       {
         "id": "exampleSearch",
         "name": "Example Search",
         "description": "An example that queries a data set of people.",
         "resultItemTypeIds": ["ET5", "LAS1"],
         ...
       }
     ]
   }
   ```

5. For a service whose query does not allow callers to provide parameters, you can set the `clientConfigType` to `NONE`.

   ```
   {
     "defaultValues": {
       "timeZoneId": "Europe/London"
     },
     "services": [
       {
         "id": "exampleSearch",
         "name": "Example Search",
         "description": "An example that queries a data set of people.",
         "resultItemTypeIds": ["ET5", "LAS1"],
         "clientConfigType": "NONE",
         ...
       }
     ]
   }
   ```

   If you later add parameters to the query, you can allow users to specify them by changing the value to `FORM` and providing the identifier of a client configuration in `clientConfigId`.

   If you later add support for seeds, you must add a `seedConstraints` object to the service.

6. Finally, set the `acquireUrl` of the service to the URL where you intend to make available the acquire endpoint.

This example comes from the supplied `config.json` file:

```
{
  "defaultValues": {
    "timeZoneId": "Europe/London"
  },
  "services": [
    {
      "id": "exampleSearch",
      "name": "Example Search",
      "description": "An example that queries a data set of people.",
      "resultItemTypeIds": ["ET5", "LAS1"],
      "clientConfigType": "NONE",
      "acquireUrl": "/exampleSearch/acquire"
    }
  ]
}
```

Results

This JSON structure represents (almost) the simplest response that you can send from the configuration endpoint of a connector for i2 Connect. It contains the definition of one simple service. In order for the i2 Connect gateway to retrieve it, you must add details of the connector to the i2 Analyze deployment topology.

Adding a connector to the topology

When you have an implementation of a configuration endpoint, you can tell i2 Analyze about the connector that it describes by adding it to the topology file. In that file, the `<connector>` element controls the behavior of the connector and the experience of its users.

Before you begin

Ensure that your deployment of i2 Analyze includes i2 Connect. If i2 Connect was not a part of the original deployment, you can add it by following the instructions in Adding i2 Connect to Opal deployments.

About this task

Each `<connector>` element in the topology file ties together the URL of a server, which might host several connectors, and the URL of the configuration endpoint for a particular connector. As usual, when you change the contents of the topology file, you must redeploy and restart the i2 Analyze server.

Procedure

The topology file that IBM provides in deployments that support i2 Connect includes the `<connectors>` element. Adding a connector to the topology means adding a child to that element.

1. Navigate to the `configuration\environment` directory of the toolkit for your deployment of i2 Analyze. In a text editor, open the `topology.xml` file.

2. Add a `<connector>` element to the `<connectors>` element, according to the following syntax:

   ```
   <connectors>
     <connector id="ConnectorId"
                name="ConnectorName"
                base-url="Protocol://HostName:PortNumber"
                configuration-url="Protocol://HostName:PortNumber/Path"/>
   </connectors>
   ```

   Here, *ConnectorId* must be unique within the topology file, while *ConnectorName* is likely to be displayed to users in the list of queries that they can run against external sources.

   The i2 Connect gateway uses the value that you assign to the `base-url` attribute as the stem for the URLs that you specify in the configuration (such as `/exampleSearch/acquire`). `configuration-url` is the only optional attribute.

   By default, the gateway attempts to retrieve the configuration from `<base-url>/config`. You can change this behavior by specifying a different URL as the value of `configuration-url`.
3. Save the file, and then redeploy and restart the i2 Analyze server.
   a) On the command line, navigate to `toolkit\scripts`.
   b) Run the following commands in sequence:

   ```
   setup -t deployLiberty
   setup -t startLiberty
   ```

   When the server restarts, the i2 Connect gateway makes its request to the configuration endpoint. If that endpoint (or any of the endpoints in the retrieved configuration) is not available, the result is not an unrecoverable error. Rather, the i2 Analyze server logs the problem, and users see messages about unavailable services in the client application.

Implementing an acquire endpoint

In a connector for i2 Connect, an acquire endpoint must perform two tasks in succession. Its roles are to retrieve data from an external source, and to return that data to the i2 Connect gateway in a form that the latter can use to create i2 Analyze records.

