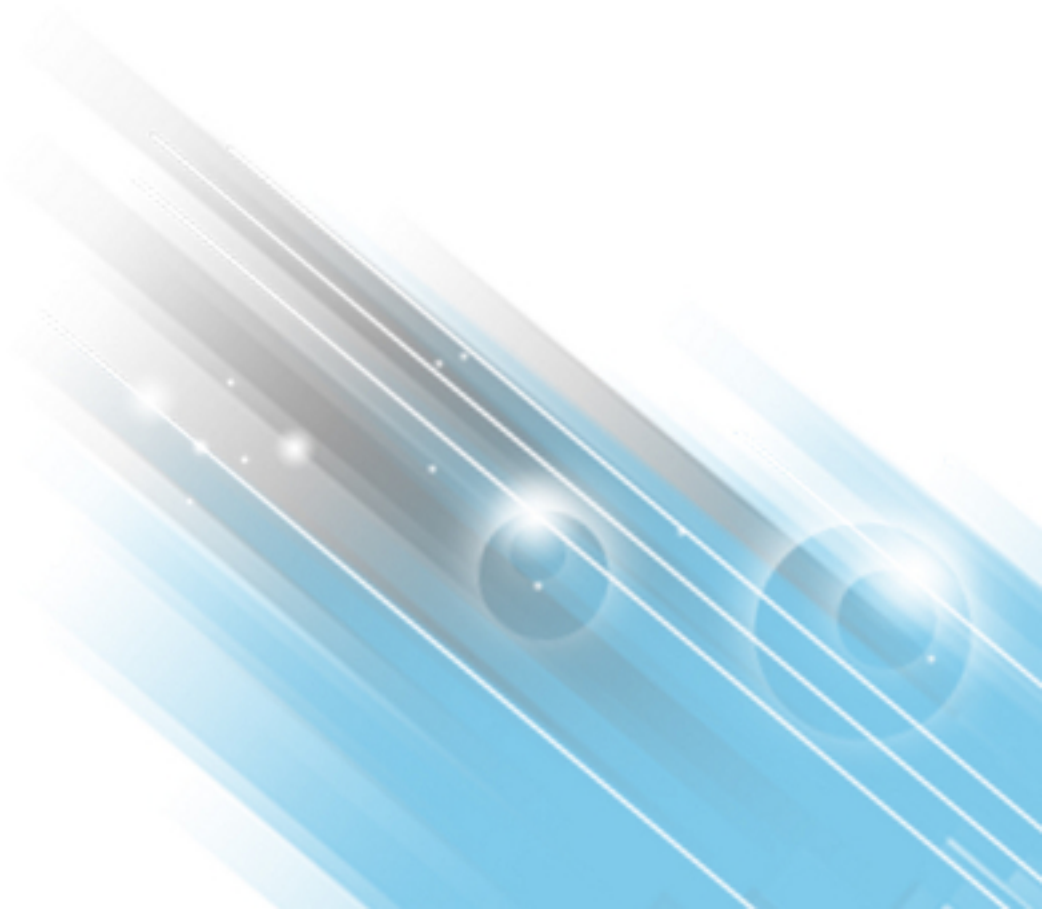
HP Universal CMDB

Software Version: 10.20

Developer Reference Guide



Document Release Date: January 2015
Software Release Date: January 2015

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

© 2002 - 2015 Hewlett-Packard Development Company, L.P.

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to: https://softwaresupport.hp.com/.

This site requires that you register for an HP Passport and to sign in. To register for an HP Passport ID, click **Register** on the HP Support site or click **Create an Account** on the HP Passport login page.

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

Support

Visit the HP Software Support site at: https://softwaresupport.hp.com.

This website provides contact information and details about the products, services, and support that HP Software offers.

HP Software online support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support website to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and to sign in. Many also require a support contract. To register for an HP Passport ID, click **Register** on the HP Support site or click **Create an Account** on the HP Passport login page.

To find more information about access levels, go to: https://softwaresupport.hp.com/web/softwaresupport/access-levels.

HP Software Solutions Now accesses the HPSW Solution and Integration Portal website. This site enables you to explore HP Product Solutions to meet your business needs, includes a full list of Integrations between HP Products, as well as a listing of ITIL Processes. The URL for this website is http://h20230.www2.hp.com/sc/solutions/index.jsp.

About this PDF Version of Online Help

This document is a PDF version of the online help. This PDF file is provided so you can easily print multiple topics from the help information or read the online help in PDF format. Because this content was originally created to be viewed as online help in a web browser, some topics may not be formatted properly. Some interactive topics may not be present in this PDF version. Those topics can be successfully printed from within the online help.

Contents

Part I: Creating Discovery and Integration Adapters

Chapter 1: Adapter Development and Writing

This chapter includes:

Adapter Development and Writing Overview

Prior to beginning actual planning for development of new adapters, it is important for you to understand the processes and interactions commonly associated with this development.

The following sections can help you understand what you need to know and do to successfully manage and execute a discovery development project.

This chapter:

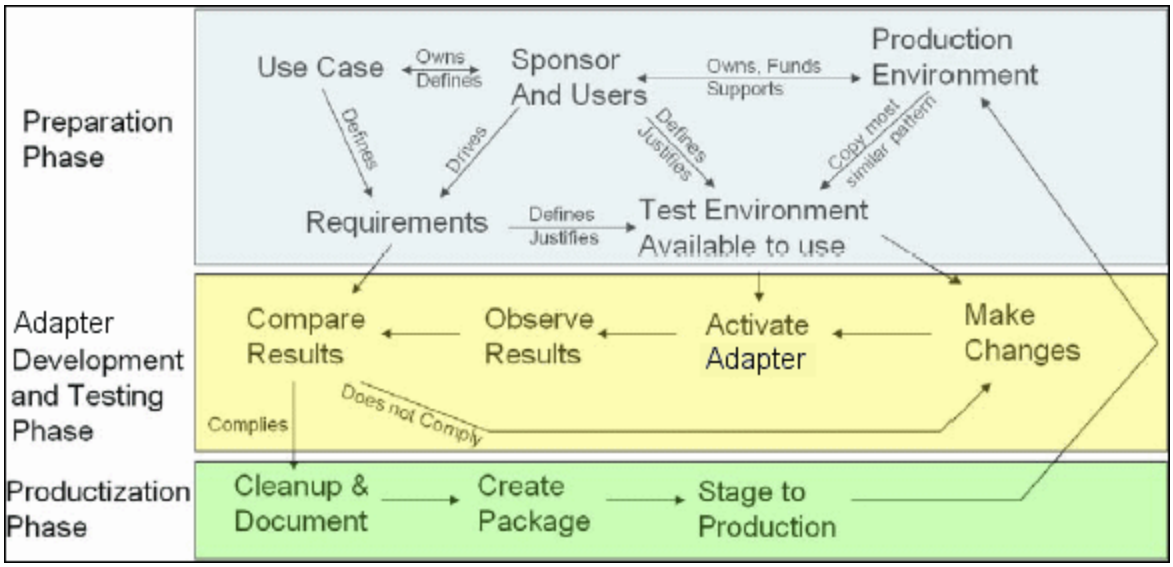
- Assumes a working knowledge of HP Universal CMDB and some basic familiarity with the elements of the system. It is meant to assist you in the learning process and does not provide a complete guide.
- Covers the stages of planning, research, and implementation of new discovery content for HP Universal CMDB, together with guidelines and considerations that need to be taken into account.
- Provides information on the key APIs of the Data Flow Management Framework. For full documentation on the available APIs, see the *HP Universal CMDB Data Flow Management API Reference*. (Other non-formal APIs exist but even though they are used on out-of-the-box adapters, they may be subject to change.)

Content Creation

This section includes:

The Adapter Development Cycle

The following illustration shows a flowchart for adapter writing. Most of the time is spent in the middle section, which is the iterative loop of development and testing.

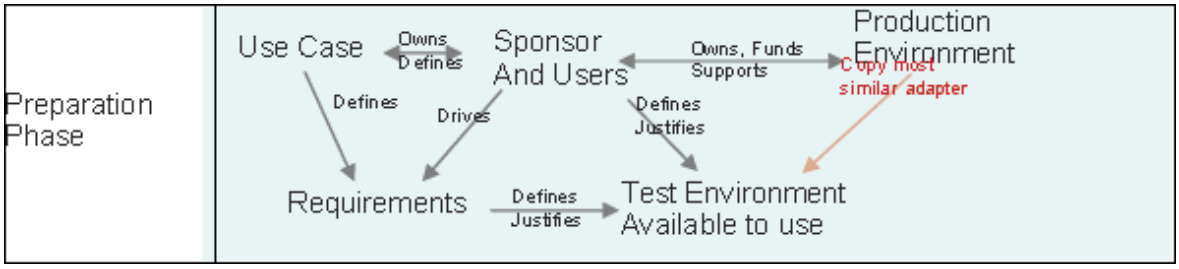


Each phase of adapter development builds on the last one.

Once you are satisfied with the way the adapter looks and works, you are ready to package it. Using either the UCMDB Package Manager or manual exporting of the components, create a package *.zip file. As a best practice, you should deploy and test this package on another UCMDB system before releasing it to production, to ensure that all the components are accounted for and successfully packaged. For details on packaging, see Package Manager in the *HP Universal CMDB Administration Guide*.

The following sections expand on each of the phases, showing the most critical steps and best practices:

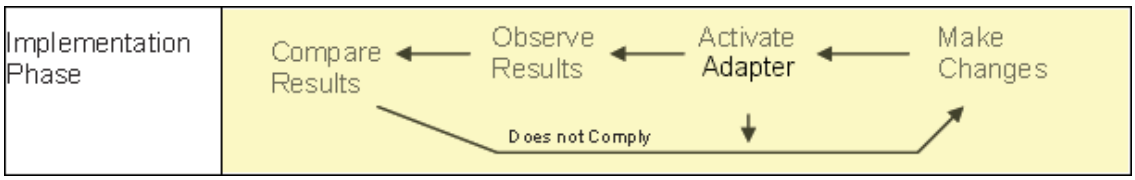
Research and Preparation Phase



The Research and Preparation phase encompasses the driving business needs and use cases, and also accounts for securing the necessary facilities to develop and test the adapter.

1. When planning to modify an existing adapter, the first technical step is to make a backup of that adapter and ensure you can return it to its pristine state. If you plan to create a new adapter, copy the most similar adapter and save it under an appropriate name. For details, see Resources Pane in the *HP Universal CMDB Data Flow Management Guide*.
2. Research the method which the adapter should use to collect data:
   - Use external tools/protocols to obtain the data
   - Develop how the adapter should create CIs based on the data
   - You now know what a similar adapter should look like
3. Determine most similar adapter based on:
   - Same CIs created
   - Same Protocols used (SNMP)
   - Same kind of targets (by OS type, versions, and so on)
4. Copy the entire package.
5. Unzip the package contents into the work space and rename the adapter (XML) and Jython (.py)

files.



Adapter Development and Testing

The Adapter Development and Testing phase is a highly iterative process. As the adapter begins to take shape, you begin testing against the final use cases, make changes, test again, and repeat this process until the adapter complies with the requirements.

Startup and Preparation of Copy

- Modify XML parts of the adapter: Name (id) in line 1, Created CI Types, and Called Jython script name.
- Get the copy running with identical results to the original adapter.
- Comment out most of the code, especially the critical result-producing code.

Development and Testing

- Use other sample code to develop changes
- Test adapter by running it
- Use a dedicated view to validate complex results, search to validate simple results

Adapter Packaging and Productization

The **Adapter Packaging and Productization** phase accounts for the last phase of development. As a best practice, a final pass should be made to clean up debugging remnants, documents, and comments, to look at security considerations, and so on, before moving on to packaging. You should always have at least a readme document to explain the inner workings of the adapter. Someone (maybe even you) may need to look at this adapter in the future and will be aided greatly by even the most limited documentation.

Cleanup and Document

- Remove debugging
- Comment all functions and add some opening comments in the main section
- Create sample TQL and view for the user to test

Create Package

- Export adapters, TQL, and so on with the Package Manager. For details, see Package Manager in the *HP Universal CMDB Administration Guide*.
- Check any dependencies your package has on other packages, for example, if the CIs created by those packages are input CIs to your adapter.
- Use Package Manager to create a package zip. For details, see Package Manager in the *HP Universal CMDB Administration Guide*.
- Test deployment by removing parts of the new content and redeploying, or deploying on another test system.

Data Flow Management and Integration

DFM adapters are capable of integration with other products. Consider the following definitions:

- DFM collects specific content from many targets.
- Integration collects multiple types of content from one system.

Note that these definitions do not distinguish between the methods of collection. Neither does DFM. The process of developing a new adapter is the same process for developing new integration. You do the same research, make the same choices for new vs. existing adapters, write the adapters the same way, and so on. Only a few things change:

- The final adapter's scheduling. Integration adapters may run more frequently than discovery, but it depends on the use cases.
- Input CIs:
  - Integration: non-CI trigger to run with no input: a file name or source is passed through the adapter parameter.
  - Discovery: uses regular, CMDB CIs for input.

For integration projects, you should almost always reuse an existing adapter. The direction of the integration (from HP Universal CMDB to another product, or from another product to HP Universal CMDB) may affect your approach to development. There are field packages available for you to copy for your own uses, using proven techniques.

From HP Universal CMDB to another project:

- Create a TQL that produces the CIs and relations to be exported.
- Use a generic wrapper adapter to execute the TQL and write the results to an XML file for the external product to read.

> **Note:** For examples of field packages, contact HP Software Support.

To integrate another product to HP Universal CMDB, depending on how the other product exposes its data, the integration adapter acts differently:

Integration Type	Reference Example to Be Reused
Access the product's database directly	HP ED
Read in a csv or xml file produced by an export	HP ServiceCenter
Access a product's API	BMC Atrium/Remedy

Associating Business Value with Discovery Development

The use case for developing new discovery content should be driven by a business case and plan to produce business value. That is, the goal of mapping system components to CIs and adding them to the CMDB is to provide business value.

The content may not always be used for application mapping, although this is a common intermediate step for many use cases. Regardless of the end use of the content, your plan should answer the following questions:

- Who is the consumer? How should the consumer act on the information provided by the CIs (and the relationships between them)? What is the business context in which the CIs and relationships are to be viewed? Is the consumer of these CIs a person or a product or both?
- Once the perfect combination of CIs and relationships exists in the CMDB, how do I plan on using them to produce business value?

- What should the perfect mapping look like?
  - What term would most meaningfully describe the relationships between each CI?
  - What types of CIs would be most important to include?
  - What is the end usage and end user of the map?
- What would be the perfect report layout?

Once the business justification is established, the next step is to embody the business value in a document. This means picturing the perfect map using a drawing tool and understanding the impact and dependencies between CIs, reports, how changes are tracked, what change is important, monitoring, compliance, and additional business value as required by the use cases.

This drawing (or model) is referred to as the **blueprint**.

For example, if it is critical for the application to know when a certain configuration file has changed, the file should be mapped and linked to the appropriate CI (to which it relates) in the drawn map.

Work with an SME (Subject Matter Expert) of the area, who is the end user of the developed content. This expert should point out the critical entities (CIs with attributes and relationships) that must exist in the CMDB to provide business value.

One method could be to provide a questionnaire to the application owner (also the SME in this case). The owner should be able to specify the above goals and blueprint. The owner must at least provide a current architecture of the application.

You should map critical data only and no unnecessary data: you can always enhance the adapter later. The goal should be to set up a limited discovery that works and provides value. Mapping large quantities of data gives more impressive maps but can be confusing and time consuming to develop.

Once the model and business value is clear, continue to the next stage. This stage can be revisited as more concrete information is provided from the next stages.

Researching Integration Requirements

The prerequisite of this stage is a **blueprint** of the CIs and relationships needed to be discovered by DFM, which should include the attributes that are to be discovered. For details, see "Adapter Development and Writing Overview" on page 13.

This section includes the following topics:

Modifying an Existing Adapter

You modify an existing adapter when an out-of-the-box or field adapter exists, but:

- It does not discover specific attributes that are needed
- A specific type of target (OS) is not being discovered or is being incorrectly discovered
- A specific relationship is not being discovered or created

If an existing adapter does some, but not all, of the job, your first approach should be to evaluate the existing adapters and verify if one of them almost does what is needed; if it does, you can modify the existing adapter.

You should also evaluate if an existing field adapter is available. Field adapters are discovery adapters that are available but are not out-of-the-box. Contact HP Software Support to receive the current list of field adapters.

Writing a New Adapter

A new adapter needs to be developed:

- When it is faster to write an adapter than to insert the information manually into the CMDB (generally, from about 50 to 100 CIs and relationships) or it is not a one-time effort.
- When the need justifies the effort.
- If out-of-the-box or field adapters are not available.
- If the results can be reused.
- When the target environment or its data is available (you cannot discover what you cannot see).

Model Research

- Browse the UCMDB class model (CI Type Manager) and match the entities and relations from your **blueprint** to existing CITs. It is highly recommended to adhere to the current model to avoid possible complications during version upgrade. If you need to extend the model, you should create new CITs since an upgrade may overwrite out-of-the-box CITs.
- If some entities, relations, or attributes are lacking from the current model, you should create them. It is preferable to create a package with these CITs (which will also later hold all the discovery, views, and other artifacts relating to this package) since you need to be able to deploy these CITs on each installation of HP Universal CMDB.

Technology Research

Once you have verified that the CMDB holds the relevant CIs, the next stage is to decide how to retrieve this data from the relevant systems.

Retrieving data usually involves using a protocol to access a management part of the application, actual data of the application, or configuration files or databases that are related to the application. Any data source that can provide information on a system is valuable. Technology research requires both extensive knowledge of the system in question and sometimes creativity.

For home-grown applications, it may be helpful to provide a questionnaire form to the application owner. In this form the owner should list all the areas in the application that can provide information needed for the blueprint and business values. This information should include (but does not have to be limited to) management databases, configuration files, log files, management interfaces, administration programs, Web services, messages or events sent, and so on.

For off-the-shelf products, you should focus on documentation, forums, or support of the product. Look for administration guides, plug-ins and integrations guides, management guides, and so on. If data is still missing from the management interfaces, read about the configuration files of the application, registry entries, log files, NT event logs, and any artifacts of the application that control its correct operation.

Guidelines for Choosing Ways to Access Data

Relevance: Select sources or a combination of sources that provide the most data. If a single source supplies most information whereas the rest of the information is scattered or hard to access, try to assess the value of the remaining information by comparison with the effort or risk of getting it. Sometimes you may decide to reduce the blueprint if the value or cost does not warrant the invested effort.

Reuse: If HP Universal CMDB already includes a specific connection protocol support it is a good reason to use it. It means the DFM Framework is able to supply a ready made client and configuration for the connection. Otherwise, you may need to invest in infrastructure development. You can view the currently supported HP Universal CMDB connection protocols in the **Data Flow Management > Data Flow Probe Setup > Domains and Probes pane**. For details about each protocol, see the section describing the supported protocols in the *HP UCMDB Discovery and Integrations Content Guide*.

You can add new protocols by adding new CIs to the model. For details, contact HP Software Support.

Note: To access Windows Registry data, you can use either WMI or NTCMD.

Security: Access to information usually requires credentials (user name, password), which are entered in the CMDB and are kept secure throughout the product. If possible, and if adding security does not conflict with other principles you have set, choose the least sensitive credential or protocol that still answers access needs. For example, if information is available both through JMX (standard administration interface, limited) and Telnet, it is preferable to use JMX since it inherently provides limited access and (usually) no access to the underlying platform.

Comfort: Some management interfaces may include more advanced features. For example, it might be easier to issue queries (SQL, WMI) than to navigate information trees or build regular expressions for parsing.

Developer Audience: The people who will eventually develop adapters may have an inclination towards a certain technology. This can also be considered if two technologies provide almost the same information at an equal cost in other factors.

Summary

The outcome of this stage is a document describing the access methods and the relevant information that can be extracted from each method. The document should also contain a mapping from each source to each relevant blueprint data.

Each access method should be marked according to the above instructions. Finally you should now have a plan of which sources to discover and what information to extract from each source into the blueprint model (which should by now have been mapped to the corresponding UCMDB model).

Developing Integration Content

Before creating a new integration, you must understand what the integration's requirements are:

- Should the integration copy data into the CMDB? Should the data be tracked by history? Is the source unreliable?

  If you answer yes to these questions, then **Population** is needed.

- Should the integration federate data on the fly for views and TQL queries? Is the accuracy of changes to data critical? Is the amount of data too large to copy to the CMDB, but the requested amount of data is usually small?

  If you answer yes to these questions, then **Federation** is needed.

- Should the integration push data to remote data sources?

  If you answer yes to these questions, then **Data Push** is needed.

- Is the length of any CI's ID greater than 60 characters?

  If you answer yes to this question, then **decrease** the ID length for all concerning CIs, so that their IDs do not exceed the maximum length of 60 characters.

> **Note:** Federation and Population flows may be configured for the same integration, for maximum flexibility.

For details about the different types of integrations, see Integration Studio in the *HP Universal CMDB Data Flow Management Guide*.

Five different options are available for creating integration adapters:

1. Jython Adapter:

   - The classic discovery pattern
   - Written in Jython
   - Used for population

   For details, see "Developing Jython Adapters" on page 41.

2. Java Adapter:

   - An adapter that implements one of the adapter interfaces in the Federation SDK Framework.
   - May be used for one or more of Federation, Population, or Data Push (depending on the required implementation).
   - Written from scratch in Java, which allows writing code that will connect to any possible source or target.
   - Suitable for jobs that connect to a single data source or target.

   For details, see "Developing Java Adapters" on page 217.

3. Generic DB Adapter:

   - An abstract adapter based on the Java Adapter that uses the Federation SDK Framework.
   - Allows creation of adapters that connect to external data repositories.
   - Supports both Federation and Population (with a Java plug-in implemented for changes support).
   - Relatively easy to define, as it is based mainly on XML and property configuration files.
   - Main configuration is based on an **orm.xml** file that maps between UCMDB classes and database columns.
   - Suitable for jobs that connect to a single data source.