Before you begin

Refer again to the `app.js` file for the example connector in your i2 Analyze distribution. It contains implementations of the acquire endpoint for several services, which cover many of the common considerations.

About this task

When a user runs a query against an external data source through their client application, the i2 Connect gateway makes a request to the acquire endpoint of the relevant service. The request always has a *payload* that can contain information (often, parameter values or seed records) that the user provided to the query.

In general, your implementation of an acquire endpoint uses the contents of the payload to refine the commands that it sends to the external source. If the service that owns the endpoint specified a `clientConfigType` of `FORM`, then the identifiers of conditions in the form match the identifiers of conditions in the payload. If the service specified `seedConstraints`, then the payload contains the property values of seed records.

For a query with no parameters and no seeds, the payload is empty. You can focus your development effort on making sure that the response from the acquire endpoint meets the requirements of i2 Connect.

Procedure

Note: It is not possible to follow the same procedure for every different kind of external data source. The following steps do not apply in all cases. Rather, they describe tasks that you are likely to require.

1. In IBM i2 Analyze Schema Designer, open the schema file that contains the definitions of the i2 Analyze entity, link, and property types that data from the external source must match.
2. Make a note of the identifiers and logical types of the property types of each of the entity and link types in the retrieved data. Make sure that you know which property types are mandatory.
3. Run the command against the external data source and store the retrieved data in a structure that you can manipulate easily.
4. If the data from the external source contains information about links between entities, but the links in the data have no natural identifiers, you must manufacture those identifiers.

   This problem is similar to the one you face when you prepare data for ingestion into the Information Store.
5. Create the arrays of entity and link data objects that must appear in the response from the acquire endpoint.
6. Populate the arrays according to the descriptions in the SPI documentation, taking care to ensure that you provide values for all mandatory properties.

Modifying and testing a connector

A connector for i2 Connect is defined by the information that the i2 Connect gateway retrieves from its configuration endpoint. To add a connector, or when you modify an existing one, you must arrange for the gateway to reload the information for all the connectors in the topology.

About this task

A connector that has only a configuration endpoint and an acquire endpoint is simple, but entirely functional. When you have a connector in this state, you can verify that many of the most important mechanisms are functioning correctly. To enable a faster test and development cycle, add a connector to a running instance of i2 Analyze as soon as you can.

The i2 Connect gateway maintains a cache of configuration information that it provides to clients on request. Depending on the stage of development of your connector, you can use different approaches to make the gateway update its cache:

- If you add or edit a `<connector>` element in the topology file, then you must redeploy and restart the i2 Analyze server, as described in “Adding a connector to the topology” on page 10.
- If you modify a connector so that the response from its configuration endpoint changes - when you add a validate endpoint, for example - you do not need to redeploy the server. Instead, you can either run the `restartLiberty` task from the deployment toolkit, or use the i2 Connect gateway's reload endpoint.

Regardless of how you update the gateway's cache of connector configuration information, the effect for clients and their users is the same. When a client next asks for a list of the i2 Connect services that it can use, any changes that you made are reflected in the list.

Procedure

To run `restartLiberty`, you can use the same technique as for any other deployment toolkit task. This section describes how to use the reload endpoint, and then explains some ways to diagnose problems with a new or updated connector.

The reload endpoint supports the POST method, which you must call through a command-line tool such as `postman` or `curl`. The endpoint requires authentication, which means that you must log in to the server and retrieve a cookie before you can POST to reload. The following steps demonstrate how to use `curl` to do so:

1. If the `curl` utility is not available on your server, download it from the project website at https://curl.haxx.se/download.html.
2. Open a command window and use `curl` to log in to the i2 Analyze server and retrieve an authorization cookie:

   ```
   curl -i --cookie-jar cookie.txt
        -d j_username=user_name
        -d j_password=password
        http://host_name:host_port/context_root/j_security_check
   ```

   This command connects to the specified i2 Analyze server as the specified user, retrieves the authentication cookie, and saves it to a local file named `cookie.txt`.
3. Use `curl` a second time to call the POST method on the reload endpoint:

   ```
   curl -i --cookie cookie.txt
        -X POST http://host_name:host_port/context_root/api/v1/connectors/reload
   ```

   This command causes the i2 Connect gateway to reload configuration information from all the connectors that the topology included when it was last deployed. Clients receive updated information about the connectors when they next connect to i2 Analyze.
4. After you reload the configuration or restart the server, check that the client reflects your changes to the connectors and their services in its list of queries.

   If it does not, different symptoms imply different causes:

   - If a query does not appear in the list at all, then the problem lies with either the implementation of the configuration endpoint, or the `<connector>` element in the `topology.xml` file.
   - If a query is in the list but marked as unavailable, then the cause of the problem is either the implementation of the acquire endpoint, or the specification of that endpoint in the response from the configuration endpoint.
   - If the displayed information about a query is faulty or incomplete, look again at the definition of the corresponding service in the response from the configuration endpoint.
5. Provided that the queries due to new or modified connectors appear correctly in the client, you can run them and view the results that they return.

Extending an i2 Connect solution

The simplest possible connector for i2 Connect requires only a configuration endpoint and an acquire endpoint in order for clients to present its services to their users. By extending these endpoints and adding new ones, you can improve the reliability and utility of the connectors that you write.

Supporting schema fragments

Sometimes, an external data source contains information about entities and links that do not translate to types in the deployed i2 Analyze schema. In that situation, a connector can provide definitions of those types (and how to display them) to i2 Analyze through the i2 Connect gateway.

Before you begin

These instructions assume that you already created schema and charting scheme fragments that define entity and link types for the data that your connector retrieves, as described in "Extending the i2 Analyze schema" on page 5. You might find it helpful to open the `app.js` and `config.json` example files, which contain simple demonstrations of following the procedure here.

About this task

i2 Analyze supports two mechanisms for extending an i2 Analyze schema to incorporate new types from external data sources. You can add types to the main schema for the deployment, or you can create schema fragments that are combined with the deployed schema when the server starts up. To enable the second mechanism, you must add endpoints for retrieving fragments to any connector that uses them.

Important: i2 Analyze supports schema fragments only in deployments that do not include an Information Store. For deployments that do include an Information Store, the only way to add new types is to modify the main schema.

Procedure

To add schema and charting scheme endpoints to a connector, you must specify the locations of those endpoints in its configuration. The `schemaUrl` and `chartingSchemesUrl` settings apply to the whole connector, and appear at the same level as the mandatory `defaultValues` and `services` settings.

1. Make your schema and charting scheme fragments available through two simple HTTP GET methods that do not require parameters.

   It is easiest, but not mandatory, to implement these endpoints on the server that hosts the connector that uses them.

   The `app.js` example file contains implementations that place the fragment files directly into the response.

2. Add `schemaUrl` and `chartingSchemesUrl` settings to the response from your implementation of the configuration endpoint:

   ```
   {
     "defaultValues": {
       ...
     },

     "schemaUrl": "/schema",
     "chartingSchemesUrl": "/chartingSchemes",

     "services": [
       {
         ...
       }
     ]
   }
   ```

   In this form, i2 Analyze appends the URLs that you provide to the `base-url` that you specified for the connector in the topology file. By definition, these endpoints are on the same server as the connector. To use endpoints that are implemented elsewhere, you must provide the full URLs here.

Adding endpoints like these changes the connector but not the topology, so you do not need to redeploy i2 Analyze. Restarting the server or using the reload endpoint on the i2 Connect gateway is enough.

3. Follow the steps in "Modifying and testing a connector" on page 12 to restart the server or force the gateway to reload your connector configuration.

   If there are no conflicts between the types in your fragments and the types in any other fragments (or the main schema), the new types become available for use.

Supporting seeded queries

If you configure them to be so, the services in connectors for i2 Connect can be run with *seed records* that enable or change their functionality. Your implementations of the acquire endpoint receive identity and property data from selected records that you can use to drive operations against the external data source.

Before you begin

As you read the following information, also look at the example services named "Example Seeded Search" in `config.json`, and the implementations of their acquire endpoints in `app.js`.

About this task

The behavior of a query that supports parameters is modified by values that users specify when they run it. The behavior of a query that supports seeds is modified by the records that a user selects. Seeded queries generally fall into one of a handful of categories:

- A "find like this" query looks at the property values of a selected record and searches for data in the external source that has the same or similar property values.