   For details, see "Developing Generic Database Adapters" on page 133.

4. Generic Push Adapter:

   - An abstract adapter based on the Java Adapter (the Federation SDK Framework) and the Jython Adapter.
   - Allows creation of adapters that push data to remote targets.
   - Relatively easy to define, as you need only to define the mapping between UCMDB classes and XML, and a Jython script that pushes the data to the target.

- Suitable for jobs that connect to a single data target.
- Used for Data Push.

For details, see "Developing Push Adapters" on page 258.

5. Enhanced Generic Push Adapter:
   - All the above features of the Generic Push Adapter
   - A root-element-based adapter
   - Maps a UCMDB tree data structure to a target tree data structure

   For details, see "Achieving Data Push using the Generic Adapter" on page 315.

The following table displays the capabilities of each adapter:

Flow/Adapter	Jython Adapter	Java Adapter	Generic DB Adapter	Generic Push Adapter	Enhanced Generic Push Adapter
Population	✓	✓	✓	×	×
Federation	×	✓	✓	×	×
Data Push	×	✓	×	✓	✓

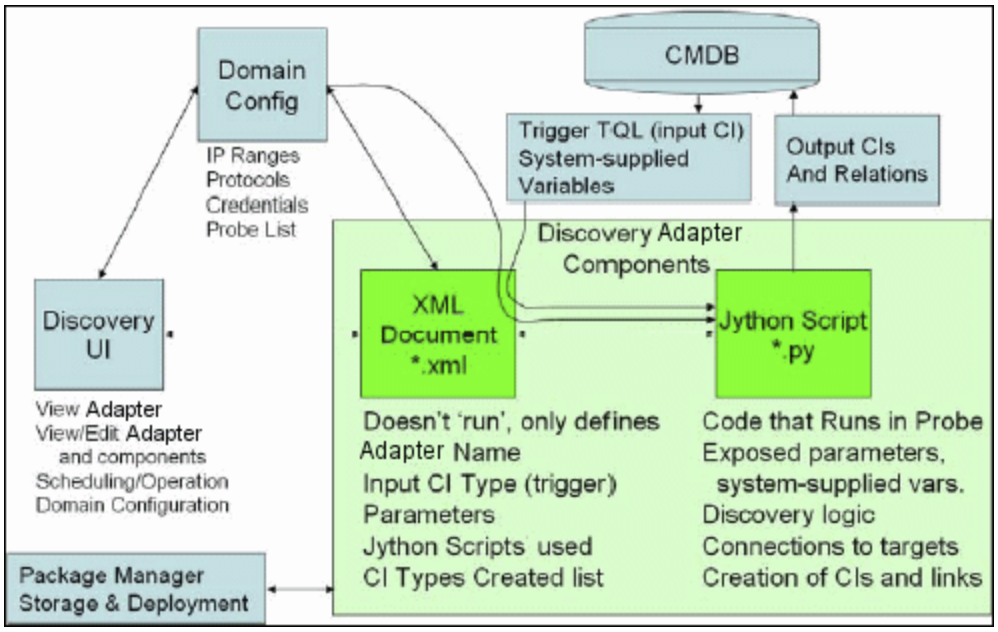
Developing Discovery Content

This section includes:

- "Discovery Adapters and Related Components " below
- "Separating Adapters" on the next page

Discovery Adapters and Related Components

The following diagram shows an adapter's components and the components they interact with to execute discovery. The components in green are the actual adapters, and the components in blue are components that interact with adapters.



Note that the minimum notion of an adapter is two files: an XML document and a Jython script. The Discovery Framework, including input CIs, credentials, and user-supplied libraries, is exposed to the adapter at run time. Both discovery adapter components are administered through Data Flow Management. They are stored operationally in the CMDB itself; although the external package remains, it is not referred to for operation. The Package Manager enables preservation of the new discovery and integration content capability.

Input CIs to the adapter are provided by a TQL, and are exposed to the adapter script in system-supplied variables. Adapter parameters are also supplied as destination data, so you can configure the adapter's operation according to an adapter's specific function.

The DFM application is used to create and test new adapters. You use the Universal Discovery, Adapter Management, and Data Flow Probe Setup pages during adapter writing.

Adapters are stored and transported as packages. The Package Manager application and the JMX console are used to create packages from newly created adapters, and to deploy adapters on new systems.

Separating Adapters

An entire discovery could be defined in a single adapter. But good design demands that a complex system be separated into simpler, more manageable components.

The following are guidelines and best practices for dividing the adapter process:

- Discovery should be done in stages. Each stage should be represented by an adapter that should map an area or tier of the system. Adapters should rely on the previous stage or tier to be discovered, to continue discovery of the system. For example, Adapter A is triggered by an application server TQL result and maps the application server tier. As part of this mapping, a JDBC connection component is mapped. Adapter B registers a JDBC connection component as a trigger TQL and uses the results of adapter A to access the database tier (for example, through the JDBC URL attribute) and maps the database tier.

- **The two-phase connect paradigm**: Most systems require credentials to access their data. This means that a user/password combination needs to be tried against these systems. The DFM administrator supplies credentials information in a secure way to the system and can give several, prioritized login credentials. This is referred to as the **Protocol Dictionary**. If the system is not accessible (for whatever reason) there is no point in performing further discovery. If the connection is successful, there needs to be a way to indicate which credential set was successfully used, for future discovery access.

  These two phases lead to a separation of the two adapters in the following cases:

  - **Connection Adapter**: This is an adapter that accepts an initial trigger and looks for the existence of a remote agent on that trigger. It does so by trying all entries in the Protocol Dictionary which match this agent's type. If successful, this adapter provides as its result a remote agent CI (SNMP, WMI, and so on), which also points to the correct entry in the Protocol Dictionary for future connections. This agent CI is then part of a trigger for the content adapter.

  - **Content Adapter**: This adapter's precondition is the successful connection of the previous adapter (preconditions specified by the TQLs). These types of adapters no longer need to look through all of the Protocol Dictionary since they have a way to obtain the correct credentials from the remote agent CI and use them to log in to the discovered system.

- Different scheduling considerations can also influence discovery division. For example, a system may only be queried during off hours, so even though it would make sense to join the adapter to the same adapter discovering another system, the different schedules mean that you need to create two adapters.

- Discovery of different management interfaces or technologies to discover the same system should be placed in separate adapters. This is so that you can activate the access method appropriate for each system or organization. For example, some organizations have WMI access to machines but do not have SNMP agents installed on them.

Implement a Discovery Adapter

A DFM task has the aim of accessing remote (or local) systems, modeling extracted data as CIs, and saving the CIs to the CMDB. The task consists of the following steps:

1. **Create an adapter.**

   You configure an adapter file that holds the context, parameters, and result types by selecting the scripts that are to be part of the adapter. For details, see "Step 1: Create an Adapter" on page 30.

2. **Create a Discovery job.**

   You configure a job with scheduling information and a trigger query. For details, see "Step 2: Assign a Job to the Adapter" on page 38.

3. **Edit Discovery code.**

   You can edit the Jython or Java code that is contained in the adapter files and that refers to the DFM Framework. For details, see "Step 3: Create Jython Code" on page 39.

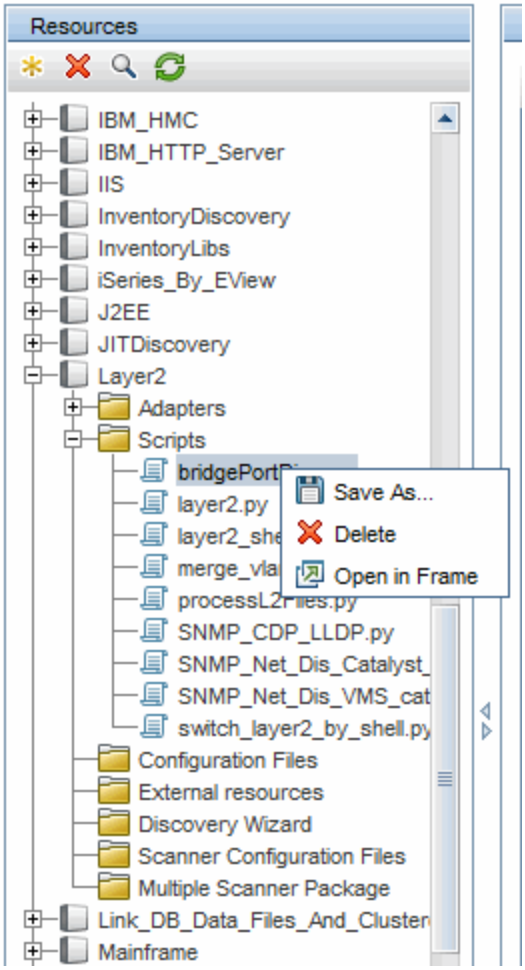
To write new adapters, you create each of the above components, each one of which is automatically bound to the component in the previous step. For example, once you create a job and select the relevant adapter, the adapter file binds to the job.

Adapter Code

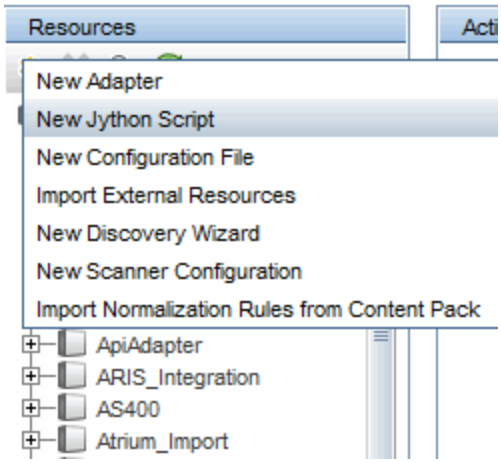
The actual implementation of connecting to the remote system, querying its data, and mapping it as CMDB data is performed by the Jython code. For example, the code contains the logic for connecting to a database and extracting data from it. In this case, the code expects to receive a JDBC URL, a user name, a password, a port, and so on. These parameters are specific for each instance of the database that answers the TQL query. You define these variables in the adapter (in the Trigger CI data) and when the job runs, these specific details are passed to the code for execution.

The adapter can refer to this code by a Java class name or a Jython script name. In this section we discuss writing DFM code as Jython scripts.

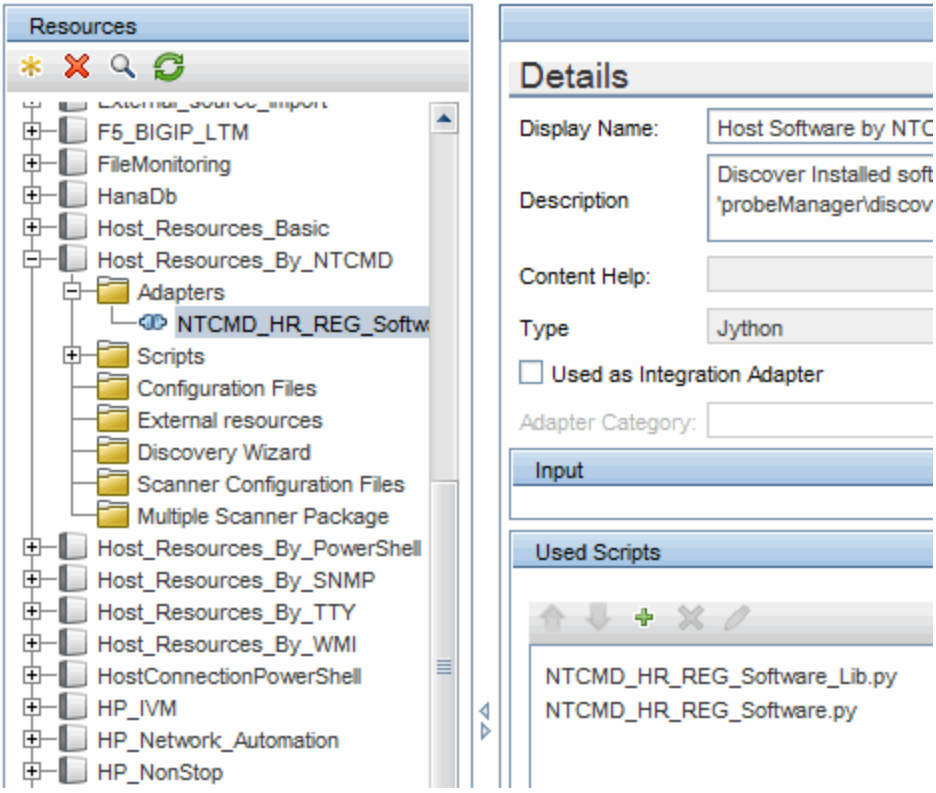
An adapter can contain a list of scripts to be used when running discovery. When creating a new adapter, you usually create a new script and assign it to the adapter. A new script includes basic templates, but you can use one of the other scripts as a template by right-clicking it and selecting **Save as**:



For details on writing new Jython scripts, see "Step 3: Create Jython Code" on page 39. You add scripts through the Resources pane:



The list of scripts are run one after the other, in the order in which they are defined in the adapter:



Note: A script must be specified even though it is being used solely as a library by another script. In this case, the library script must be defined before the script using it. In this example, the `processdbutils.py` script is a library used by the last `host_processes.py` script. Libraries are distinguished from regular runnable scripts by the lack of the `DiscoveryMain()` function.

Step 1: Create an Adapter

An adapter can be considered as the definition of a function. This function defines an input definition, runs logic on the input, defines the output, and provides a result.

Each adapter specifies input and output: Both input and output are Trigger CIs that are specifically defined in the adapter. The adapter extracts data from the input Trigger CI and passes this data as parameters to the code. Data from related CIs is sometimes passed to the code too. For details, see "Related CIs Window" in the *HP Universal CMDB Data Flow Management Guide*. An adapter's code is generic, apart from these specific input Trigger CI parameters that are passed to the code.

For details on input components, see "Data Flow Management Concepts" in the *HP Universal CMDB Data Flow Management Guide*.

This section includes the following topics:

- "Define Adapter Input (Trigger CIT and Input Query)" below
- "Define Adapter Output" on page 34
- "Override Adapter Parameters" on page 35
- "Override Probe Selection - Optional" on page 36
- "Configure a classpath for a remote process - Optional" on page 37

1. **Define Adapter Input (Trigger CIT and Input Query)**

   You use the Trigger CIT and Input Query components to define specific CIs as adapter input:

   - The Trigger CIT defines which CIT is used as the input for the adapter. For example, for an adapter that is going to discover IPs, the input CIT is Network.
   - The Input query is a regular, editable query that defines the query against the CMDB. The Input Query defines additional constraints on the CIT (for example, if the task requires a `hostID` or `application_ip` attribute), and can define more CI data, if needed by the adapter.

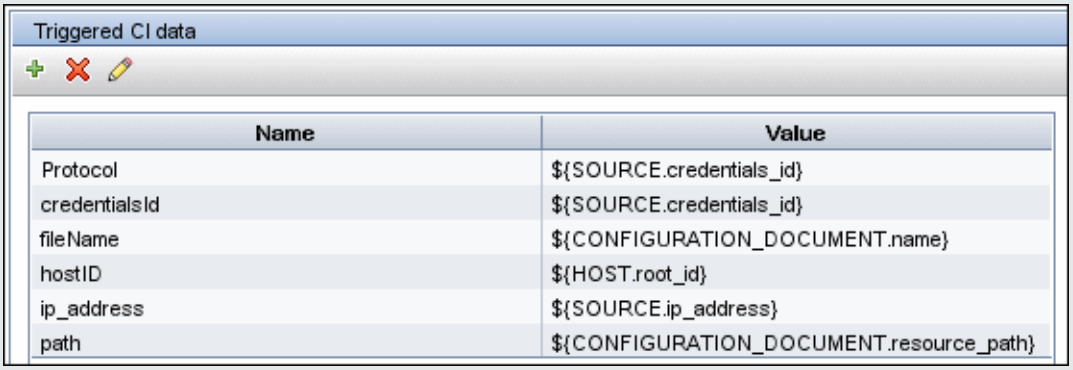
     If the adapter requires additional information from the CIs that are related to the Trigger CI, you can add additional nodes to the input TQL. For details, see "How to Add Query Nodes and Relationships to a TQL Query" in the *HP Universal CMDB Modeling Guide*.
   - The Trigger CI data contains all the required information on the Trigger CI as well as information from the other nodes in the Input TQL, if they are defined. DFM uses variables to retrieve data from the CIs. When the task is downloaded to the Probe, the Trigger CI data variables are replaced with actual values that exist on the attributes for real CI instances.
   - If the value of the destination data is a list, you can define the number of items from the list to be sent to the probe. To define it, add a colon after the default value followed by the number of items. If there is no default value for the destination data, enter two colons.

     For example, if the following destination data is entered: `name=portId, value= ${PHYSICALPORT.root_id:NA:1}` or `name=portId, value= ${PHYSICALPORT.root_id::1}`, only the first port from the port list is sent to the probe.

   Example of Replacing Variables with Actual Data:

   In this example, variables replace the **IpAddress** CI data with actual values that exist on real **IpAddress** CI instances in your system.

The Triggered CI data for the **IpAddress** CI includes a `fileName` variable. This variable enables the replacement of the **CONFIGURATION_DOCUMENT** node in the Input TQL with the actual values of the configuration file located on a host:

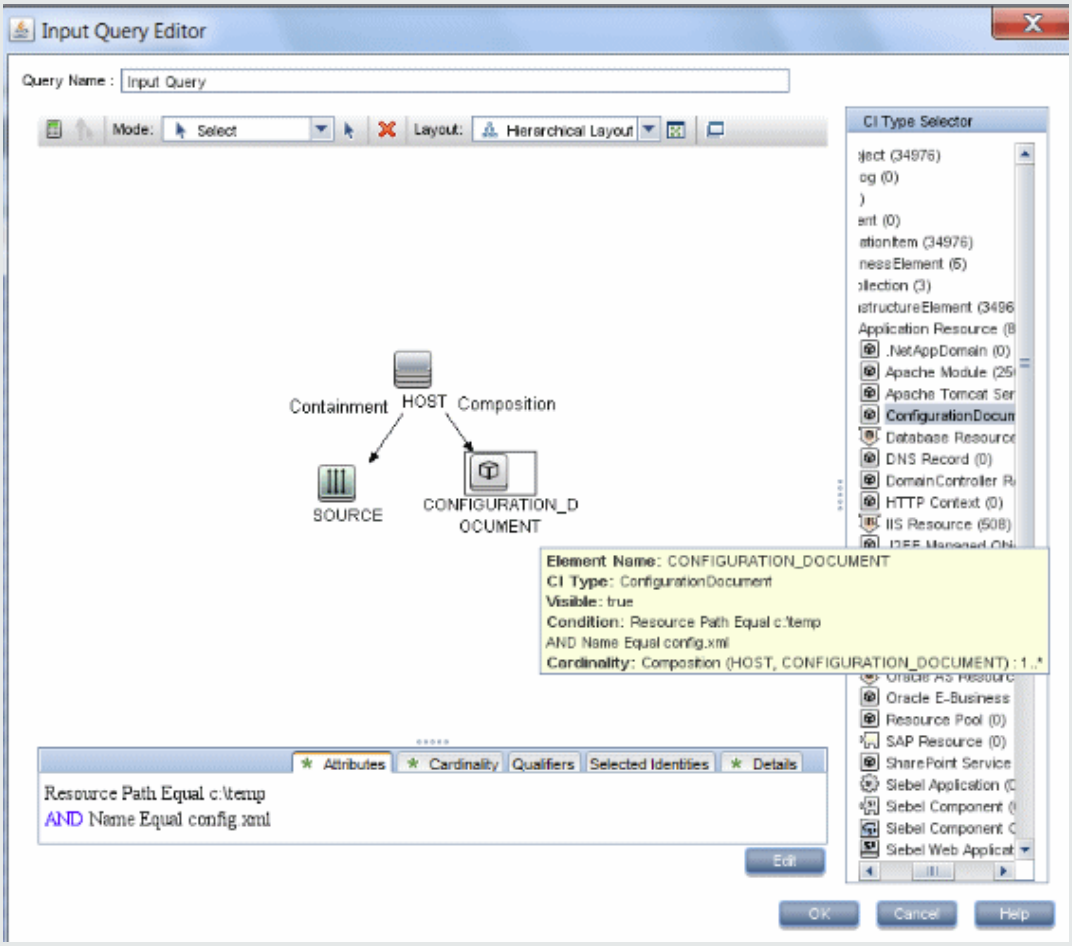

Triggered CI data

Name	Value
Protocol	${SOURCE.credentials_id}
credentialsId	${SOURCE.credentials_id}
fileName	${CONFIGURATION_DOCUMENT.name}
hostID	${HOST.root_id}
ip_address	${SOURCE.ip_address}
path	${CONFIGURATION_DOCUMENT.resource_path}

The Trigger CI data is uploaded to the Probe with all variables replaced by actual values. The adapter script includes a command to use the DFM Framework to retrieve the actual values of the defined variables:

```
Framework.getTriggerCIData ('ip_address')
```

The `fileName` and `path` variables use the `data_name` and `document_path` attributes from the **CONFIGURATION_DOCUMENT** node (defined in the Input Query Editor – see previous example).



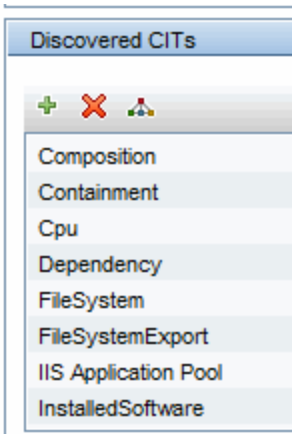
Click thumbnail to view full size image.