- A "get latest" query uses the identifying information of a selected record to look for the same data in the external source, with the aim of updating the record.

- An "expand" query uses the identifying information of a selected record to search for data in the external source that is connected to that record.
- A "find path" query receives the identifying information for exactly two selected records and searches for a path that connects those records in the external source.

To indicate that a particular service supports seeds, you add a `seedConstraints` object to its definition in your response from the configuration endpoint. The REST SPI documentation for the configuration endpoint describes the structure of this object:

```
"seedConstraints": {
  "connectorIds": [""],
  "min": 0,
  "max": 0,
  "seedTypes": {
    "allowedTypes": "",
    "itemTypes": [
      {
        "id": "",
        "min": 0,
        "max": 0
      }
    ]
  }
}
```

A service can specify which records are acceptable as seeds by restricting the connectors they came from. It can also set boundaries on the number of seed records it accepts, regardless of their type. A service must specify whether it accepts entity or link records as seeds, and then (optionally) restrict the range to a subset of those types. It can also set boundaries on the number of seeds on a per-type basis.

If you configure the constraints so that requests must contain at least one of each permitted seed type, then the service *requires* seeds, and users cannot run it without an appropriate selection. Otherwise, your implementation of the acquire endpoint must support requests that might or might not contain seed information.

When the i2 Connect gateway calls your acquire endpoint to perform a seeded query, it includes a payload that contains identifiers and property values for all seeds. For information about the structure of the payload, see the REST SPI documentation for the `request` parameter of the acquire endpoint.

Procedure

In outline, the procedure for supporting seeded queries in one of your i2 Connect services is to start by adding seed constraints to its configuration. Clients then interpret the configuration and present the queries to users. When users run a query, your implementation of the acquire endpoint receives seeds that you can use to guide your response.

1. Decide what kind of seeded operation you want to perform, and its implications for the service.

   For an "expand" query, for example, you might accept a fairly large number of entities of any type as seeds. For "find like this", the seed is more likely to be a single record of a specific type.
2. Add a `seedConstraints` object that reflects the requirements of the query, to the response from the configuration endpoint.

In the supplied `config.json` file, the example services with identifiers `exampleSeededSearch1` and `exampleSeededSearch2` demonstrate "find like this" and "expand" queries, in that order.

3. Implement the acquire endpoint for your service.

   If at least one of the constraints does not specify a minimum record count, the endpoint might be called with or without seeds.

   The REST SPI documentation for the acquire endpoint describes the structure of the seed data, while the supplied `app.js` file contains several implementation examples.

   Important: For "get latest" and "expand" queries, it is common for seed records also to appear in the results. In that case, you must ensure that the `id` of the outgoing record matches the `seedId` of the incoming record. In an "expand" query, for example, you can identify a seed as being one end of a link in the response by setting the `fromEndId` of the link to the `seedId`.

4. Restart the i2 Analyze server or instruct the i2 Connect gateway to reload the configuration. Then, connect to the server with a client and ensure that the service is working properly.

   If your service supports being used with and without seeds, then its query appears in the client's lists of seeded and "unseeded" queries. Selecting the query in either list displays information about what seeds the user can or must select before they can use it.

Supporting parameters

In nearly all real situations, users want to be able to customize the queries that they can run against an external data source. To make that possible, you must configure the service to present parameters to its users. In your implementation of the acquire endpoint, you can act on the values that they specify.

Before you begin

As you read the following information, also look at the example services named "Example Search" and "Example Seeded Search 1" in the supplied `config.json` file, and the implementations of their acquire endpoints in `app.js`.

About this task

When users create visual queries for the Information Store, they can add conditions that place constraints on the records that a query returns. To support parameters in an i2 Connect service, you add a fixed set of conditions for which users can supply values. In a query for people, you might allow users to search for particular names or characteristics. In a "find path" seeded query, you might allow users to specify how long the path can be.

Note: A significant difference between visual query and i2 Connect is that in the latter, conditions are not bound to property types. An i2 Connect condition can be anything that you can process usefully in your implementation.