The `Protocol`, `credentialsId`, and `ip_address` variables use the `root_class`, `credentials_id`, and `application_ip` attributes:

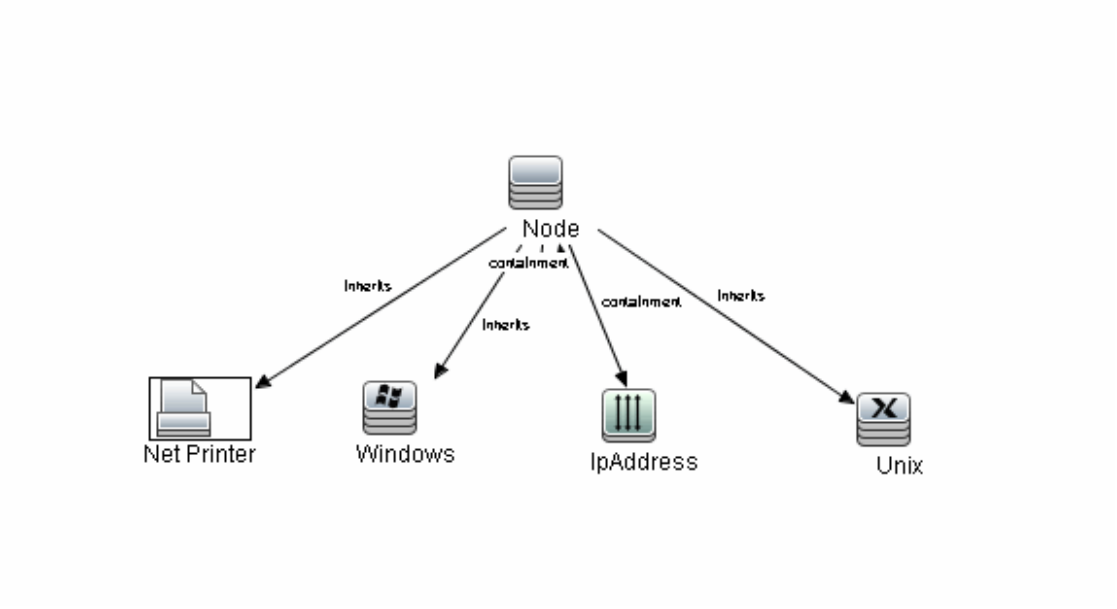
Key	Name	Display Name	Type	Description	Default Value	Visible
	ack_cleared_time	ack_cleared_time	long			
	ack_id	ack_id	string			
	BODY_ICON	BODY_ICON	string		host	
	city	City	string	City location		✔
	codepage	CodePage	string	System su...		
	contextmenu	Context Menu	string_list	Context me...	itCIs	
	country	Country	string	Country loc...		✔
	credentials_id	Reference to the cre...	string	Reference ...		
	data_adminstate	Admin State	adminstate...	Admin State	Managed	

2. **Define Adapter Output**

The output of the adapter is a list of discovered CIs (**Data Flow Management > Adapter Management > Adapter Definition** tab **> Discovered CITs**) and the links between them:



You can also view the CITs as a topology map, that is, the components and the way in which they are linked together (click the **View Discovered CITs as Map** button):



The discovered CIs are returned by the DFM code (that is, the Jython script) in the format of UCMDB's `ObjectStateHolderVector`. For details, see "Results Generation by the Jython Script" on page 46.

Example of Adapter Output:

In this example, you define which CITs are to be part of the IP CI output.

a. Access **Data Flow Management > Adapter Management**.

b. In the Resources pane, select **Network > Adapters > NSLOOKUP_on_Probe**.

c. In the Adapter Definition tab, locate the Discovered CITs pane.

d. The CITs that are to be part of the adapter output are listed. Add CITs to, or remove from, the list. For details, see "Adapter Definition Tab" in the *HP Universal CMDB Data Flow Management Guide*.

3. ## Override Adapter Parameters

To configure an adapter for more than one job, you can override adapter parameters. For example, the adapter `SQL_NET_Dis_Connection` is used by both the `MSSQL Connection by SQL` and the `Oracle Connection by SQL` jobs.

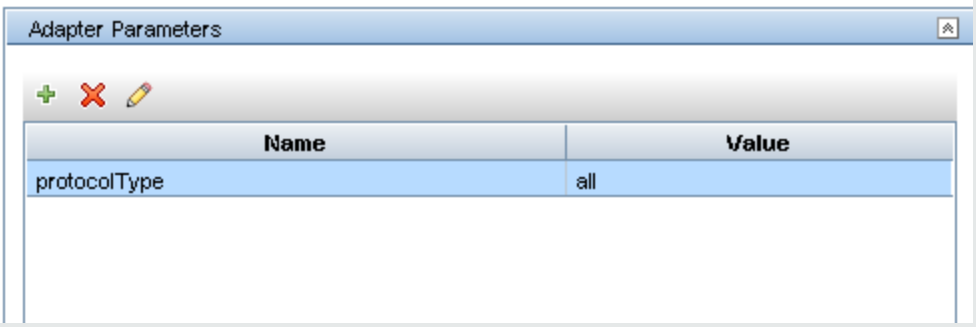
Example of Overriding an Adapter Parameter:

This example illustrates overriding an adapter parameter so that one adapter can be used to discover both Microsoft SQL Server and Oracle databases.

a. Access **Data Flow Management > Adapter Management**.

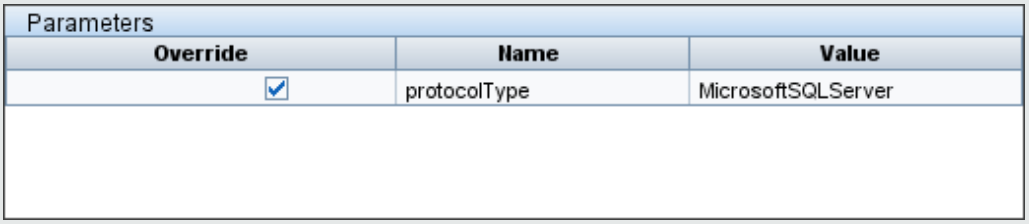
b. In the Resources pane, select **Database_Basic > Adapters > SQL_NET_Dis_Connection**.

c. In the Adapter Definition tab, locate the **Adapter Parameters** pane. The `protocolType` parameter has a value of **all**:



d. Right-click the **SQL_NET_Dis_Connection_MsSql** adapter and choose **Go to Discovery Job > MSSQL Connection by SQL**.

e. Display the **Properties** tab. Locate the Parameters pane:



The `all` value is overwritten with the `MicrosoftSQLServer` value.

Note: The **Oracle Connection by SQL** job includes the same parameter but the value is overwritten with an Oracle value.

For details on adding, deleting, or editing parameters in the Adapter Parameters pane, see "Adapter Definition Tab" in the *HP Universal CMDB Data Flow Management Guide*.

DFM begins looking for Microsoft SQL Server instances according to this parameter.

4. **Override Probe Selection - Optional**

On the UCMDB server there is a dispatching mechanism that takes the trigger CIs received by the UCMDB and automatically chooses which probe should run the job for each trigger CI according to one of the following options:

- **For the IP address CI type:** Take the probe that is defined for this IP.
- **For the running software CI type:** Use the attributes **application_ip** and **application_ip_domain** and choose the probe that is defined for the IP in the relevant domain.
- **For other CI types:** Take the node's IP according to the CI's related node (if it exists).

The automatic probe selection is done according to the CI's related node. After obtaining the CI's related node, the dispatching mechanism chooses one of the node's IPs and chooses the probe according to the probe's network scope definitions.

In the following cases, you need to specify the probe manually and not use the automatic dispatching mechanism:

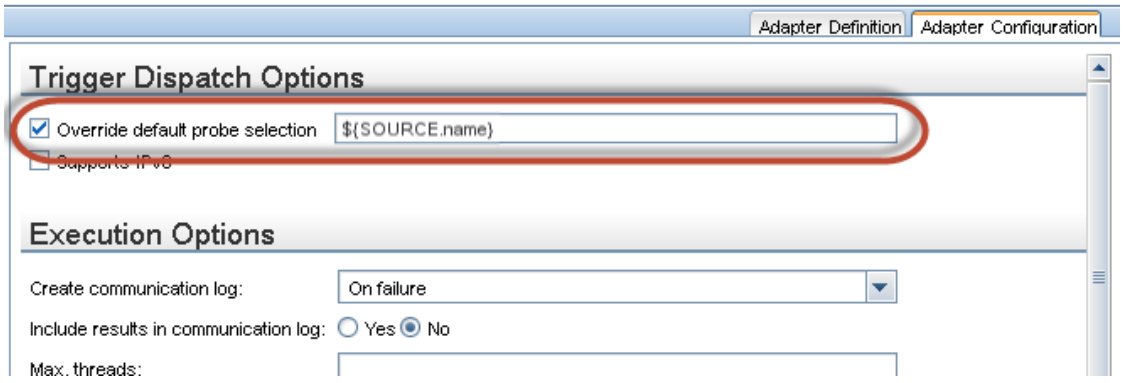
- You already know which probe should be run for the adapter and you do not need the automatic dispatching mechanism to select the probe (for example if the trigger CI is the probe gateway).
- The automatic probe selection might fail. This can happen in the following situations:
  - A trigger CI does not have a related node (such as the network CIT)
  - A trigger CI's node has multiple IPs, each belonging to a different probe.

To manually specify which probe to use with the adapter:

- Select the adapter and click the **Adapter Configuration** tab.
- Under **Trigger Dispatch Options**, select **Override default probe selection.**
- In the box, enter the Probe in one of the following formats:

Probe name	The name of the Probe
IP address	The Probe's IP address—can be defined in either IPv4 or IPv6 format
IP,Domain	IPv4 format: **16.59.63.86,DefaultDomain** IPv6 format: **2001:0:9d46:953c:34a9:1e6b:f2ff:fffe,CustomDomain**
Domain name	The domain from which the Probe should be selected.

For example:



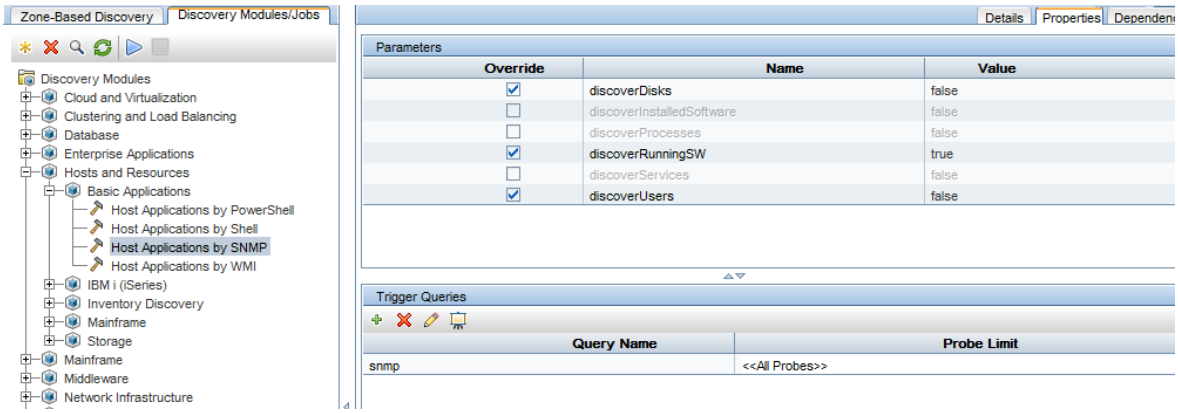
5. **Configure a classpath for a remote process - Optional**

   For details, see "Configure Remote Process Execution" on page 39.

Step 2: Assign a Job to the Adapter

Each adapter has one or more associated jobs that define the execution policy. Jobs enable scheduling the same adapter differently over different sets of Triggered CIs and also enable supplying different parameters for each set.

The jobs appear in the Discovery Modules tree, and this is the entity that the user activates, as shown in the picture below.



Choose a Trigger TQL

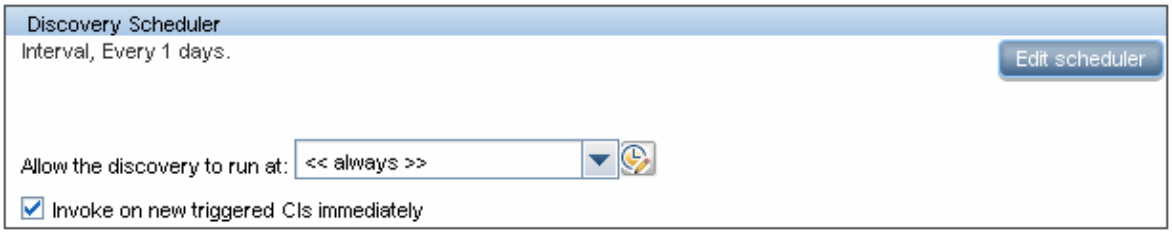
Each job is associated with Trigger TQLs. These Trigger TQLs publish results that are used as Input Trigger CIs for the adapter of this job.

A Trigger TQL can add constraints to an Input TQL. For example, if an input TQL's results are IPs connected to SNMP, a trigger TQL's results can be IPs connected to SNMP within the range 195.0.0.0-195.0.0.10.

Note: A trigger TQL must refer to the same objects that the input TQL refers to. For example, if an input TQL queries for IPs running SNMP, you cannot define a trigger TQL (for the same job) to query for IPs connected to a host, because some of the IPs may not be connected to an SNMP object, as required by the input TQL.

Set Scheduling Information

The scheduling information for the Probe specifies when to run the code on Trigger CIs. If the **Invoke on new triggered CIs Immediately** check box is selected, the code also runs once on each Trigger CI when it reaches the Probe, regardless of future schedule settings.



For each scheduled occurrence for each job, the Probe runs the code against all Trigger CIs accumulated for that job. For details, see Discovery Scheduler Dialog Box in the *HP Universal CMDB Data Flow Management Guide*.

Override Adapter Parameters

When configuring a job you can override the adapter parameters. For details, see "Override Adapter Parameters" on page 35.

Step 3: Create Jython Code

HP Universal CMDB uses Jython scripts for adapter-writing. For example, the `SNMP_Connection.py` script is used by the `SNMP_NET_Dis_Connection` adapter to try and connect to machines using SNMP. Jython is a language based on Python and powered by Java.

For details on how to work in Jython, you can refer to these websites:

- http://www.jython.org
- http://www.python.org

For details, see "Create Jython Code" on page 41.

Configure Remote Process Execution

You can run discovery for a discovery job in a process separate from the Data Flow Probe's process.

For example, you can run the job in a separate remote process if the job uses **.jar** libraries that are a different version than the Probe's libraries or that are incompatible with the Probe's libraries.

You can also run the job in a separate remote process if the job potentially consumes a lot of memory (brings a lot of data) and you want to isolate the Probe from potential **OutOMemory** problems.

To configure a job to run as a remote process, define the following parameters in its adapter's configuration file:

Parameter	Description
remoteJVMArgs	JVM parameters for the remote Java process.
runInSeparateProcess	When set to **true**, the discovery job runs in a separate process.

remoteJVMClasspath	(Optional) Enables customization of the classpath of the remote process, overriding the default Probe classpath. This is useful if there might be version incompatibility between the Probe's jars and custom jars required for the customer-defined discovery.
	
	If the **remoteJVMClasspath** parameter is not defined, or is left empty, the default Probe classpath is used.
	
	If you develop a new discovery job and you want to ensure that the Probe jar library version does not collide with the job's jar libraries, you must use at least the minimal classpath required to execute basic discovery. The minimal classpath is defined in the **DataFlowProbe.properties** file in the **basic_ discovery_minimal_classpath** parameter.
	
	Examples of **remoteJVMClasspath** customization:
	
	• To prepend or append custom jars to the default Probe classpath. customize the **remoteJVMClasspath** parameter as follows:
	
	`custom1.jar;%classpath%;custom2.jar -`
	
	In this case, **custom1.jar** is placed before default Probe classpath, and **custom2.jar** is appended to the Probe classpath.
	
	• To use the minimal classpath, customize the **remoteJVMClasspath** parameter as follows:
	
	`custom1.jar;%minimal_classpath%;custom2.jar`

Chapter 2: Developing Jython Adapters

This chapter includes:

HP Data Flow Management API Reference

For full documentation on the available APIs, see *HP Universal CMDB Data Flow Management API Reference*. These files are located in the following folder:

<UCMDB install directory>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_JavaDoc\index.html

Create Jython Code

HP Universal CMDB uses Jython scripts for adapter-writing. For example, the **SNMP_Connection.py** script is used by the **SNMP_NET_Dis_Connection** adapter to try to connect to machines using SNMP. Jython is a language based on Python and powered by Java.

For details on how to work in Jython, you can refer to these websites:

- http://www.jython.org
- http://www.python.org

The following section describes the actual writing of Jython code inside the DFM Framework. This section specifically addresses those contact points between the Jython script and the Framework that it calls, and also describes the Jython libraries and utilities that should be used whenever possible.

Note:

- Scripts written for Universal Discovery should be compatible with Jython version 2.5.3.
- For full documentation on the available APIs, see the *HP Universal CMDB Data Flow Management API Reference*.

This section includes the following topics:

Use External Java JAR Files within Jython

When developing new Jython scripts, external Java Libraries (JAR files) or third-party executable files are sometimes needed as Java utility archives, connection archives such as JDBC Driver JAR files, or executable files (for example, **nmap.exe** is used for credential-less discovery).

These resources should be bundled in the package under the **External Resources** folder. Any resource put in this folder is automatically sent to any Probe that connects to your HP Universal CMDB server.

In addition, when discovery is launched, any JAR file resource is loaded into the Jython's classpath, making all the classes within it available for import and use.

Execution of the Code

After a job is activated, a task with all the required information is downloaded to the Probe.

The Probe starts running the DFM code using the information specified in the task.

The Jython code flow starts running from a main entry in the script, executes code to discover CIs, and provides results of a vector of discovered CIs.

Modifying Out-of-the-Box Scripts

When making out-of-the-box script modifications, make only minimal changes to the script and place any necessary methods in an external script. You can track changes more efficiently and, when moving to a newer HP Universal CMDB version, your code is not overwritten.

For example, the following single line of code in an out-of-the-box script calls a method that calculates a Web server name in an application-specific way:

```
serverName = iplanet_cspecific.PlugInProcessing(serverName, transportHN, mam_utils)
```

The more complex logic that decides how to calculate this name is contained in an external script:

```
# implement customer specific processing for 'servername' attribute of httpplugin
#
def PlugInProcessing(servername, transportHN, mam_utils_handle):
        # support application-specific HTTP plug-in naming
        if servername == "appsrv_instance":
                # servername is supposed to match up with the j2ee server name,
however some groups do strange things with their
                # iPlanet plug-in files. this is the best work-around we could
find. this join can't be done with IP address:port
                # because multiple apps on a web server share the same IP:port for
multiple websphere applications
                logger.debug('httpcontext_webapplicationserver attribute has been
changed from [' + servername + '] to [' + transportHN[:5] + '] to facilitate
websphere enrichment')
                servername = transportHN[:5]
        return servername
```

Save the external script in the External Resources folder. For details, see Resources Pane in the *HP Universal CMDB Data Flow Management Guide*. If you add this script to a package, you can use this script for other jobs, too. For details on working with Package Manager, see Package Manager in the *HP Universal CMDB Administration Guide*.

During upgrade, the change you make to the single line of code is overwritten by the new version of the out-of-the-box script, so you will need to replace the line. However, the external script is not overwritten.

Structure of the Jython File

The Jython file is composed of three parts in a specific order:

1. Imports
2. Main Function - DiscoveryMain
3. Functions definitions (optional)

The following is an example of a Jython script:

```
#   imports section
from appilog.common.system.types import ObjectStateHolder
from appilog.common.system.types.vectors import ObjectStateHolderVector
# Function definition
def foo:
    # do something
#   Main Function
def DiscoveryMain(Framework):
    OSHVResult = ObjectStateHolderVector()
    ## Write implementation to return new result CIs here...
    return OSHVResult
```

Imports

Jython classes are spread across hierarchical namespaces. In version 7.0 or later, unlike in previous versions, there are no implicit imports, and so every class you use must be imported explicitly. (This change was made for performance reasons and to enable an easier understanding of the Jython script by not hiding necessary details.)

- To import a Jython script:

  ```
  import logger
  ```

- To import a Java class:

  ```
  from appilog.collectors.clients import ClientsConsts
  ```

Main Function – DiscoveryMain

Each Jython runnable script file contains a main function: DiscoveryMain.

The `DiscoveryMain` function is the main entry into the script; it is the first function that runs. The main function may call other functions that are defined in the scripts:

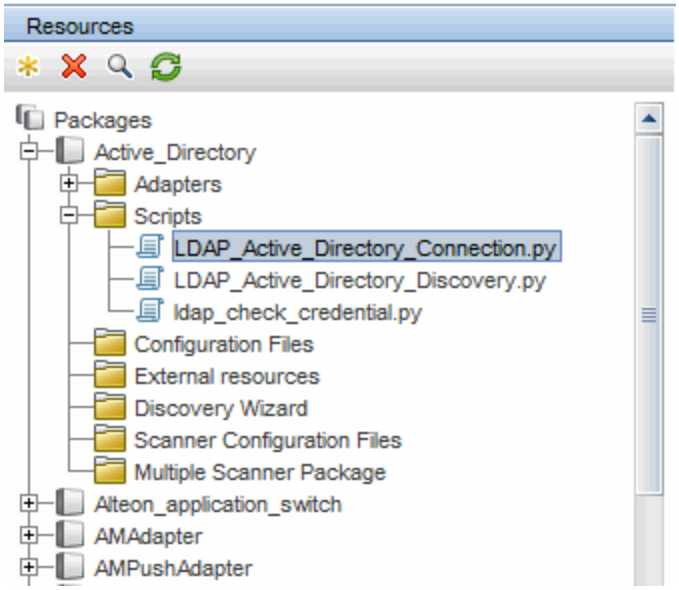
```
def DiscoveryMain(Framework):
```

The Framework argument must be specified in the main function definition. This argument is used by the main function to retrieve information that is required to run the scripts (such as information on the Trigger CI and parameters) and can also be used to report on errors that occur during the script run.

You can create a Jython script without any main method. Such scripts are used as library scripts that are called from other scripts.

Functions Definition

Each script can contain additional functions that are called from the main code. Each such function can call another function, which either exists in the current script or in another script (use the `import` statement). Note that to use another script, you must add it to the Scripts section of the package:



Example of a Function Calling Another Function:

In the following example, the main code calls the `doQueryOSUsers(..)` method which calls an internal method `doOSUserOSH(..)`:

```
def doOSUserOSH(name):
    sw_obj = ObjectStateHolder('winosuser')

    sw_obj.setAttribute('data_name', name)
    # return the object
    return sw_obj
def doQueryOSUsers(client, OSHVResult):
    _hostObj = modeling.createHostOSH(client.getIpAddress())
    data_name_mib = '1.3.6.1.4.1.77.1.2.25.1.1,1.3.6.1.4.1.77.1.2.25.1.2,string'
    resultSet = client.executeQuery(data_name_mib)
```

```
    while resultSet.next():
        UserName = resultSet.getString(2)
        ########## send object ##############
        OSUserOSH = doOSUserOSH(UserName)
        OSUserOSH.setContainer(_hostObj)
        OSHVResult.add(OSUserOSH)
def DiscoveryMain(Framework):
    OSHVResult = ObjectStateHolderVector()
    try:
        client = Framework.createClient(Framework.getTriggerCIData
(BaseClient.CREDENTIALS_ID))
    except:
        Framework.reportError('Connection failed')
    else:
        doQueryOSUsers(client, OSHVResult)
        client.close()
    return OSHVResult
```

If this script is a global library that is relevant to many adapters, you can add it to the list of scripts in the `jythonGlobalLibs.xml` configuration file, instead of adding it to each adapter (**Adapter Management > Resources Pane > AutoDiscoveryContent > Configuration Files**).

Results Generation by the Jython Script

Each Jython script runs on a specific Trigger CI, and ends with results that are returned by the return value of the `DiscoveryMain` function.

The script result is actually a group of CIs and links that are to be inserted or updated in the CMDB. The script returns this group of CIs and links in the format of `ObjectStateHolderVector`.

The `ObjectStateHolder` class is a way to represent an object or link defined in the CMDB. The `ObjectStateHolder` object contains the CIT name and a list of attributes and their values. The `ObjectStateHolderVector` is a vector of `ObjectStateHolder` instances.

The ObjectStateHolder Syntax

This section explains how to build the DFM results into a UCMDB model.

Example of Setting Attributes on the CIs:

The ObjectStateHolder class describes the DFM result graph. Each CI and link (relationship) is placed inside an instance of the `ObjectStateHolder` class as in the following Jython code sample:
siebel application server 1 appServerOSH = ObjectStateHolder('siebelappserver') 2 appServerOSH.setStringAttribute('data_name', sblsvrName) 3 appServerOSH.setStringAttribute ('application_ip', ip) 4 appServerOSH.setContainer(appServerHostOSH)

- Line 1 creates a CI of type **siebelappserver**.
- Line 2 creates an attribute called **data_name** with a value of **sblsvrName** which is a Jython variable set with the value discovered for the server name.
- Line 3 sets a non-key attribute that is updated in the CMDB.
- Line 4 is the building of containment (the result is a graph). It specifies that this application server is contained inside a host (another `ObjectStateHolder` class in the scope).

Note: Each CI being reported by the Jython script must include values for all the key attributes of the CI's CI Type.

Example of Relationships (Links):

The following link example explains how the graph is represented:
1 linkOSH = ObjectStateHolder('route') 2 linkOSH.setAttribute('link_end1', gatewayOSH) 3 linkOSH.setAttribute('link_end2', appServerOSH)

- Line 1 creates the link (that is also of the `ObjectStateHolder` class. The only difference is that `route` is a link CI Type).
- Lines 2 and 3 specify the nodes at the end of each link. This is done using the **end1** and **end2** attributes of the link which must be specified (because they are the minimal key attributes of each link). The attribute values are `ObjectStateHolder` instances. For details on End 1 and End 2, see Link in the *HP Universal CMDB Data Flow Management Guide*.

Caution: A link is directional. You should verify that End 1 and End 2 nodes correspond to valid CITs at each end. If the nodes are not valid, the result object fails validation and is not reported correctly. For details, see CI Type Relationships in the *HP Universal CMDB Modeling Guide*.

Example of Vector (Gathering CIs):

After creating objects with attributes, and links with objects at their ends, you must now group them together. You do this by adding them to an `ObjectStateHolderVector` instance, as follows:

```
oshvMyResult = ObjectStateHolderVector()
oshvMyResult.add(appServerOSH)
oshvMyResult.add(linkOSH)
```

For details on reporting this composite result to the Framework so that it can be sent to the CMDB server, see the sendObjects method.

Once the result graph is assembled in an `ObjectStateHolderVector` instance, it must be returned to the DFM Framework to be inserted into the CMDB. This is done by returning the `ObjectStateHolderVector` instance as the result of the `DiscoveryMain()` function.

Note: For details on creating **OSH** for common CITs, see "modeling.py" on page 69.

Sending Large Amounts of Data

Sending large amounts of data (usually more than 20 KB) is difficult to process in UCMDB. Data of this size should be split into smaller chunks before sending to UCMDB. In order for all the chunks to be correctly inserted to UCMDB, each chunk needs to contain required identification information for the CIs in the chunk. This is a common scenario when developing Jython integrations. The sendObjects method is used to send the results in chunks. If the Jython script sends a large number of results (the default value is 20,000, but this value can be configured in the DataFlowProbe.properties File using the **appilog.agent.local.maxTaskResultSize** key) it should chunk the results according to their topology. This chunking should be performed taking into account identification rules so that the results are entered correctly in UCMDB. If the Jython script does not chunk the results, the probe attempts to chunk them; however, this can lead to poor performance for a large result set.

Note: Chunking should be used for Jython integration adapters and not for regular discovery jobs. This is because discovery jobs usually discover information regarding a specific trigger and do not send large amounts of information. With Jython integrations, large amounts of data are discovered on the single trigger of the integration.

It is also possible to use chunking for a small number of results. In such a case, there is a relationship between CIs in different chunks and the developer of the Jython script has two options:

- Send the entire CI and all of its identification information again in every chunk that contains a link to it.
- Use the UCMDB ID of the CI. To do this, the Jython script has to wait for each chunk to be processed in the UCMDB server in order to get the UCMDB IDs. To enable this mode (called synchronous result sending), add the SendJythonResultsSynchronously tag to the adapter. This tag ensures that when you finish sending the chunk, the UCMDB IDs of the CIs in the chunk have already been received by

the probe. The adapter developer can use the UCMDB IDs for generating the next chunk. To use the UCMDB IDs, use the framework API getIdMapping.

Example of Using getIdMapping

In the first chunk you send nodes. In the second chunk you send processes. The root container of the process is a node. Instead of sending the entire objectStateHolder of the node in the process root_container attribute, you can get the UCMDB ID of the node using the getIdMapping API and use only the node ID in the process root_container attribute to make the chunk smaller.

The Framework Instance

The Framework instance is the only argument that is supplied in the main function in the Jython script. This is an interface that can be used to retrieve information required to run the script (for example, information on trigger CIs and adapter parameters), and is also used to report on errors that occur during the script run. For details, see "HP Data Flow Management API Reference" on page 41.

The correct usage of Framework instance is to pass it as argument to each method that uses it.

Example:

```
def DiscoveryMain(Framework):
         OSHVResult = helperMethod (Framework)
            return OSHVResult
def helperMethod (Framework):
    ....
     probe_name    = Framework.getDestinationAttribute('probe_name')
     ...
     return result
```

This section describes the most important Framework usages:

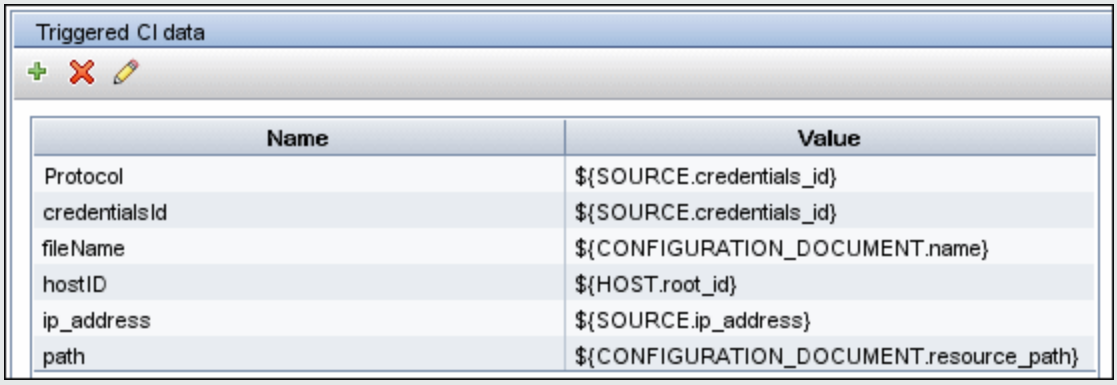
- "Framework.getTriggerCIData(String attributeName)" on the next page
- "Framework.createClient(credentialsId, props)" on the next page
- "Framework.getParameter (String parameterName)" on page 51
- "Framework.reportError(String message) and Framework.reportWarning(String message)" on page 52

Framework.getTriggerCIData(String attributeName)

This API provides the intermediate step between the Trigger CI data defined in the adapter and the script.

Example of Retrieving Credential Information:

You request the following Trigger CI data information:



Name	Value
Protocol	${SOURCE.credentials_id}
credentialsId	${SOURCE.credentials_id}
fileName	${CONFIGURATION_DOCUMENT.name}
hostID	${HOST.root_id}
ip_address	${SOURCE.ip_address}
path	${CONFIGURATION_DOCUMENT.resource_path}

To retrieve the credential information from the task, use this API:

```
credId = Framework.getTriggerCIData('credentialsId')
```

Framework.createClient(credentialsId, props)

You make a connection to a remote machine by creating a client object and executing commands on that client. To create a client, retrieve the `ClientFactory` class. The getClientFactory() method receives the type of the requested client protocol. The protocol constants are defined in the ClientsConsts class. For details on credentials and supported protocols, see the *HP UCMDB Discovery and Integrations Content Guide*.

Example of Creating a Client Instance for the Credentials ID:

To create a `Client` instance for the credentials ID:

```
properties = Properties()
codePage = Framework.getCodePage()
properties.put( BaseAgent.ENCODING, codePage)
client = Framework.createClient(credentailsID ,properties)
```

You can now use the `Client` instance to connect to the relevant machine or application.

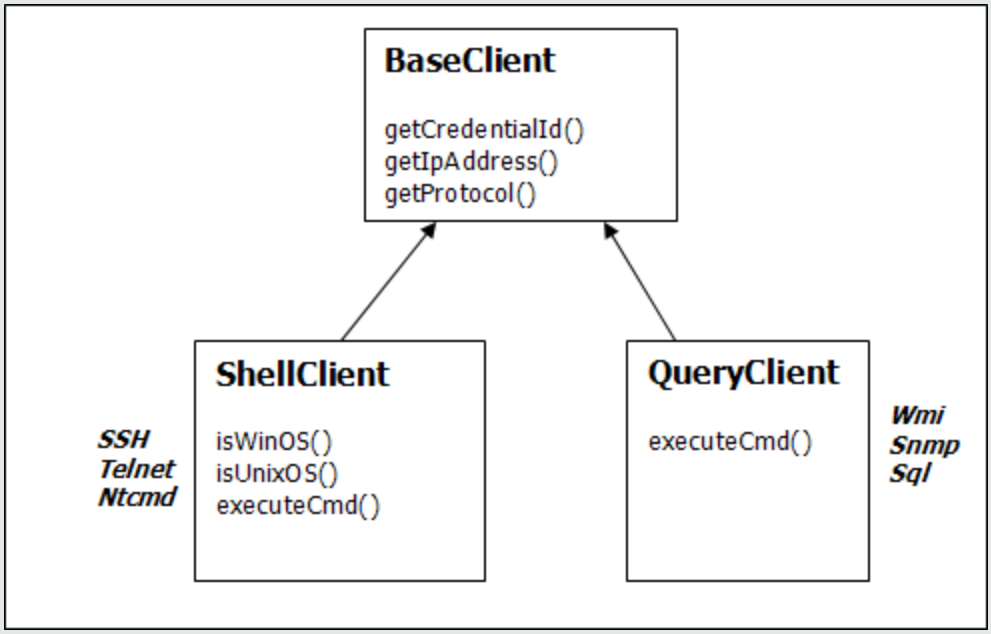
Example of Creating a WMI Client and Running a WMI Query:

To create a WMI client and run a WMI query using the client:

```
wmiClient = Framework.createClient(credential)
resultSet = wmiClient. executeQuery("SELECT TotalPhysicalMemory
                                                    FROM    Win32_
LogicalMemoryConfiguration")
```

Note: To make the `createClient()` API work, add the following parameter to the Trigger CI data parameters: **credentialsId = ${SOURCE.credentials_id}** in the Triggered CI Data pane. Or you can manually add the credentials ID when calling the function:
wmiClient = clientFactory().createClient(credentials_id).

The following diagram illustrates the hierarchy of the clients, with their commonly-supported APIs:

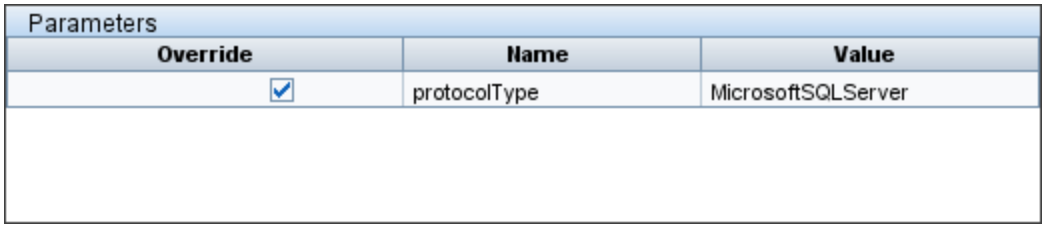


For details on the clients and their supported APIs, see BaseClient, ShellClient, and QueryClient in the DFM Framework. These files are located in the following folder:

<UCMDB root directory>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_Schema\webframe.html

Framework.getParameter (String parameterName)

In addition to retrieving information on the Trigger CI, you often need to retrieve an adapter parameter value. For example:


Parameters

Override	Name	Value
☑	protocolType	MicrosoftSQLServer

Example of Retrieving the Value of the protocolType Parameter:

To retrieve the value of the `protocolType` parameter from the Jython script, use the following API:

```
protocolType = Framework.getParameterValue('protocolType')
```

Framework.reportError(String message) and Framework.reportWarning(String message)

Some errors (for example, connection failure, hardware problems, timeouts) can occur during a script run. When such errors are detected, Framework can report on the problem. The message that is reported reaches the server and is displayed for the user.

Example of a Report Error and Message:

The following example illustrates the use of the `reportError(<Error Msg>)` API:

```
try:
    client = Framework.createClient(Framework.getTriggerCIData
(BaseClient.CREDENTIALS_ID))

except:
    strException = str(sys.exc_info()[1]).strip()
    Framework. reportError ('Connection failed: %s' % strException)
```

You can use either one of the APIs—`Framework.reportError(String message)`, `Framework.reportWarning(String message)`—to report on a problem. The difference between the two APIs is that when reporting an error, the Probe saves a communication log file with the entire session's parameters to the file system. In this way you are able to track the session and better understand the error.

For details on error messages, see "Error Messages" on page 71.

Finding the Correct Credentials (for Connection Adapters)

An adapter trying to connect to a remote system needs to try all possible credentials. One of the parameters needed when creating a client is the credentials ID. The connection script gains access to possible credential sets and tries them one by one using the `Framework.getAvailableProtocols()`

method. When one credential set succeeds, the adapter reports a CI connection object on the host of this trigger CI (with the credentials ID that matches the IP) to the CMDB. Subsequent adapters can use this connection object CI directly to connect to the credential set (that is, the adapters do not have to try all possible credentials again).

Note: Access to sensitive data (passwords, private keys, and so on) is blocked for the following protocol types:

sshprotocol, ntadminprotocol, as400protocol, vmwareprotocol, wmiprotocol, vcloudprotocol, sapjmxprotocol, webspherepreprotocol, siebelgtwyprotocol, sapprotocol, ldapprotocol, udaprotocol, ntcmdprotocol, snmpprotocol, jbossprotocol, telnetprotocol, powershellprotocol, sqlprotocol, weblogicprotocol

Utilization of these protocol types should be done by using dedicated clients.

The following example shows how to obtain all entries of the SNMP protocol. Note that here the IP is obtained from the Trigger CI data (`# Get the Trigger CI data values`).

The connection script requests all possible protocol credentials (`# Go over all the protocol credentials`) and tries them in a loop until one succeeds (`resultVector`). For details, see the **two-phase connect paradigm** entry in "Separating Adapters" on page 26.

Example

```
import logger
import netutils
import sys
import errorcodes
import errorobject

# Java imports
from java.util import Properties
from com.hp.ucmdb.discovery.common import CollectorsConstants
from appilog.common.system.types.vectors import ObjectStateHolderVector
from com.hp.ucmdb.discovery.library.clients import ClientsConsts
from com.hp.ucmdb.discovery.library.scope import DomainScopeManager

TRUE = 1
FALSE = 0

def mainFunction(Framework, isClient, ip_address = None):
       _vector = ObjectStateHolderVector()
       errStr = ''
       ip_domain  = Framework.getDestinationAttribute('ip_domain')
       # Get the Trigger CI data values
       ip_address = Framework.getDestinationAttribute('ip_address')
```

```
if (ip_domain == None):
   ip_domain = DomainScopeManager.getDomainByIp(ip_address, None)

protocols = netutils.getAvailableProtocols(Framework,
ClientsConsts.SNMP_PROTOCOL_NAME, ip_address, ip_domain)
if len(protocols) == 0:
    errStr = 'No credentials defined for the triggered ip'
    logger.debug(errStr)
    errObj = errorobject.createError(errorcodes.NO_CREDENTIALS_FOR_
TRIGGERED_IP, [ClientsConsts.SNMP_PROTOCOL_NAME], errStr)
    return (_vector, errObj)

connected = 0
# Go over all the protocol credentials
for protocol in protocols:
    client = None
    try:
        try:
            logger.debug('try to get snmp agent for: %s:%s' % (ip_
address, ip_domain))
            if (isClient == TRUE):
                properties = Properties()
                properties.setProperty
(CollectorsConstants.DESTINATION_DATA_IP_ADDRESS, ip_address)
                properties.setProperty
(CollectorsConstants.DESTINATION_DATA_IP_DOMAIN, ip_domain)
                client = Framework.createClient(protocol, properties)
            else:
                properties = Properties()
                properties.setProperty
(CollectorsConstants.DESTINATION_DATA_IP_ADDRESS, ip_address)
                client = Framework.createClient(protocol, properties)
            logger.debug('Running test connection queries')
            testConnection(client)
            Framework.saveState(protocol)
            logger.debug('got snmp agent for: %s:%s' % (ip_address,
ip_domain))
            isMultiOid = client.supportMultiOid()
            logger.debug('snmp server isMultiOid state=%s'
%isMultiOid)

         client.close()
         client = None

        except:
            if client != None:
                client.close()
                client = None
            logger.debugException('Unexpected SNMP_AGENT Exception:')
```

```
                    lastExceptionStr = str(sys.exc_info()[1]).strip()
            finally:
                if client != None:
                    client.close()
                    client = None

        return (_vector, error)
```

Handling Exceptions from Java

Some Java classes throw an exception upon failure. It is recommended to catch the exception and handle it, otherwise it causes the adapter to terminate unexpectedly.

When catching a known exception, in most cases you should print its stack trace to the log and issue a proper message to the UI.

Note: It is very important to import the Java base exception class as shown in the following example due to the presence of the base exception class in Python with the same name.

```
from java.lang import Exception as JException
try:
       client = Framework.createClient(Framework.getTriggerCIData(BaseClient.CREDENTIALS_
ID))
except JException, ex:
       # process java exceptions only
       Framework.reportError('Connection failed')
       logger.debugException(str(ex))
        return
```

If the exception is not fatal and the script can continue, you should omit the call for the `reportError()` method and enable the script to continue.

Troubleshooting Migration from Jython Version 2.1 to 2.5.3

Universal Discovery now uses Jython version 2.5.3. All out-of-the-box scripts have been properly migrated. If you developed your own Jython scripts prior to this upgrade for use by Discovery, you may run into the following issues and have to make the fixes indicated.

Note: You must be an experienced Jython developer to make these changes.

String Formatting

- **Error message:** `TypeError: int argument required`
- **Possible cause:** Using string formatting to decimal integer from string variable containing integer data.
- **Problematic Jython 2.1 code:**

```
variable = "43"
print "%d" % variable
```

- **Correct Jython 2.5.3 code:**

```
variable = "43"
print "%s" % variable
```

or

```
variable = "43"
print "%d" % int(variable)
```

Checking String Type

The code below may not work correctly if input contains unicode strings:

- **Problematic Jython 2.1 code:** `isinstance(unicodeStringVariable,'')`
- **Correct Jython 2.5.3 code:** `isinstance(unicodeStringVariable,basestring)`

  The comparison should be done with `basestring` to test whether an object is an instance of `str` or unicode.