To add parameters to a service, you add a *client configuration* to its definition that describes how each condition is presented to users. The REST SPI for the configuration endpoint describes the structure of `clientConfig` objects, which look like this outline:

```
{
  "id": "",
  "config": {
    "sections": [
      {
        "title": "",
        "conditions": [
          {
            "id": "",
            "label": "",
            "description": "",
            "mandatory": false,
            "logicalType": ""
          }
        ]
      }
    ]
  }
}
```

The client configuration is responsible for the appearance of the conditions that users see, and the restrictions on the values they can provide. As well as controlling the types of conditions, and specifying whether supplying a value is mandatory, you can add further validation that is appropriate to their type. For more information about this kind of validation, see the REST SPI documentation for the configuration endpoint.

When a user opens a query that supports parameters, they see a form that displays your conditions. When they provide values and run the query, your implementation of the acquire endpoint receives a payload that contains those values. You can write the implementation to act on those values in any way that makes sense for your data source.

Procedure

Like many other aspects of developing a connector for i2 Connect, supporting parameters means changing your implementations of the configuration and acquire endpoints.

1. In your response from the configuration endpoint, write or modify the service definition so that its `clientConfigType` is "FORM", and add a `clientConfigId`.

   This identifier links to a client configuration elsewhere in the response. In some circumstances, it might be appropriate to use the same client configuration for more than one service.
2. Add a `clientConfigs` array to the response, and within it a client configuration object whose `id` matches the identifier that you specified in Step 1.

   The "`exampleForm`" in `config.json` demonstrates this relationship.
3. Add your conditions to the client configuration.

   To begin with, you might consider leaving out validation checks in the interests of getting a working implementation quickly.
4. Add code to your implementation of the acquire endpoint that unpacks the condition values from the payload.

You can then use those values to affect the query that you perform against the external source.

The supplied `app.js` contains examples of this approach.

5. Restart the i2 Analyze server or instruct the i2 Connect gateway to reload the configuration.

   You can now test the code and make sure that it does what you expect.
6. Return to the response from the configuration endpoint, edit the condition descriptions, and add client-side validation to improve the user experience.
7. Restart or reload again, and ensure that the validation has your intended effect.

What to do next

The validation that you can specify in the response from the configuration endpoint is performed by the client, and applies to values in isolation. If your service (and your users) might benefit from some more complex validation, consider adding a validate endpoint.

Supporting validation

The support in i2 Connect for queries that take parameters includes the ability to perform relatively simple, client-side validation on the values that users supply. In cases where running a query might fail due to a set of values that are mutually incompatible, you can write an endpoint to perform server-side validation.

About this task

When you write a service definition that includes conditions, you have the option (and sometimes the duty) to include logic that validates the values that users supply for those conditions. For example, you might insist that a string is shorter than a certain length, or that its contents match a particular pattern. If you ask the user to select from a discrete set of values, then you must provide the values for them to select from.

However, there are other kinds of validation that the mechanism for defining conditions does not support. In particular, you cannot use it to validate values that are reasonable in isolation, but faulty in combination. (For example, dates of birth and death might both contain reasonable values, but it would make no sense to search for individuals where the latter is earlier than the former.)

If you have an implementation of the acquire endpoint for which it might be useful to perform this kind of validation, you can write a validate endpoint. When your configuration specifies a validate endpoint for a service, the gateway uses it before the acquire endpoint, and passes the same payload. If you decide that validation fails, the request to the acquire endpoint does not happen, and the user sees an error message that you specify.

Procedure

To add a validate endpoint to an i2 Connect service:

1. In your response from the configuration endpoint, add the `validateUrl` setting alongside the existing `acquireUrl` setting. Set its value to the location of the implementation.
2. Implement the rules for the validate endpoint in a way that is consistent with its definition in the REST SPI documentation.

   The payload that the endpoint receives in the request contains all the same seed and parameter information as the acquire endpoint receives.

3. If validation succeeds according to your rules, return an empty response. If it fails, set the response to a simple object that contains an `errorMessage`:

   ```
   {
     "errorMessage": ""
   }
   ```

   When the i2 Connect gateway receives a response that is not empty, it does not then send a request to the acquire endpoint.
4. Restart the i2 Analyze server or instruct the i2 Connect gateway to reload the configuration. Test that your new validation has the correct behavior.

Notices

Trademarks