Non-ASCII character in file

- **Error Message:**
  `SyntaxError: Non-ASCII character in file 'x', , but no encoding declared; see http://www.python.org/peps/pep-0263.html for details`
- **Correct Jython 2.5.3 code:** (add this to the first line in the file)

```
# coding: utf-8
```

Import sub-packages

- **Error message:**
  `AttributeError: 'module' object has no attribute 'sub_package_name'`
- **Possible cause:** A sub-package is imported without explicitly specifying the name of sub-package in the import statement.
- **Problematic Jython 2.1 code:**

  ```
  import a
  print dir(a.b)
  ```

  The sub-package is not explicitly imported.
- **Correct Jython 2.5.3 code:**

  ```
  import a.b
  ```

  or

  ```
  from a import b
  ```

Iterator Changes

Starting from Jython 2.2, the **__iter__** method is used to loop over a collection in the scope of a **for-in** block. The iterator should implement the **next** method, returning an appropriate element or throw the **StopIteration** error if it reached the end of the collection. If the **__iter__** method is not implemented, the **getitem** method is used instead.

Raising Exceptions

- **Jython 2.1 method for raising exceptions is obsolete:**
  `raise Exception, 'Failed getting contents of file'`
- **Recommended Jython 2.5.3 method for raising exceptions:**
  `raise Exception('Failed getting contents of file')`

Support Localization in Jython Adapters

The multi-lingual locale feature enables DFM to work across different operating system (OS) languages, and to enable appropriate customizations at runtime.

This section includes:

Add Support for a New Language

This task describes how to add support for a new language.

This task includes the following steps:

1. **Add a Resource Bundle (*.properties Files)**

   Add a resource bundle according to the job that is to be run. The following table lists the DFM jobs and the resource bundle that is used by each job:

Job	Base Name of Resource Bundle
File Monitor by Shell	langFileMonitoring

Job	Base Name of Resource Bundle
Host Resources and Applications by Shell	langHost_Resources_By_TTY, langTCP
Hosts by Shell using NSLOOKUP in DNS Server	langNetwork
Host Connection by Shell	langNetwork
Collect Network Data by Shell or SNMP	langTCP
Host Resources and Applications by SNMP	langTCP
Microsoft Exchange Connection by NTCMD, Microsoft Exchange Topology by NTCMD	msExchange
MS Cluster by NTCMD	langMsCluster

For details on bundles, see "Work with Resource Bundles" on page 62.

2. **Declare and Register the Language Object**

To define a new language, add the following two lines of code to the **shellutils.py** script, that currently contains the list of all supported languages. The script is included in the `AutoDiscoveryContent` package. To view the script, access the Adapter Management window. For details, see Adapter Management Window in the *HP Universal CMDB Data Flow Management Guide*.

a. Declare the language, as follows:

```
LANG_RUSSIAN = Language(LOCALE_RUSSIAN, 'rus', ('Cp866', 'Cp1251'), (1049,), 866)
```

For details on class language, see "API Reference" on page 63. For details on the Class Locale object, see http://java.sun.com/j2se/1.5.0/docs/api/java/util/Locale.html. You can use an existing locale or define a new locale.

b. Register the language by adding it to the following collection:

```
LANGUAGES = (LANG_ENGLISH, LANG_GERMAN, LANG_SPANISH, LANG_RUSSIAN, LANG_JAPANESE)
```

Change the Default Language

If the OS language cannot be determined, the default one is used. The default language is specified in the **shellutils.py** file.

```
#default language for fallback
DEFAULT_LANGUAGE = LANG_ENGLISH
```

To change the default language, you initialize the DEFAULT_LANGUAGE variable with a different language. For details, see "Add Support for a New Language" on page 58.

Determine the Character Set for Encoding

The suitable character set for decoding command output is determined at runtime. The multi-lingual solution is based on the following facts and assumptions:

1. It is possible to determine the OS language in a locale-independent way, for example, by running the **chcp** command on Windows or the **locale** command on Linux.
2. Relation Language-Encoding is well known and can be defined statically. For example, the Russian language has two of the most popular encoding: `Cp866` and `Windows-1251`.
3. One character set for each language is preferable, for example, the preferable character set for Russian language is `Cp866`. This means that most of the commands produce output in this encoding.
4. Encoding in which the next command output is provided is unpredictable, but it is one of the possible encoding for a given language. For example, when working with a Windows machine with a Russian locale, the system provides the **ver** command output in Cp866, but the **ipconfig** command is provided in Windows-1251.
5. A known command produces known key words in its output. For example, the **ipconfig** command contains the translated form of the **IP-Address** string. So the **ipconfig** command output contains **IP-Address** for the English OS, IP-Адрес for the Russian OS, **IP-Adresse** for the German OS, and so on.

Once it is discovered in which language the command output is produced (# 1), possible character sets are limited to one or two (# 2). Furthermore, it is known which key words are contained in this output (# 5).

The solution, therefore, is to decode the command output with one of the possible encoding by searching for a key word in the result. If the key word is found, the current character set is considered the correct one.

Define a New Job to Operate With Localized Data

This task describes how to write a new job that can operate with localized data.

Jython scripts usually execute commands and parse their output. To receive this command output in a properly decoded manner, use the API for the **ShellUtils** class. For details, see "HP Universal CMDB Web Service API Overview" on page 379.

This code usually takes the following form:

```
client = Framework.createClient(protocol, properties)
shellUtils = shellutils.ShellUtils(client)
languageBundle = shellutils.getLanguageBundle ('langNetwork',
shellUtils.osLanguage, Framework)
strWindowsIPAddress = languageBundle.getString('windows_ipconfig_str_ip_address')
ipconfigOutput = shellUtils.executeCommandAndDecode('ipconfig /all',
strWindowsIPAddress)
#Do work with output here
```

1. Create a client:

   ```
   client = Framework.createClient(protocol, properties)
   ```

2. Create an instance of the **ShellUtils** class and add the operating system language to it. If the language is not added, the default language is used (usually English):

   ```
   shellUtils = shellutils.ShellUtils(client)
   ```

   During object initialization, DFM automatically detects machine language and sets preferable encoding from the predefined `Language` object. Preferable encoding is the first instance appearing in the encoding list.

3. Retrieve the appropriate resource bundle from **shellclient** using the **getLanguageBundle** method:

   ```
   languageBundle = shellutils.getLanguageBundle ('langNetwork',
   shellUtils.osLanguage, Framework)
   ```

4. Retrieve a keyword from the resource bundle, suitable for a particular command:

   ```
   strWindowsIPAddress = languageBundle.getString('windows_ipconfig_str_ip_
   address')
   ```

5. Invoke the **executeCommandAndDecode** method and pass the keyword to it on the **ShellUtils** object:

   ```
   ipconfigOutput = shellUtils.executeCommandAndDecode('ipconfig /all',
   strWindowsIPAddress)
   ```

   The ShellUtils object is also needed to link a user to the API reference (where this method is described in detail).

6. Parse the output as usual.

Decode Commands Without a Keyword

The current approach for localization uses a keyword to decode all of the command output. For details, see the step about retrieving a keyword from the resource bundle in "Define a New Job to Operate With Localized Data" on page 60.

However, another approach uses a keyword to decode the first command output only, and then decodes further commands with the character set used to decode the first command. To do this, you use the **getCharsetName** and **useCharset** methods of the **ShellUtils** object.

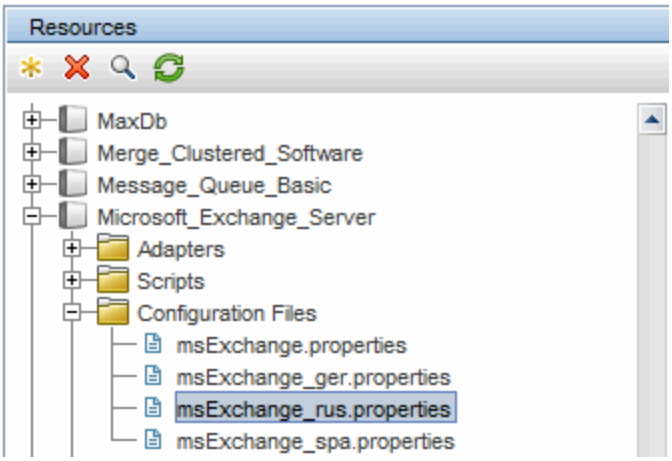
The regular use case works as follows:

1. Invoke the **executeCommandAndDecode** method once.
2. Obtain the most recently used character set name through the **getCharsetName** method.
3. Make **shellUtils** use this character set by default, by invoking the **useCharset** method on the **ShellUtils** object.
4. Invoke the **execCmd** method of **ShellUtils** one or more times. The output is returned with the character set specified in the previous step. No additional decoding operations occur.

Work with Resource Bundles

A resource bundle is a file that takes a properties extension (***.properties**). A properties file can be considered a dictionary that stores data in the format of `key = value`. Each row in a properties file contains one `key = value` association. The main functionality of a resource bundle is to return a value by its key.

Resource bundles are located on the Probe machine: **C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryConfigFiles**. They are downloaded from the UCMDB Server as any other configuration file. They can be edited, added, or removed, in the Resources window. For details, see Configuration File Pane in the *HP Universal CMDB Data Flow Management Guide*.



When discovering a destination, DFM usually needs to parse text from command output or file content. This parsing is often based on a regular expression. Different languages require different regular expressions to be used for parsing. For code to be written once for all languages, all language-specific data must be extracted to resource bundles. There is a resource bundle for each language. (Although it is possible for a resource bundle to contain data for different languages, in DFM each resource bundle contains data for one language only.)

The Jython script itself does not include hard coded, language-specific data (for example, language-specific regular expressions). The script determines the language of the remote system, loads the proper resource bundle, and obtains all language-specific data by a specific key.

In DFM, resource bundles take a specific name format: `<base_name>_<language_identifier>.properties`, for example, `langNetwork_spa.properties`. (The default resource bundle takes the following format: `<base_name>.properties`, for example, `langNetwork.properties`.)

The `base_name` format reflects the intended purpose of this bundle. For example, **langMsCluster** means the resource bundle contains language-specific resources used by the MS Cluster jobs.

The `language_identifier` format is a 3-letter acronym used to identify the language. For example, `rus` stands for the Russian language and `ger` for the German language. This language identifier is included in the declaration of the `Language` object.

API Reference

This section includes:

- "The getCharsetName Method" on the next page
- "The useCharset Method" on the next page
- "The getLanguageBundle Method" on the next page
- "The osLanguage Field" on page 66

The Language Class

This class encapsulates information about the language, such as resource bundle postfix, possible encoding, and so on.

Fields

Name	Description
locale	Java object which represents locale.
bundlePostfix	Resource bundle postfix. This postfix is used in resource bundle file names to identify the language. For example, the **langNetwork_ger.properties** bundle includes a **ger** bundle postfix.
charsets	Character sets used to encode this language. Each language can have several character sets. For example, the Russian language is commonly encoded with the `Cp866` and `Windows-1251` encoding.
wmiCodes	The list of WMI codes used by the Microsoft Windows OS to identify the language. All possible codes are listed at http://msdn.microsoft.com/en-us/library/aa394239(VS.85).aspx (the OSLanguage section). One of the methods for identifying the OS language is to query the WMI class OS for the `OSLanguage` property.
codepage	Code page used with a specific language. For example, `866` is used for Russian machines and `437` for English machines. One of the methods for identifying the OS language is to retrieve its default codepage (for example, by the `chcp` command).

The executeCommandAndDecode Method

This method is intended to be used by business logic Jython scripts. It encapsulates the decoding operation and returns a decoded command output.

Arguments

Name	Description
cmd	The actual command to be executed.
keyword	The keyword to be used for the decoding operation.
framework	The Framework object passed to every executable Jython script in DFM.
timeout	The command timeout.
waitForTimeout	Specifies if client should wait when timeout is exceeded.
useSudo	Specifies if `sudo` should be used (relevant only for UNIX machine clients).
language	Enables specifying the language directly instead of automatically detecting a language.

The getCharsetName Method

This method returns the name of the most recently used character set.

The useCharset Method

This method sets the character set on the `ShellUtils` instance, which uses this character set for initial data decoding.

Arguments

Name	Description
charsetName	The name of the character set, for example, `windows-1251` or `UTF-8`.

See also "The getCharsetName Method" above.

The getLanguageBundle Method

This method should be used to obtain the correct resource bundle. This replaces the following API:

```
Framework.getEnvironmentInformation().getBundle(...)
```

Arguments

Name	Description
baseName	The name of the bundle without the language suffix, for example, `langNetwork`.

Name	Description
language	The language object. The `ShellUtils.osLanguage` should be passed here.
framework	The Framework, common object which is passed to every executable Jython script in DFM.

The osLanguage Field

This field contains an object that represents the language.

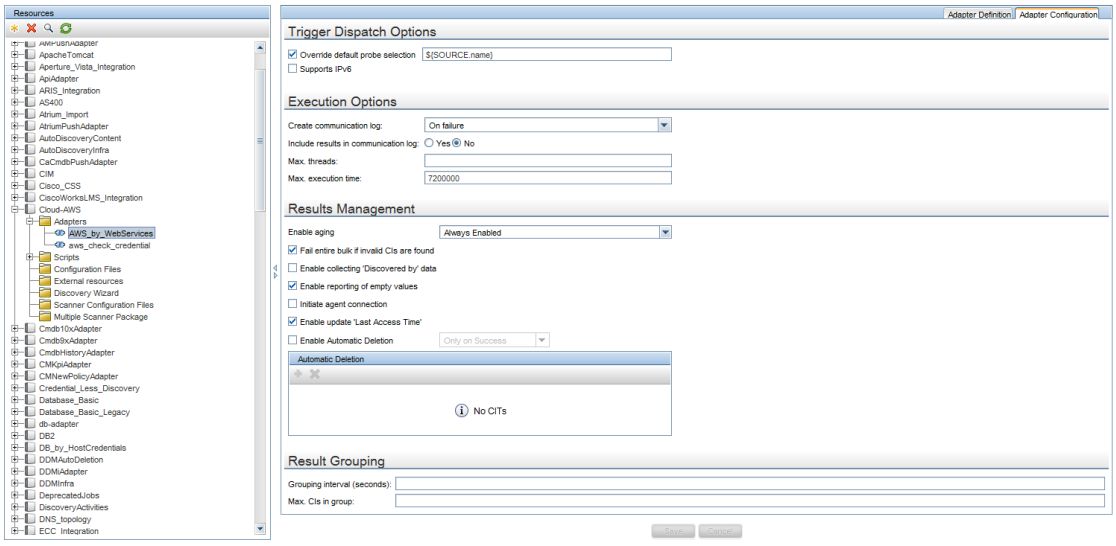
Record DFM Code

It can be very useful to record an entire execution, including all parameters, for example, when debugging and testing code. This task describes how to record an entire execution with all relevant variables. Furthermore, you can view extra debug information that is usually not printed to log files even at the debug level.

To record DFM code:

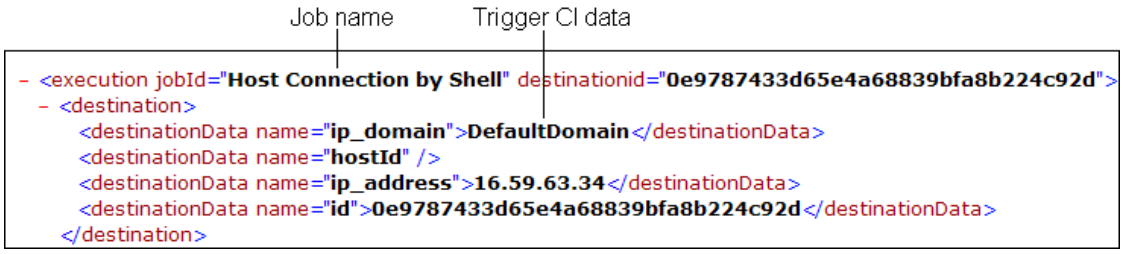
1. Access **Data Flow Management > Universal Discovery**. Right-click the job whose run must be logged and select **Go to Adapter** to open the Adapter Management application.

2. Locate the **Execution Options** pane in the **Adapter Configuration** tab, as shown below.

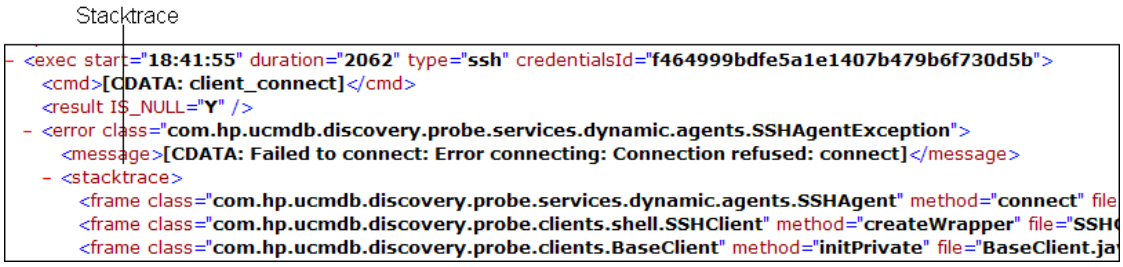


3. Change the **Create communication log** box to **Always**. For details on setting logging options, see Execution Options Pane in the *HP Universal CMDB Data Flow Management Guide*.

The following example is the XML log file that is created when the **Host Connection by Shell** job runs and the **Create communication logs** box is set to **Always** or **On Failure**:



The following example shows the message and stacktrace parameters:



Jython Libraries and Utilities

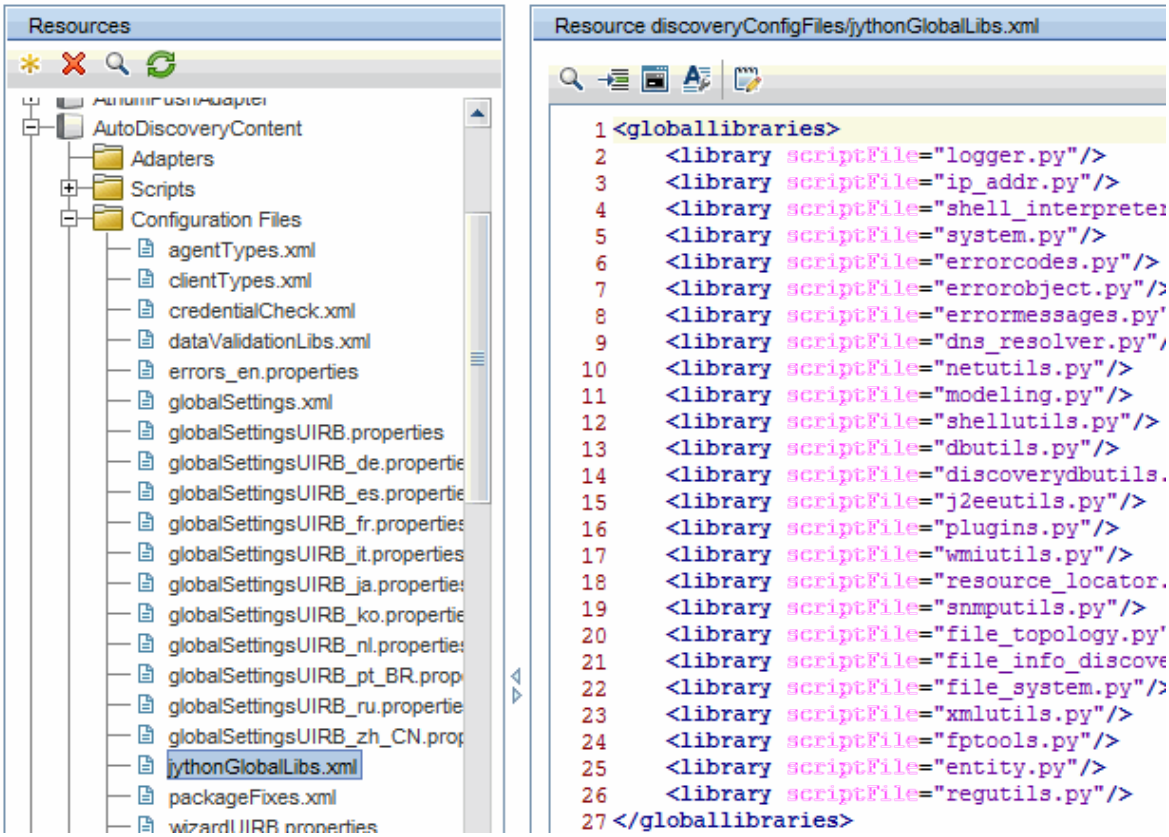
Several utility scripts are used widely in adapters. These scripts are part of the `AutoDiscovery` package and are located under: **C:\hp\UCMDB\DataFlowProbe\runtime\probeManager\discoveryScripts** with the other scripts that are downloaded to the Probe.

> **Note:** The `discoveryScript` folder is created dynamically when the Probe begins working.

To use one of the utility scripts, add the following import line to the import section of the script:

```
import <script name>
```

The AutoDiscovery Python library contains Jython utility scripts. These library scripts are considered DFM's external library. They are defined in the `jythonGlobalLibs.xml` file (located in the **Configuration Files** folder).



Each script that appears in the `jythonGlobalLibs.xml` file is loaded by default at Probe startup, so there is no need to use them explicitly in the adapter definition.

This section includes the following topics:

logger.py

The **logger.py** script contains log utilities and helper functions for error reporting. You can call its debug, info, and error APIs to write to the log files. Log messages are recorded in **C:\hp\UCMDB\DataFlowProbe\runtime\log**.

Messages are entered in the log file according to the debug level defined for the PATTERNS_DEBUG appender in the **C:\hp\UCMDB\DataFlowProbe\conf\log\probeMgrLog4j.properties** file. (By default, the level is DEBUG.) For details, see "Error Severity Levels" on page 75.

```
#########################################################################
################         PATTERNS_DEBUG log
####################
#########################################################################
log4j.category.PATTERNS_DEBUG=DEBUG, PATTERNS_DEBUG
log4j.appender.PATTERNS_DEBUG=org.apache.log4j.RollingFileAppender
log4j.appender.PATTERNS_
DEBUG.File=C:\hp\UCMDB\DataFlowProbe\runtime\log/probeMgr-patternsDebug.log
log4j.appender.PATTERNS_DEBUG.Append=true
log4j.appender.PATTERNS_DEBUG.MaxFileSize=15MB
log4j.appender.PATTERNS_DEBUG.Threshold=DEBUG
log4j.appender.PATTERNS_DEBUG.MaxBackupIndex=10
log4j.appender.PATTERNS_DEBUG.layout=org.apache.log4j.PatternLayout
log4j.appender.PATTERNS_DEBUG.layout.ConversionPattern=<%d> [%-5p] [%t] - %m%n
log4j.appender.PATTERNS_DEBUG.encoding=UTF-8
```

The info and error messages also appear in the Command Prompt console.

There are two sets of APIs:

- `logger.<debug/info/warn/error>`
- `logger.<debugException/infoException/warnException/errorException>`

The first set issues the concatenation of all its string arguments at the appropriate log level and the second set issues the concatenation as well as issuing the stack trace of the most recently-thrown exception, to provide more information, for example:

```
logger.debug('found the result')
logger.errorException('Error in discovery')
```

modeling.py

The **modeling.py** script contains APIs for creating hosts, IPs, process CIs, and so on. These APIs enable the creation of common objects and make the code more readable. For example:

```
ipOSH= modeling.createIpOSH(ip)
host = modeling.createHostOSH(ip_address)
member1 = modeling.createLinkOSH('member', ipOSH, networkOSH)
```

netutils.py

The **netutils.py** library is used to retrieve network and TCP information, such as retrieving operating

system names, checking if a MAC address is valid, checking if an IP address is valid, and so on. For example:

```
dnsName = netutils.getHostName(ip, ip)
isValidIp = netutils.isValidIp(ip_address)
address = netutils.getHostAddress(hostName)
```

shellutils.py

The **shellutils.py** library provides an API for executing shell commands and retrieving the end status of an executed command, and enables running multiple commands based on that end status. The library is initialized with a Shell Client, and uses the client to run commands and retrieve results. For example:

```
ttyClient = Framework.createClient(Framework.getTriggerCIData
(BaseClient.CREDENTIALS_ID), Props)
clientShUtils = shellutils.ShellUtils(ttyClient)
if (clientShUtils.isWinOs()):
        logger.debug ('discovering Windows..')
```

Chapter 3: Error Messages

This chapter includes:

Error Messages Overview

During discovery, many errors may be uncovered, for example, connection failures, hardware problems, exceptions, time-outs, and so on. These errors are displayed in the Universal Discovery window whenever the regular discovery flow does not succeed. You can drill down from the Trigger CI that caused the problem to view the error message itself.

DFM differentiates between errors that can sometimes be ignored (for example, an unreachable host) and errors that must be dealt with (for example, credentials problems or missing configuration or DLL files). Moreover, DFM reports errors once, even if the same error occurs on successive runs, and reports an error even it if occurs once only.

When creating a package, you can add appropriate messages as resources to the package. During package deployment, the messages are also deployed in the correct location. Messages must conform to conventions, as described in "Error-Writing Conventions" below.

DFM supports multi-language error messages. You can localize the messages you write so that they appear in the local language.

For details on searching for errors, see "Discovery Progress and Results" in the *HP Universal CMDB Data Flow Management Guide*.

For details on setting communication logs, see "Execution Options Pane" in the *HP Universal CMDB Data Flow Management Guide*.

Error-Writing Conventions

- Each error is identified by an error message code and an array of arguments (**int, String[]**). A combination of a message code and an array of arguments defines a specific error. The array of

parameters can be null.

- Each error code is mapped to a **short message** which is a fixed string and a **detailed message** which is a template string contains zero or more arguments. Matching is assumed between the number of arguments in the template and the actual number of parameters.

Example of Error Message Code:

10234 may represent an error with the short message:

```
Connection Error
```

and the detailed message:

```
Could not connect via {0} protocol due to timeout of {1} msec
```

where

{0} = the first argument: a protocol name

{1} = the second argument: the timeout length in msec

This section also includes the following topics:

- "Property File Content" below
- "Error Messages Property File" on the next page
- "Locale Naming Conventions" on the next page
- "Error Message Codes" on the next page
- "Unclassified Content Errors" on page 74
- "Changes in Framework" on page 74

Property File Content

A property file should contain two keys for each error message code. For example, for error 45:

- **DDM_ERROR_MESSAGE_SHORT_45**. Short error description.
- **DDM_ERROR_MESSAGE_LONG_45**. Long error description (can contain parameters, for example, {**0**}, {**1**}).

Error Messages Property File

A property file contains a map between an error message code and two messages (short and detailed).

Once a property file is deployed, its data is merged with existing data, that is, new message codes are added while old message codes are overridden.

Infrastructure property files are part of the **AutoDiscoveryInfra** package.

Locale Naming Conventions

- For the default locale: **<file name>.properties.errors**
- For a specific locale: **<file name>_xx.properties.errors**

  where **xx** is the locale (for example, **infraerr_fr.properties.errors** or **infraerr_en_us.properties.errors**).

Error Message Codes

The following error codes are included by default with HP Universal CMDB. You can add your own error messages to this list.

Error Name	Error Code	Description
Internal	100-199	Mostly resolved from exceptions thrown during Jython script runs
Connection	200-299	Connection failed, no agent on target machine, destination unreachable, and so on
Credential Related	300-399	Permission denied, connection attempt blocked due to a lack of credentials
Timeout	400-499	Time-out during connection/command
Unexpected or Invalid Behavior	500-599	Missing configuration files, unexpected interruptions, and so on
Information Retrieval	600-699	Missing information on target machines, failure querying agent for information, and so on

Error Name	Error Code	Description
Resources Related	700-799	Errors relating to out-of-memory or clients not released properly
Parsing	800-899	Error parsing text
Encoding	900	Error in input, unsupported encoding
SQL Related	901-903, 924	Errors received from SQL operations
HTTP Related	904-909	Errors generated during HTTP connections, parsed from HTTP error codes.
Specific Application	910-923	Error reported due to application-specific problems, for example, wrong LSOF version, No Queue Managers found, and so on

Unclassified Content Errors

To support old content without causing a regression, the application and SDK relevant methods handle errors of message code 100 (that is, unclassified script error) differently.

These errors are not grouped (that is, they are not considered as being errors of the same type) by their message code but are grouped by the content of the message. That is, if a script reports an error by the old, deprecated methods (with a message string and without an error code), all messages receive the same error code, but in the application or in the SDK relevant methods, different messages are displayed as different errors.

Changes in Framework

(com.hp.ucmdb.discovery.library.execution.BaseFramework)

The following methods are added to the interface:

- `void reportError(int msgCode, String[] params);`
- `void reportWarning(int msgCode, String[] params);`
- `void reportFatal(int msgCode, String[] params);`

The following old methods are still supported for backward compatibility purposes but are marked as deprecated:

- `void reportError(String message);`
- `void reportWarning (String message);`
- `void reportFatal (String message);`

Error Severity Levels

When an adapter finishes running against a trigger CI, it returns a status. If no error or warning is reported, the status is **Success**.

Severity levels are listed here from the narrowest to widest scope:

Fatal Errors

This level reports serious errors such as a problem with the infrastructure, missing DLL files, or exceptions:

- Failed generating the task (Probe is not found, variables are not found, and so on)
- It is not possible to run the script
- Processing of the results fails on the Server and the data is not written to the CMDB

Errors

This level reports problems that cause DFM not to retrieve data. Look through these errors as they usually require some action to be taken (for example, increase time-out, change a range, change a parameter, add another user credential, and so on).

- In cases where user intervention may help, an error is reported, either a credentials or network problem that may need further investigation. (These are not errors in discovery but in configuration.)
- Internal failure, usually because of unexpected behavior from the discovered machine or application, for example, missing configuration files, and so on

Warnings

When a run is successful but there may be non-serious problems that you should be aware of, DFM marks the severity as **Warning**. You should look at these CIs to see whether data is missing before beginning a more detailed debugging session. **Warning** can include messages about the lack of an installed agent on a remote host, or that invalid data caused an attribute to be calculated incorrectly.

- Missing connection agent (SNMP, WMI)
- Discovery succeeds, but not all available information is discovered

Chapter 4: Mapping Consumer-Provider Dependencies

This chapter includes:

Dependency Discovery Overview

Dependency mapping provides a flexible, method of discovering relationships between deployable components or running software. This method allows the use of user-defined dependency mapping rules (using simple programming syntax), which the Universal Discovery process uses to automatically discover dependencies.

A service can be either a business or IT service. A business service is a service that a business provides to another business (B2B) or that one organization provides to another within a business (such as payment processing). An IT service is a business service that an IT organization provides to support business services or IT's own operations.

A deployable component is a software component that is deployed within running software, such as an application server or web server. Examples of deployable components are JEE EAR components or a schema within an Oracle database. For the purpose of dependency discovery, running software is considered to be a deployable component.

A **provider** deployable component delivers a service, and declares how other deployable components can consume that service. A **consumer** deployable component "consumes" a service provided by a provider deployable component. The dependency between these deployable components is a **consumer-provider** dependency.

Note:

- To have the ability to create consumer-provider dependency adapters and use the Dependency Mapping framework, you must select the **Enable Search** option when setting up the UCMDB schema during installation.

- Consumer-provider dependency adapters can only be executed on Data Flow probes in union mode.

For more information, see:

- "Providers and Consumers" below
- "Dependency Signatures" on the next page
- "Dependency Mapping Flow" on the next page

Providers and Consumers

You connect to providers using connection strings. For example, if an Oracle database is a provider, to connect to its services, you might need:

- the IP address of the machine
- the SID
- the TCP port

These three pieces of information would comprise the connection strings required by a consumer, which are needed to connect to a service offered by that provider. For example, an Oracle connection string could contain the following information:

- IP addresses: 1.1.1.1, 2.2.2.2
- Port: 1521
- SID: abcd

A consumer is aware of at least one connection string for a provider, and this connection string is found in a known location, such as a configuration document, a database table, Windows registry, and so on. By searching through these locations, dependencies between consumers and providers can be discovered.

If the connection strings of a provider are found in a certain configuration document, then the provider and the container of the configuration document are connected with a consumer-provider relationship.

The process of discovering consumer-provider dependencies then becomes straightforward: Connection strings from the provider are searched for in the consumer's configuration documents, and

the search results contain all configuration documents owned by the consumers of the specified provider.

For more information, see "Define Dependencies" on page 87.

Dependency Signatures

A different search term can be used for each configuration document and provider type. These search terms are defined in a dependency signature file.

A dependency signature is a rule that defines whether or not a consumer-provider dependency exists between a given provider and a given consumer, using the connection string of the provider and the configuration documents of the consumer.

A dependency signature is composed of search expressions. These search expressions are dependent on the definition of the connection strings, and not on the actual values for a specific provider. In addition, the search expressions are dependent on the consumer's configuration document's name, location, and format, but not on the actual content of the files. When the connection strings of the provider are known, they are injected into the search expressions and this results in concrete search expressions. The concrete search expressions are then evaluated using the consumer's configuration documents. The search expressions return "True" only if a provider's connection strings exist, in some specific way, in the consumer’s configuration documents.

For more information, see "Dependency Signature Files" on the next page.

Dependency Mapping Flow

This section provides a brief overview of the basic flow that occurs during dependency mapping:

1. The deployable components and their connection strings are discovered.
2. Each type of provider deployable component triggers a specific dependency mapping job. Each job's adapter knows how to extract the relevant connection string for the specific deployable component type. The adapter searches for other deployable components that consume the service.
3. A consumer-provider relationship is created between each deployable component that is found and its trigger (provider).

Dependency Signature Files

This section includes:

Structure of a Dependency Signature File

A dependency signature file defines one or more dependencies between deployable components.

The consumer part of the dependency is defined as the <Deployable> element. Each <Deployable> element may contain one or more <Dependency> elements. Each such <Dependency> element contains the conditions for which a dependency exists between a consumer (a <Deployable> element) and a provider.

The <Dependency> element can be viewed as a Boolean function that takes as its input the provider CI and the consumer's configuration documents, and returns "True" only if a dependency exists between those two CIs, from the consumer to the provider.

In the <Dependency> element, the provider is only identified in the file using its CI type. Each dependency may be used for a single CI type of a provider. The following example defines a dependency function between any consumer that is a Running Software and a provider that is a Running Software:

```
<Deployable name="ApolloOnNod">
   <Descriptor cit="running_software">
   </Descriptor>
   <Dependency name="history_db" providerCiType="running_software"
scope="default">
      …
```

Variables and Concepts

A variable can contain different values during the execution of the program. In the case of dependency mapping, a variable can contain different values for different executions of a dependency search. These variables are used when defining search expressions to determine whether or not a connection string exists in a configuration document. Since the connection strings are different for from one provider to another, variables allow you to define generic search expressions, regardless of the specific values of the connection strings.

Variables

Note: The term "variable" denotes both variables and concept variables.

The syntax to use a variable is `${VARIABLE_NAME}`. Each variable value must either be a string value or a string list. For example, part of the connection string might be the IP address (or addresses) of the provider. To search for the IP address in the configuration document, you can define a variable named IP_ADDRESS and use it in the expression in this way: `${IP_ADDRESS}`.

Variables must first be defined, either on the scope of an entire dependency signature file (referred to as a global scope), or on the scope of a dependency (referred to as local scope). The definition of the variable enables you to use this variable.

Variables that are used but are not defined generate an error when you deploy the signature file.

Global Scope

A global scope variable can be used by any dependency in the file where it is defined. To define a global scope variable:

```
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
   <VariableDeclarations>
      <Variable name="IPADDRESS"/>
      <Variable name="PROTOCOL"/>
</DependencySignatures>
```

The above defines two global scope variables, IPADDRESS and PROTOCOL. Like all variables, they can hold any string value or string list.

Values of global variables can be set only by a trigger's destination data.

Local Scope

A local scope variable can only be used by the dependency for which it is defined. It is not visible to any other dependency, whether or not for the same deployable component.

You can define multiple local scope variables with the same name in different dependencies. Each of these is a completely separate variable, and its value(s) can only be used in the scope in which it is defined.

Note: You may not define a local scope variable to have the same name as a global scope variable, or define two local scope variables with the same name in the context of the same dependency.

To define a local scope variable, use the following syntax:

```
<Deployable name="StrongXmlApplication">
   <Descriptor cit="cluster_software"/>
   <Dependency name="app_cluster" providerCiType="running_software"
scope="default">
      <VariableDeclarations>
         <Variable name="IPADDRESS"/>
         <Variable name="PROTOCOL"/>
      </VariableDeclarations>
      …
   </Dependency>
</Deployable>
```

Local scope variables can only be assigned values by using injection statements. There are different, specific injection statements, according to the type of file from which the value is extracted. An injection statement can appear in two places in a configuration document.

- In a dedicated `<Variables>` section. The injection statements in the `<Variables>` section are only evaluated if the entire file's search expression was evaluated to True.
- In a `<Variables>` section as part of a search condition. With this option, the variable will only be injected if the search condition was evaluated to True. This is not supported when using alternative expressions. For more information, see "Using Default Values of Variables" on page 90.

For more information about assigning variables, see:

- "Properties Configuration Documents" on page 99
- "XML Configuration Documents" on page 103
- "Text Configuration Documents" on page 107

Concepts

Search expressions are applied on each relevant configuration document. Some values in the connection strings are tightly coupled with one another, and should only be used with each other in concrete search expressions. Each such set of coupled connection strings must have a unique name and is referred to as a concept.

For example, a provider deployable component may be accessed from either `1.1.1.1:8080` or `2.2.2.2:85`. It is clear that any consumer of the provider must keep, in one of its configuration-files, either either `1.1.1.1:8080` or `2.2.2.2:85`. However, you are unlikely to find `1.1.1.1:85` in a

configuration document of any consumer of this provider, as this provider does not listen on `1.1.1.1:85`. Even if `1.1.1.1:85` is found in one of the configuration documents of a consumer, it does not represent a dependency with this provider, but instead a dependency with some other deployable component that is running on the same node as the provider and listening on `1.1.1.1:85`.

The search must look for correct match of IP address and port, because otherwise the search would return false positive dependencies.

In addition, the search must also look for correct matches of the name of each IP address and port, as each IP address may have multiple names (for example, one authoritative DNS name and multiple alias names).

Therefore, it is likely to find the following matches or pairs in the configuration documents of consumers of that provider: `XYZ:8080`, `FOO:8080`, or `ABC:85` (where `XYZ`, `FOO`, and `ABC` are other names for the provider's address), as these matches represent dependencies with that provider. However, the match `ABC:8080` does not represent a dependency with that provider, and therefore such a match should not be searched.

To this end, concepts are used. Each connection string attribute that should be used in conjunction with another should be defined in the same concept. Each connection string attribute should be a variable in that concept.

Concepts can only be defined on a global scope. Each concept instance, representing a set of tightly coupled connection strings, must have its value set by the adapter, similarly to variables.

Each concept is comprised of exactly one key variable, and addition variables. Each of these variables (key variables and additional variables) are used to store the tightly coupled connection strings. The key variable is used to distinguish between different instances of the same concept. This means that there cannot be two concept instances from the same concept that have the same value in their key variable. For more information, see "Specify Concept Variable Values" on page 120.

Note: A key variable can be used like any other concept variable. Its importance is in the way concept instances are instantiated during a trigger's execution.

To define a concepts and its variables, use the following syntax:

```
<Concept name="ConceptName">
   <Properties>
      <KeyProperty name="KeyVariableName"/>
      <Property name="VariableName1"/>
      <Property name="VariableName2"/>
   </Properties>
```

```
</Concept>
```

To use a concept's variable in a search expression, use the following syntax:

```
${ConceptName.VariableName}.
```

For each concept used in a search expression, there must be a logical operator that contains all of the concept's relevant variables, and does not contain variables of other concepts.

Following are examples of valid search expressions containing concepts:

```
${C.A} = X AND ${X.B} = Y

(${C1.A} = X AND ${C1.B} = Y) OR ${C2.C} = Z
```

This is an example of an invalid search expression:

```
${C1.A} = X AND ${C2.B} = Y AND ${C2.X} = Z
```

Default Values

Global variables and concept variables can, optionally, have a default value. If a variable is injected from the adapter with a value which is the same as the default value, then it is possible to define alternative search expressions. For more information on alternative search expressions, see "Composing a Search Expression" on page 87.

Use the following syntax to define a default value:

```
<Variable name=”PORT” defaultValue=”8080” />
```

IP Address Variable Types

IP address is an important type of connection string; however, the address of the provider may not always be stated in the configuration files.

A common example of this is when the consumer and the provider are located on the same host machine. In this case, instead of the address of the provider, strings such as “localhost” or “127.0.0.1” might appear, or no address would appear at all.

By marking a variable with the type **IP Address**, the framework will automatically try to find local dependencies by ignoring the IP address variables and finding providers with the remainder of the connection strings that are located on the same host.

To mark a variable as an IP address

- On a global variable

```
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
   <VariableDeclarations>
      <Variable name="MY_VARIABLE" type=”IP Address” />
   </VariableDeclarations>
         …
</DependencySignatures>
```

- On a concept variable

```
<Concept name="IpEndpoint">
   <Properties>
      <KeyProperty name="PORT"/>
      <Property name="MY_VARIABLE" type=”IP Address” />
   </Properties>
</Concept>
```

Note: Local variables cannot have an IP Address type.

Define the Descriptor of a Consumer

To define a consumer-provider dependency, the search adapter looks for consumers with configuration documents that use the provider's connection string. However, sometimes it is meaningful to limit the deployable components that may be a consumer for a particular service, even if the consumer's configuration documents contain the connection string or have different output variables depending on the deployable component's properties.

To this end, it is possible to describe the deployable component in greater detail than just its CI type, by using the `<Descriptor>` element. The deployable component can be described using:

- CI Type, that is, the dependencies are only relevant for deployable component of type “J2EE Application”. (Mandatory)
- Conditions on its string attributes; for example, on deployable components of type “J2EE Application” where the application name is “MyShop”. The condition can only test for equality. (Optional)

- A mandatory composition link to another CI. If the composition link is specified in the descriptor, then the deployable component must have a composition link to a CI of the given type. (Optional)
- Conditions on string attributes of the connected CI, if one was specified (as in the previous option). (Optional)
- A path (TQL query) to the node containing the deployable component. The query name is specified with the attribute **nodeToDeployableQuery**. If the node is connected directly to the deployable component with a composition link, there is no need to mention a TQL query as the path. However, if the node is still used to describe the deployable component (with or without conditions), you must still mention it using the `<ConnectedCICondition>` tag. If the deployable component does not have a containing node anywhere in its parent hierarchy, mention this using the attribute `hasContainingNode = "false"`. When you mention the path, be sure to also add `hasContainingNode = "true"`. For more information, see "Define TQL Queries" on page 112. (Optional)

Note: Only the STRING attribute type is supported on deployable descriptors or connected CIs. The attribute cannot be static and cannot contain calculated attributes.

Example of a descriptor that only mentions the CI type:

```
<Deployable name="ApolloOnNode_2">
   <Descriptor cit="running_software">
   </Descriptor>
      …
```

Example of a descriptor with a string attribute:

```
<Deployable name="ApolloOnCluster">
   <Descriptor cit="node">
      <Attribute name="default_gateway_ip_address_type" value="IPv6" />
   </Descriptor>
      …
```

Example of a descriptor with a connected CI by testing case-insensitive equality:

```
<Deployable name="MyRunningSoftware">
   <Descriptor cit="running_software">
      <ConnectedCiCondition cit=”node” linkType=”composition”
isDirectionForward=”true”>
         <Attribute name=”name” value=”MyNode” operator=”equalIgnoreCase” />
      </ConnectedCiCondition>
   </Descriptor>
```

```
        ...
```

Note: For ConnectedCiCondition, you may only use linkType=”composition” and isDirectionForward=”true” .

Define Dependencies

For every consumer, you can define multiple dependencies. Each dependency is associated with a specific provider CI type. During the evaluation of the dependency signature for a provider CI, all dependencies that are associated with the provider's CI type are evaluated. Each dependency has a search expression from one or more configuration documents. If the search expression evaluates to True, there is a dependency (a consumer-provider relationship) between the consumer and the provider. Even if multiple dependencies between the same consumer and provider CI type exists and evaluate to True, only one relationship will be created.

An example of dependency syntax is:

```
<Deployable name="Websphere J2EE Application">
   <Descriptor cit="j2eeapplication"/>
   <Dependency name="J2EE Application to DB by JNDI" providerCiType="oracle"
scope="default">
      ...
```

Every dependency also has a scope. A scope is those consumers, out of all the consumers that follow the descriptor, that are relevant for this specific dependency. For more information, see "Define the Descriptor of a Consumer" on page 85.

Composing a Search Expression

Note: Only the STRING attribute type is supported on deployable descriptors or connected CIs. The attribute cannot be static and cannot contain calculated attributes.

A search expression is composed from logical operators and conditions. Supported logical operators are “And” and “Or”.

Conditions are expressions that are evaluated using a provider’s connection strings and a consumer’s configuration documents. Conditions differ according to the type of the file, that is, properties configuration documents have different conditions than XML documents.

Here is an example of a search expression for a properties configuration document: To create a dependency between a provider and consumer, only if the provider's IP address exists in the consumer's **MyConfig.properties** configuration document under the key IPADDRESS. This would be written in the dependency configuration document as:

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and name of
the configuration document)
   <Condition> (Beginning of the search expression)
      <Operator type="and"> (Operator)
         <KeyCondition key="IPADDRESS"> (Start a condition and state its type)
            <Values>
               <Value>${IP_ADDRESS}</Value> (Use a variable stating the
provider's IP address)
            </Values>
         </KeyCondition>
      </Operator>
   </Condition>
</PropertiesConfigurationDocument>
```

Note: There must be at least one `<Operator>` element in a `<Condition>` element, even if it is logically unnecessary, since there is only one search condition, as in the example above.

Let's look at a more complicated search expression: Either the provider's IP address exists in the consumer's **MyConfig.properties** configuration document under the key IPADDRESS, or the provider's host name exists in the same file under the key HOST:

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and Name of
the configuration document)
   <Condition> (Beginning of the search expression)
      <Operator type="or"> (Operator)
         <KeyCondition key="IPADDRESS"> (Start a condition and state its type)
            <Values>
               <Value>${IP_ADDRESS}</Value> (Use a variable stating the
provider's IP address)
            </Values>
         </KeyCondition>
         <KeyCondition key="HOST"> (Start another condition, under the same
operator)
            <Values>
               <Value>${HOSTNAME}</Value> (Use a variable stating the provider's
host name)
            </Values>
         </KeyCondition>
      </Operator>
   </Condition>
```

```
</PropertiesConfigurationDocument>
```

The logical operators can be nested to create conditions such as `(C1 AND (C2 OR C3))`, so the XML would look like this:

```
<PropertiesConfigurationDocument name="MyConfig.properties"> (Type and Name of
the configuration document)
   <Condition>
      <Operator type="and">
         C1
         <Operator type=”or”>
            C2
            C3
         </Operator>
      </Operator>
   </Condition>
</PropertiesConfigurationDocument>
```

where `C1`, `C2`, `C3` are the XML snippets of the required search conditions.

There are three supported type of files, and each has its own type of search expression:

- PropertiesConfigurationDocument – a file where each line has the format of `key=value`. For more information, see "Properties Configuration Documents" on page 99.

  KeyCondition – a condition on the value of a given key

- XmlConfigurationDocument – an XML file. For more information, see "XML Configuration Documents" on page 103.

  XPathCondition – evaluates an XPath query

- TextConfigurationDocument – any text configuration document. For more information, see "Text Configuration Documents" on page 107.

  RegExp – evaluates a regular expression

For more information about default values, see "Using Default Values of Variables" on the next page.

Troubleshooting

- Having different concepts in the same node of condition tree is not permitted. Likewise, having two leaves with different concepts under the same operator is also not permitted.

For example, the following condition definition is invalid:

```
<Condition>
    <Operator type="and">
        <XPathCondition>
            <XPath>/Setup/Configuration/Hostname[matches(@Name,
'.*?${Concept1.HOSTNAME}.*')]</XPath>
        </XPathCondition>
        <XPathCondition>
            <XPath>/Setup/Configuration/Port[matches(text(),
'${Concept2.PORT}')]</XPath>
        </XPathCondition>
    </Operator>
</Condition>
```

Using Default Values of Variables

There are situations where the configuration files will not contain the value of a certain connection string if it is equal to some default value. For example, when defining an HTTP URL, a port value of 80 is probably not specifically mentioned in the URL. This means that the configuration file is more likely to contain http://www.hp.com/ than http://www.hp.com:80/, even though both are correct. This might cause a problem when writing a search condition, since if the PORT variable contains the value 80 and the condition requires such value to exist, the test for the URL will fail if the value is not specified.

To solve this, each variable can, optionally, have a default value. Search expressions can be altered in case the variable's value is the same as its default value.

There are two ways to alter a search expression:

- Ignore a condition if a variable's value is the same as its default value

  Use this option to completely ignore a condition if one or more of the variables in the search condition has the same value as its default value. If a condition is ignored, it does not return True or False, it is ignored so the result of the logical operator depends on the operator and the other operands. For example:

  - In the case of a condition having the form **A AND B**, where **A** and **B** are other expressions, if **A** is ignored, the results will be the same as the result of **B**.
  - For the expression: **A AND B AND C**, if **A** is ignored, the result will be the same as **B AND C**.
  - For the expression: **A OR B OR C**, if **A** is ignored, the result will be the same as **B OR C**.

In the rare case where all sub-conditions were ignored, meaning that the entire condition was ignored, the result of the configuration document condition will be False.

To ignore a condition, add the **ignoreIfDefaultValue** attribute to the condition, following a list of variables that will trigger this condition to be ignored. For example:

```
<KeyCondition key="serverName" ignoreIfDefaultValue="IPADDRESS">
```

- Use a different condition if a variable's value is the same as its default

Let's continue with the example with PORT 80 and the URL. Use the following regular expression to test for the URL:

```
http://${DOMAIN}:${PORT}/
```

It is clear that this regular expression will never evaluate to True for strings such as http://www.hp.com/ since the colon (:) does not exist in the string. Therefore, we would also want to test for the following regular expression if the PORT variable has the value of **80**:

```
http://${DOMAIN}/
```

In these kinds of scenarios, you can use alternative expressions. To define an alternative expression, define multiple search expressions under the same conditions, and state which one should be used when a variable has its default value. For example:

```
<KeyCondition key="url">
   <RegExp>http://${DOMAIN}:${PORT}/</RegExp>
   <RegExp alternativeFor=”PORT”>http://${DOMAIN}/</RegExp>
</KeyCondition>
```

Using the **alternativeFor** attribute, state that in case the variable has its default value, the alternative expression will be evaluated, instead of the original expression. The original expression is the one without the **alternativeFor** attribute or with an empty **alternativeFor** attribute.

For all variables that appear in an expression and have a default value, you must define alternative expression to cover any combination of those variables. Otherwise, you will get a compilation error. In case there is just one variable, you need only one alternative expression, as in the example above. If there are multiple variables with default values, you will need multiple alternative expressions.

If, for example, the DOMAIN variable also had a default value, using an **alternativeFor** statement means having all the following alternative conditions:

```
<KeyCondition key="url">
   <RegExp>http://${DOMAIN}:${PORT}/</RegExp>
   <RegExp alternativeFor=”PORT”>http://${DOMAIN}/</RegExp>
   <RegExp alternativeFor=”DOMAIN”>http://www.hp.com:${PORT}/</RegExp>
   <RegExp alternativeFor=”PORT, DOMAIN”>http://www.hp.com/</RegExp>
</KeyCondition>
```

Only one of those combinations will be evaluated during runtime.

For more information about specifying default values for variables, see "Default Values" on page 84.

Specify Paths of Configuration Documents

The path to a configuration document does not refer to the path to the file on the host's file system, but to the topological path between the consumer deployable and the configuration document.

By default, if a path is not specified, it is assumed that the configuration document is connected by a composition link to the deployable. In other words, it is assumed that the deployable component CI owns the configuration document CI.

To specify a different path:

1. Define a TQL query as specified in "Define TQL Queries" on page 112.
2. Reference the TQL query from the configuration document using the `<DocumentCILocation>` element, like this:

```
<TextConfigurationDocument name="cmdb.properties">
   <DocumentCILocation>
      <ReferenceLocation>YourQueryName</ReferenceLocation>
   <DocumentCILocation>
   <Condition>
      ...
   </Condition>
</TextConfigurationDocument>
```

In this example, YourQueryName is a reference to the query name in the `<Queries>` section.

When you build the TQL query to specify the path:

- The TQL query must define a path between the deployable component and the configuration document. This path must be simple (no cycles).

- The TQL query must include the following two end nodes with these specific and case-sensitive names:
  - Deployable – the deployable component CI in the path
  - Configuration_document – the CI in the path that specifies the configuration document
- Conditions are not permitted on the Deployable and Configuration_document nodes.
- Cardinalities between all nodes must be 1..1.
- Only a regular link type is supported. No compounds, joins, or sub-graphs are allowed.

Configuration Document Overrides

There are situations where a configuration document can override or be overridden by other configuration documents. For example, a configuration document that exists on a host might override configuration documents that exist on a cluster to which the host belongs. In such cases, values for keys must look through all of the files that might override the values, and choose the one with the highest priority. In this example, the host's local configuration document has a higher priority than the cluster document.

To define file overrides in the dependency signature, use the following syntax to add multiple paths with priority to a configuration document:

```
<PropertiesConfigurationDocument name="resources.xml">
   <DocumentCILocation>
      <ReferenceLocation priority="2">websphereas_resource_
configfiles</ReferenceLocation>
      <ReferenceLocation priority="3">j2ee_cluster_
configfiles</ReferenceLocation>
      <ReferenceLocation priority="1">j2eeapplication_
configfiles</ReferenceLocation>
   </DocumentCILocation>
   …
</PropertiesConfigurationDocument
```

The file name (**resources.xml** in the above example) must be the same in all reference locations.

Keep in mind that a reference location is a reference to a TQL query that specifies a topological path from the consumer deployable component to the configuration document. This also means that such a path must exist between the deployable component and all of the different locations. For more information about `<DocumentCILocation>` see "Specify Paths of Configuration Documents" on the previous page.

The condition itself does not change when you add multiple paths that have priorities. At runtime (when the search expression is evaluated), the correct values according to the priorities are used. For example, in properties files, if given a Key K that exists both in priority 1 and priority 2 (with a condition on its value), the condition will only be evaluated on the document with priority 1. If the key K exists only in priority 2, the condition will only be evaluated on the document with priority 2.

For more information on properties conditions, see "Properties Configuration Documents" on page 99.

In XML documents, each XPath is considered a key. For example, `\Root\Element\@Attribute` means that the attribute "Attribute" with that path is a key, and its value might be overridden in different files.

If the XPath also has some constant condition (a condition not involving variables), that condition will be part of the key. For example: `\Root\Element[@name = 'name']\@Attribute`.

However, if the XPath has a non-constant condition, those conditions are stripped from the key, and are considered to be part of the value. So for the path `\Root\Element[@name = ${NAME}]\@Attribute`, the key would be the path `\Root\Element\@Attribute` and the condition `Element[@name = 'name']` will only be evaluated on the highest priority document where that key exists.

If an XML document uses priorities, it is important to make sure the XPath conditions are built as described above, to prevent incorrect and unexpected behavior.

For more information on XPath conditions, see "XML Configuration Documents" on page 103.

Note: Multiple reference locations with priorities are only allowed on Property and XML configuration documents. They are not allowed on Text documents.

Multiple Configuration Documents with the same Priority"

It is possible to given the same priority to different `<ReferenceLocation>` elements. In this case, when evaluating a condition, the framework will look at all files with the same priority and will try and match all of them.

The condition will evaluate to True if at least a set number of files evaluates to True. The number of files is defined by the **samePriorityMatchAtLeast** attribute .

For example:

```
<PropertiesConfigurationDocument name="resources.xml">
   <DocumentCILocation samePriorityMatchAtLeast="1">
      <ReferenceLocation priority="1">websphereas_resource_
configfiles</ReferenceLocation>
      <ReferenceLocation priority="1">j2ee_cluster_
configfiles</ReferenceLocation>
```

```
      <ReferenceLocation priority="1">j2eeapplication_
configfiles</ReferenceLocation>
   </DocumentCILocation>
…
</PropertiesConfigurationDocument>
```

This means that when evaluating a condition, at least one of the referenced file locations is to a file where the condition evaluates to True.

Note: Currently, only the value of **1** is supported for the **samePriorityMatchAtLeast** attribute.

Dependencies Defined across Multiple Documents

In many cases, determining whether or not a consumer provides a service requires searching through multiple configuration documents to locate all of a provider's connection strings.

Searching through multiple unrelated configuration documents

The configuration documents might be unrelated to each other, with some of the connection strings appearing in one document and some in another document. For example, a consumer might have two configuration documents, **A.conf** and **B.conf**. The provider's connection strings are C1, which is defined in **A.conf**, and C2, which is defined in **B.conf**. Both C1 and C2 are required to determine that there is indeed a dependency between the consumer and the provider. In this case, there are two required search expressions, each in one of the documents. The <Dependency> tag would then have the following structure:

```
<Dependency name="dependency_name" providerCiType="webmodule" scope="my_scope">
   <PropertiesConfigurationDocument name="A.conf">
      <Condition>
         <Operator type="and">
            <KeyCondition key="C1">
               <Values>
                  <Value>${VAR_1}</Value>
               </Values>
            </KeyCondition>
         </Operator>
      </Condition>
   </PropertiesConfigurationDocument>
   <TextConfigurationDocument name="B.conf">
      <Condition>
         <Operator type="and">
            <RegExpCondition>
```

```
            <RegExp>C2?${VAR_2}</RegExp>
          </RegExpCondition>
        </Operator>
      </Condition>
   </TextConfigurationDocument>
</Dependency>
```

Note:

- There is no limit to the number of files that can appear in a <Dependency> element.
- Each file may be of a different type (properties, XML, or text).
- The order of the files in the <Dependency> element is ignored. The files are tested in an arbitrary order.
- The search expressions from all files must return True for the dependency to exist.
- Since the connection strings are spread across multiple files, the scope's search expression must be broad enough so that at least one of the required files can return from the scope. For more information, see "Specify Scope of Search" on page 109.

To sum up the flow when searching through multiple files:

1. The filtering scope is executed, as described in "Specify Scope of Search" on page 109. If none of the required files is returned, the dependency does not exist between the consumer and provider.
2. If at least one of the files is returned by the filter, the consumer connected to that file and the remainder of the required configurations are downloaded to the Data Flow Probe.
3. In some arbitrary order, the conditions from each file are evaluated.
4. If all conditions return True, the dependency exists; otherwise, it does not.

Searching through multiple configuration documents with dependencies

In other cases, a consumer's configuration documents are related to one another. For example, the **DBConnections.conf** file defines multiple connection strings to multiple databases and schemas, and each such connection string is give a name. In the **MyApp.conf** file, one of these connection names is used, and this defines that MyApp uses that specific database and schema.

Local scope variables are used in configuration documents with dependencies. The values for local scope variables are injected (using an injection statement) in one of the configuration documents, so that the variable value (for example, the name of a connection string) exists in that document. Then the variable can be used in the conditions of a different configuration document (for example, using the connection name in the **MyApp.conf** file to make sure that MyApp uses the provider (DB)). For more information, see "Local Scope" on page 81.

Use Variable Injection statements to inject values into a variable. There is a different statement for each document type:

- Properties files – KeyVariable and KeyRegExpVariable. For more information, see "Properties Configuration Documents" on page 99.
- XML files – XPathVariable. For more information, see "XML Configuration Documents" on page 103.
- Text files – RegExpVariable. For more information, see "Text Configuration Documents" on page 107.

In this case, there are two required search expressions, one in each of the files. So the <Dependency> tag will have the following structure:

```
<Dependency name="dependency_name" providerCiType="webmodule" scope="my_scope">
      <VariableDeclarations>
         <Variable name=”REFERENCE_NAME” />
      </VariableDeclarations>
   <PropertiesConfigurationDocument name="DBConnections.conf">
      <Condition>
         <Operator type="and">
            <KeyCondition key="C1">
               <Values>
                  <Value>${VAR_1}</Value>
               </Values>
            </KeyCondition>
         </Operator>
      </Condition>
      <Variables>
         <KeyVariable variable=”REFERENCE_NAME” key=”reference_name” />
      </Variables>
   </PropertiesConfigurationDocument>
   <TextConfigurationDocument name="MyApp.conf">
      <Condition>
         <Operator type="and">
            <RegExpCondition>
               <RegExp>C2?${REFERENCE_NAME}</RegExp>
            </RegExpCondition>
         </Operator>
      </Condition>
```

```
    </TextConfigurationDocument>
</Dependency>
```

Note:

- Use the <VariableDeclarations> element to define one or more local variables to be used in the dependency.
- The <Variables> element is used to inject values into the local variables. The section is executed only if the <Condition> element returns True.
- In this example, <KeyVariable> is used to inject the value of the "reference_name" key into the REFERENCE_NAME variable.
- The ${REFERENCE_NAME} variable is used in the **MyApp.conf** condition. This variable depends on the **DBConnections.conf** file, and it will be evaluated only after **DBConnections.conf** and only if the search expression evaluates to True.
- Cyclic dependencies between files are not permitted.
- When the **MyApp.conf** document is evaluated, the ${REFERENCE_NAME} variable already contains the value from **DBConnections.conf**.
- Since the connection strings are spread across multiple files, the scope's search expression must be broad enough so that files without dependencies can return from the scope. For more information, see "Specify Scope of Search" on page 109.

To sum up the flow when searching through multiple files with dependencies between them:

1. The filtering scope is executed, as described in "Specify Scope of Search" on page 109. If none of the required files is returned or no files are dependent on other files, the dependency does not exist between the consumer and provider.
2. At least one of the files that has no dependencies is returned by the filter.
3. The consumer connected to that file and the remainder of the required configurations are downloaded to the Data Flow Probe.

The injection statement might not return a value, for example, if the key to which it refers does not exist. In this case, the variable will get an empty value. However, this might not be the desired behavior. If the required value does not exist, it might mean the dependency should not exist, and there's no point

in continuing to evaluate the subsequent files. For this scenario use `allowNull=”false”` in the injection statement. For example:

```
 <KeyVariable variable=”REFERENCE_NAME” key=”reference_name” allowNull=”false”
/>
```

Properties Configuration Documents

Properties configuration documents store configuration in a `Key=value` format. For example, a properties configuration document might be:

```
# My configuration document
Hostname=MyNode
```

where `Hostname` is a key, and `MyNode` is its associated value.

“#” marks a commented line. Commented lines are ignored when testing the search expression.

To state that the configuration document is a properties configuration document, use the `<PropertiesConfigurationDocument>` element. For example:

```
<Dependency name="history_db" providerCiType="cmdb" scope="default">
   <PropertiesConfigurationDocument name="cmdb.conf">
      ...
```

Defining Conditions

There are several ways to test for the existence of connection string variables in properties files. For all condition types, the value of the key specified in the **key** attribute is tested against some value. The key is always a constant value (no variables are allowed):

- Test that a constant value exists as the value of a key.

  ```
  <KeyCondition key="dal.datamodel.name">
     <Values>
        <Value>ConstantValue1</Value>
        <Value>ConstantValue2</Value>
     </Values>
  </KeyCondition>
  ```

  The condition returns True only if the key’s value equals (case-insensitive) one of the constant's values.

- Test that a variable value exists as the value of a key.

```
<KeyCondition key="dal.datamodel.name">
   <Values>
      <Value>${VARIABLE1}</Value>
      <Value>${VARIABLE2}</Value>
   </Values>
</KeyCondition>
```

The condition returns True only if the key's value equals (case-insensitive) one of the variable's values. If the variable value is a list of values, then all values are tested, and a match of the key with one of the values in the list is enough to return True.

Note: You can specify both variables and constant values in the same condition.

- Test that the value of a key comply with some regular expression.

```
<KeyCondition key="dal.datamodel.name">
   <RegExp>
      ^SomeText${VARIABLE}MoreText$
   </RegExp>
</KeyCondition>
```

The regular expression can also contain one or more variables as part of the condition. During runtime, variables are replaced with their actual values when generating the concrete search expression.

In case the variable contains a list of values, an expression similar to the following will be generated:

```
^SomeText(Value1 | Value2  | Value3)MoreText$
```

In this case, either value may return True for the condition.

Example of conditions in a properties configuration document

```
<PropertiesConfigurationDocument name="MyConfig.properties">
   <Condition>
      <Operator type="and">
         <KeyCondition key="TYPE">
            <Values>
               <Value>HTTP</Value>
               <Value>HTTPS</Value>
            </Values>
```

```
        </KeyCondition>
        <KeyCondition key="IPADDRESS">
           <Values>
              <Value>${IP_ADDRESS}</Value>
              <Value>${HOSTNAME}</Value>
           </Values>
        </KeyCondition>
        <KeyCondition key="SITE">
           <RegExp>
              //${SITE_NAME}//.*
           </RegExo>
        </KeyCondition>
      </Operator>
   </Condition>
</PropertiesConfigurationDocument>
```

This means that the properties configuration document called **MyConfig.properties** must have all of the following:

- A key named TYPE contains one of the values HTTP or HTTPS.
- A key named IPADDRESS contains either the provider's IP address (or one of its IP addresses), or its hostname.
- A key named SITE that starts with a "/", continues with the provider's SITE_NAME, another "/" and then any string.

Injecting Variable Values

There are two ways to extract a key's value or part of a value into a variable:

- By injecting a key's value.
  - In a dedicated `<Variables>` section, use the following syntax:

    ```
    <KeyVariable variable="VARIABLE_NAME" key="KEY_NAME" />
    ```

  - As part of a search condition, use the following syntax:

    ```
    <KeyCondition key="KEY_NAME">
       <Values>
          <Value>REQUIRED_VALUE</Value>
       </Values>
    ```

```
   <Variables>
      <KeyVariable variable="VARIABLE_NAME" />
   </Variables>
</KeyCondition>
```

The key of the VARIABLE_NAME variable is defined in the <KeyCondition> element.

This syntax is the same if you use <RegExp> instead of <Values>. In either case, the variable will be injected with the complete value of the key.

- By injecting part of a key's value.
  - In a dedicated <Variables> section, use the following syntax:

```
<KeyRegExpVariable variable="VARIABLE_NAME" key="KEY_NAME"
expression="REGULAR_EXPRESSION" group="GROUP_NUMBER" />
```

In this case, GROUP_NUMBER is the index (starting with 0) of the group in the regular expression defined by REGULAR_EXPRESSION. For example, in a properties document containing the line

```
myKey = 123Value123
```

and the statement

```
<KeyRegExpVariable variable="VAR" key="myKey" expression="[0-9]*([a-zA-Z]
*)[0-9]*" group="0" />
```

injects the value "Value" into the variable "VAR".

You can use variables that were defined in the same file or in other files. For example:

```
<KeyRegExpVariable variable="VAR" key="myKey" expression="${PREV_VAR}([a-
zA-Z]*)${PREV_VAR}" group="0" />
```

Assuming PREV_VAR contains the value "123", VAR will get the value "Value", in the same way as injecting a key's entire value. With either option, the variable will always contain a single value.

  - As part of a search condition, use the following syntax:

```
<KeyCondition key="KEY_NAME">
   <RegExp>REGULAR_EXPRESSION</RegExp>
   <Variables>
```

```
      <KeyRegExpVariable variable="VARIABLE_NAME" group="GROUP_NUMBER" />
   </Variables>
</KeyCondition>
```

The key of the VARIABLE_NAME variable is defined in the `<KeyCondition>` element.

This is only supported when using `<RegExp>`, and the regular expression that the variable's "group" attribute refers to is the same as the one in `<RegExp>`.

XML Configuration Documents

With an XML configuration document, you can use XPath queries to easily write search expressions for configuration documents based on the XML standard.

To state that a configuration document is an XML configuration document, use the `<XmlConfigurationDocument>` element. For example:

```
<Dependency name="some_reference" providerCiType="webmodule" scope="default">
   <XmlConfigurationDocument name="web.xml">
      ...
```

Defining Conditions

The only search conditions allowed in XML configuration documents are valid XPath 2.0 queries. If a node was selected from the XPath, then the condition returns True; otherwise, it returns False. A query can contain variables in the following locations:

- In element names

  ```
  /Root/${SITE_NAME}
  ```

  If a variable contains multiple values, the framework executes multiple XPath statements. For example, if `SITE_NAME` has the values SITE1 and SITE2, then the statement in this example generates the following two concrete searches:

  ```
  \Root\SITE1
  \Root\SITE2
  ```

- In equality tests

  ```
  /Element[@att = ${VAR}]
  ```
  or

```
/Element/text() = ${VAR}
```

Equality, as shown in the above examples, works only for variables containing single values. If there is a chance that a variable may contain more than one value, use the following syntax instead:

```
/Element[${equals(@att, VAR)]
or
/Element[${equals-ignore-case(text(), VAR}]
```

Note:

- Use `${equals()}` to test for case-sensitive equality, or `${equals-ignore-case()}` for case-insensitive equality.
- The first parameter should be an XPath function or an attribute name.
- The second parameter should always be a variable name. The variable name should appear without the `${}`.
- This syntax works for variables with a single value or multiple values, and is recommended for use in all cases.
- If there are multiple values and the value of the first parameter equals just one value of the variables, the function returns True.

- In regular expressions

```
/datasources/mbean[matches(@name, '.*?,ip=${IPADDRESS}.*')]
```

If the variable contains a list of values, an expression similar to the following is generated:

```
/datasources/mbean[matches(@name, '.*?,ip=(Value1 | Value2 | Value3).*')]
```

Either value returns True for the condition.

Example of conditions in an XML configuration document

```
<XmlConfigurationDocument name="MyConfig.xml">
   <Condition>
```

```
      <Operator type="and">
         <XPathCondition>
            <XPath>\URL\Protocol[@name='HTTP' or @name='HTTPS']</XPath>
         </XPathCondition>
         <XPathCondition>
            <XPath>\URL\Host[${equals-ignore-case(@name, HOSTNAME)} or ${equals
(@name, IP_ADDRESS)}]</XPath>
         </XPathCondition >
         <XPathCondition>
            <XPath>\URL\Site[matches(text(), '//${SITE_NAME}//.*')]</XPath>
         </XPathCondition>
      </Operator>
   </Condition>
</XmlConfigurationDocument>
```

This example shows that the **MyConfig.xml** XML configuration document must have all of the following:

- `\URL\Protocol\@name` must have the value HTTP or HTTPS.
- `\URL\Host\@name` must contain either the provider's IP address (or one of its IP addresses) or its hostname.
- `\URL\Site\text()` must starts with a /, continue with the provider's SITE_NAME, contain another /, and then contain any string.

Injecting Variable Values

You can use XPath to query textual values and inject them into a variable. It is not possible to inject entire elements into a variable. To achieve this in a dedicated `<Variables>` section, use the following syntax:

```
<XPathVariable variable="VARIABLE_NAME" xpath="XPATH_QUERY" />
```

Here are a few examples:

- Selecting an attribute value

  ```
  <XPathVariable variable="VAR" xpath="\Root\Element\@Att" />
  ```

  selects the values from the **Att** attribute from all \Root\Element elements in the XML document.

- Selecting element text

```
<XPathVariable variable=”VAR” xpath=”\Root\Element\text()” />
```

selects the text of all elements matching \Root\Element .

- Selecting attribute value with conditions (using variables from the same or different configuration documents)

```
<XPathVariable variable=”VAR” xpath=”\Root\Element[${equals(@Att, VAR)}]
\@AnotherAtt” />
```

selects the value of @AnotherAtt from all \Root\Element where `@Att=${VAR}`.

You can use XPath variables as part of XPath conditions. In this mode, XPath can also contain a relative path from the result node of the condition, assuming there was a result (meaning that the condition evaluated to True). Use the following syntax:

```
<XPathCondition>
   <XPath>XPATH_QUERY</XPath>
   <Variables>
      <XPathVariable variable="VARIABLE_NAME" relativePath="RELATIVE_XPATH_
QUERY" />
   </Variables>
</XPathCondition>
```

For example:

```
<XPathCondition>
   <XPath>/${VAR_1}/chcpCodeToCharsetName[@name='test1' and matches(text(),
'.*?${OUTPUT_VAR2}.*')]</XPath>
   <Variables>
      <XPathVariable variable="OUTPUT_VAR1" relativePath="./@type" />
   </Variables>
</XPathCondition>
```

Assuming the XPathCondition evaluated to True and some XML node (specifically, an element in this case) was selected by `<XPath>`, the variable will contain that node’s “type” attribute value.

It is possible to define an XPath for a variable which is independent of that variable, by specifying an XPath from the document’s root starting with “/”.

After XPath injection, the variable may contain a list of values.

Text Configuration Documents

A text configuration document is an text document with a format that is known to the content developer, but it is not a properties configuration document or an XML configuration document. You can use regular expressions to write search expressions in text configuration documents.

To state that a configuration document is a text file type, use the `<TextConfigurationDocument>` element. For example:

```
<Dependency name="some_reference" providerCiType="oracle" scope="default">
   <TextConfigurationDocument name="tnsnames.ora">
      ...
```

Defining Conditions

The only search conditions allowed in text configuration documents are regular expressions. If the pattern matches, the condition returns True.

The regular expressions may contain one or more variables as part of a condition. Variables are replaced with their actual values during runtime, when generating the concrete search expressions.

If the variable contains a list of values, an expression similar to the following is generated:

```
^SomeText(Value1 | Value2 | Value3)MoreText$
```

Either value returns True for the condition.

Example of conditions in a text configuration document

```
<TextConfigurationDocument name="MyConfig.txt">
   <Condition>
      <Operator type="or">
         <RegExpCondition>
            <RegExp>^HTTPS?://(${HOSTNAME)/${SITE_NAME}/.*$</RegExp>
         </RegExpCondition>
         <RegExpCondition>
            <RegExp>^HTTPS?://(${IP_ADDRESS)/${SITE_NAME}/.*$</RegExp>
         </RegExpCondition>
      </Operator>
   </Condition>
</TextConfigurationDocument>
```

This example shows that the **MyConfig.txt** text configuration document must have all of the following:

- `\URL\Protocol\@name` must have the value HTTP or HTTPS.
- `\URL\Host\@name` must contain either the provider's IP address (or one of its IP addresses) or its hostname.
- `\URL\Site\text()` must starts with a /, continue with the provider's SITE_NAME, contain another /, and then contain any string.

Injecting Variable Values

You can use regular expressions with groups to inject a value into a variable for text documents. The groups mark the text that should be injected. In a dedicated `<Variables>` section, use the following syntax:

```
<RegExpVariable variable="VARIABLE_NAME" expression="REGULAR_EXPRESSION"
group="GROUP_NUMBER" />
```

In this case, GROUP_NUMBER is the index (starting with 0) of the group in the regular expression defined by REGULAR_EXPRESSION.

For example, for a file containing the line

```
This is part of a configuration document
```

and the statement

```
<RegExpVariable variable="VAR" expression="(.*)configuration (.*)" group="1" />
```

puts the value "document" into the variable VAR. You may use variables that were defined in different documents together in the same document. For example:

```
<RegExpVariable variable="VAR" expression=".*${PREV_VAR} (.*)" group="0" />
```

If PREV_VAR has the value "configuration", VAR is injected with the string "document".

You can use `<RegExpVariable>` as part of a `<RegExpCondition>` tag, by using the following syntax:

```
<RegExpCondition>
   <RegExp>REGULAR_EXPRESSION</RegExp>
   <Variables>
      <RegExpVariable variable="VARIABLE_NAME" group="GROUP_NUMBER" />
```

```
    </Variables>
</RegExpCondition>
```

The regular expressions that the variable's "group" attribute refers to is the same as the one in `<RegExp>`.

The injected variable is always injected with a single value.

Specify Scope of Search

The purpose a search's scope is to find all the consumers that may potentially be part of the dependency with a specific provider type. A scope has two uses:

- To filter out irrelevant deployable components from UCMDB, so that fewer results are searched. (Mandatory)

  To achieve this, an attempt is made to find a subset of the provider's connection strings in all configuration documents and limit the search to only the deployable components that are connected to those configuration documents.

  This search expression is different than the provider-specific dependency search, since it must return all potential consumers as quickly as possible, and not to return exact results for whether or not a consumer-provider dependency exists.

  The search expression syntax is composed in a similar way to how the configuration document search expressions are written; however, it does not have any special conditions per file type and there is no need to specify the name of the files. For more information, see "Composing a Search Expression" on page 87.

  This example represents the following expression: `(x | ((y & z & w) & (h | g)))`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ConfigurationDocumentSearchCondition xmlns="http://www.hp.com/ucmdb/1-0-
0/DependenciesDefaultSearch">
    <Operator type="Or">
        <Operator type="And">
            <Operand value="y"/>
            <Operand value="z"/>
            <Operand value="w"/>
            <Operator type="Or">
                <Operand value="h"/>
```

```
                <Operand value="g"/>
            </Operator>
        </Operator>
        <Operand value="x"/>
    </Operator>
</ConfigurationDocumentSearchCondition>
```

x, y, z, w, h and g can be constant values, variables, or concept variables.

The values (or variable values) will be searched using this expression in all relevant consumer configuration documents in UCMDB.

Only those consumers that have such files will continue on to the provider-specific search and be evaluated using the accurate search expressions.

- To limit the deployable components to a subset of a provider's connected components; for example, to only search for dependencies that are part of the provider's J2EE Domain. (Optional)

  To achieve this, a TQL query is used to find all deployable components that are connected in some way to a provider type. For more information, see "Define TQL Queries" on page 112.

  The result of the TQL query (given a specific provider CI) defines all possible deployable components that might have a dependency to the provider. The TQL query must follow these rules:

  - The query node for the deployable component must be called “Deployable” (case-sensitive). This results of this query node are the possible deployable components for the scope.
  - The “Deployable” component represents both consumer and provider components that might be in the same scope. Therefore, the query node CI type must be an ancestor for the consumer and provider deployable CI types.
  - There must be only one other query node in the TQL query (other than “Deployable”). The name of the other query node is not restricted. This node is referred to as the Scope query node.
  - The Deployable and the Scope query nodes are connected with a compound link that contains all available paths between these two nodes.

- All scopes implicitly filter out consumer deployable components that are not in the same routing domain as the provider (assuming the provider was discovered by a certain routing domain). Filtering with a routing domain requires the deployable descriptor to have a path to the node. For more information, see "Define the Descriptor of a Consumer" on page 85.

For example, if the provider deployable component is some type of running software, that running software's routing domain will be the same as its contained node. The node's routing domain is determined by the routing domain of its IP addresses. The node may be related to more than one routing domain, if its IP addresses are from different routing domains. On the other hand, if the provider deployable component is a CI that is not related to any node or IP address (such as a business service), then it can be connected to any consumer deployable component, regardless of its routing domain.

For example, to define a J2EE domain scope, do the following:

1. Create a Deployable query node of type "J2EE Deployed Object". This is the lowest common denominator class among all CI types that represent deployable components in a J2EE Domain. For example, a deployable component in this scope might be Web Module or J2EE Application.

2. Create a Scope query node of type "J2EE Domain".

3. Create a compound link between the Dependency and Scope query nodes with all paths between a J2EE Deployed Object and a J2EE Domain. For example, the compound link might contain the following:

   - J2EE Application – Composition – J2EE Domain

   - Web Module – Composition – J2EE Application

The TQL query definition must be embedded in the Scope XML.

Default Values of Variables in Scope Search

Let's use the following example:

- You are using the PORT variable, which has a default value of 80. This default value may not actually appear in some of the configuration files.

- You are using `IP_ADDRESS AND PORT` as the scope search expression, and for certain providers the PORT variable has the value of 80. Deployable components that are connected to configuration documents that do not contain the default port value of 80 will not be returned by this scope's search, even though they should, since we state in the search expression that the port must exist in the configuration file.

Therefore, when the PORT variable has its default value, it will be removed from the scope search expression in the same way as it is ignored in dependency search expressions. For more information, see "Using Default Values of Variables" on page 90.

When a variable contains a value other than its default value, this value will be injected to the search expression in the usual way.

Troubleshooting

To test whether or not a scope returns the expected consumers, use the **searchConfigurationDocuments** JMX method in the Discovery Manager service on the UCMDB server. The “searchString” parameter should be the search expression XML, as explained in "Specify Scope of Search" on page 109. The XML should only contain constant values and no variables.

This XML can also be retrieved from the communication log of a provider in one of the search adapters. For more information, see "Dependency Search Adapters" on page 116.

Define TQL Queries

1. Add the TQL query definition to the dependency signature in the `<Queries>` section, like this:

```
...
<Queries>
   <Query name=”YourQueryName”>
      [TQL XML]
   </Query>
</Queries>
```

2. a. Use a text editor to create an empty document.

   b. Add the following content:

```
<tql:query xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition"
xmlns:ns3=
"http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefinition" xmlns:tql=
"http://www.hp.com/ucmdb/1-0-0/TopologyQueryLanguage" name="<Query
Name>">

</tql:query>
```

   c. Build your TQL query in the Modeling Studio. For more information, see "Modeling Studio" in the *HP Universal CMDB Modeling Guide*.

   d. Export the TQL query to XML.

   e. Open the XML document that you exported, and copy all the text in the `<resource>` element.

   f. Paste this text into the `<tql:query>` element of the document you created in step 1.

g. Copy the entire content of your text document and place it in the <Query> element of the dependency signature file.

Note: TQL queries are embedded in the dependency signature XML file, and are not validated. However, if the query XML file itself is invalid, an exception will be thrown during deployment. For more information, see "Compilation Errors" on the next page.

Packaging and Deploying Multiple Dependency Signature Files

Multiple dependency signature files can be deployed in UCMDB. When you search for consumer-provider links, all of the deployed files are used.

Dependency Signature files are discovery configuration files that have the prefix **dependencies/** and the suffix **.xml**, for example, **dependencies/JEE.xml** .

Each dependency signature file is completely independent. Note the following:

- Global variable definitions may be used only in the file where they are defined. It is highly recommended as much as possible to use the same names for similar variables. For example, if a variable contains an IP address and there are two dependency files, in both files a variable named IP_ADDRESS should be defined. In this way, the different search adapters could use a single destination data with the name IP_ADDRESS. For more information, see "Specify Variable Values" on page 120.
- Concept definitions may be used only in the file where they are defined. It is highly recommended as much as possible to use the same concept names and concept variable names for similar purpose concepts. From more information see "Specify Concept Variable Values" on page 120.
- Deployable components may have the same name if they are defined in different files. They will be treated as different deployable components.
- TQL queries may have the same name if they are defined in different files, but will be treated as different TQL queries. When referring to a TQL query by name in a dependency signature file, a TQL query with that name must exist in the same file.
- Scopes may have the same name if they are defined in different files. They will be treated as different scopes. When referring to a scope by name in a dependency signature file, a scope with that name must exist in the same file.

- Condition functions may have the same name if they are defined in different files. When referring to a scope by name in a dependency signature file, a function with that name must exist in the same file.
- Default values for variables and concept variables are specific to the files they were defined in. Two variables or concept variables with the same name can have a different default value in different files.
- Setting different key properties for the same concept in different dependency signature files is not recommended. This would be difficult to maintain as well as to correctly assign values from different adapters in their destination data variables. For more information, see "Specify Concept Variable Values" on page 120.

Compilation Errors

Compilation validations

- Having two or more deployable components with the same name in the same file.
- Having two or more scopes with the same name in the same file.
- Having two or more TQL queries with the same name in the same file.
- Having two or more condition functions with the same name in the same file.
- Having two dependencies with the same name under the same deployable component.
- Using a variable name in a configuration file condition that is not declared as a global variable in the same file or as a local variable in the dependency to which file configuration file belongs.
- Using a variable name in a condition function that is not declared as a global variable or as a parameter for the function.
- Having a cyclic dependency between variables (for example, if the injection statement for the variable VAR_A uses the value from VAR_B and the injection value for VAR_B uses the value of variable VAR_A .
- Referencing a TQL query that does not exist in the same file.
- Referencing a scope that does not exist in the same file.

- Referencing a condition function that does not exist in the same file.
- Using a variable that does not have a default value in an ignore statement or an alternative expression will result in a compilation error. For more information, see "Using Default Values of Variables" on page 90.
- If a search expression contains variables with default values, you must handle all cases where any combination of those variables will get their default value. This can be done by either ignoring the expression completely for some of the variables or using alternative expressions. You must have alternative expressions for all combinations of variables that do not appear in the ignore statement. Failing to do so will result in a compilation error. For more information, see "Using Default Values of Variables" on page 90.
- If a TQL query is used in a `<DocumentCILocation>` tag, but does not follow these rules:
  - The TQL query must define a path between the deployable component and the configuration document. This path must be simple (no cycles).
  - The TQL query must include the following two end nodes with these specific and case-sensitive names:
    - Deployable – the deployable component CI in the path
    - Configuration_document – the CI in the path that specifies the configuration document
  - Conditions are not permitted on the Deployable and Configuration_document nodes.
  - Cardinalities between all nodes must be 1..1.
  - Only a regular link type is supported. No compounds, joins, or sub-graphs are allowed.
- If a TQL query is used in a scope, but does not follow these rules:
  - The query node for the deployable component must be called “Deployable” (case-sensitive). The results of this query node are the possible deployable components for the scope.
  - There must be only one other query node in the TQL query (other than “Deployable”). The name of the other query node is not restricted. This node is referred to as the Scope query node.
  - The Deployable and the Scope query nodes are connected with a compound link that contains all available paths between these two nodes.
  - The “Deployable” component represents both consumer and provider components that might be

in the same scope. Therefore, the query node CI type must be an ancestor for the consumer and provider deployable CI types.

- If a Text configuration document contains multiple `<ReferenceLocation>` tags with different priorities. For more information, see "Configuration Document Overrides" on page 93.

Dependency Search Adapters

The dependency search framework must be executed by a dedicated adapter. The adapter's responsibilities are:

- Gather all relevant connection strings for a provider.
- Execute the dependency search framework.
- Report the results of the search (the dependencies) to UCMDB.

Each adapter handles one type of provider. For example, an adapter could handle JAR providers.

You can also have multiple adapters handling the same provider type. In this case, however, make sure that each provider is executed only once per adapter, to obtain consistent results.

As with other adapters, once the dependency search adapter is ready, you must create a discovery job to execute the adapter logic.

This section contains:

Create an Dependency Search Adapter

1. Select **Data Flow Management > Adapter Management**.
2. Click **New**  and select **New Adapter**.
3. Enter the adapter's details and click **OK**.
4. In the Resources pane, right click the adapter that you just created and select **Edit adapter source**.

5. Replace the line:

```
<taskInfo
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.DynamicServic
e">
```

with

```
<taskInfo
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.WorkflowServic
e">
```

6. Replace the line:

```
<params
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.DynamicService
Params" ignoreMissingReconciliationRules="false" enableRecording="false"
enableAging="true" useDefaultValueForAging="false"
autoDeleteOnErrors="success" recordResult="false" />
```

with

```
<params
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.WorkflowServic
eParams" patternType="workflow_adapter" enableAging="true"
ignoreMissingReconciliationRules="false" enableRecording="false"
autoDeleteOnErrors="success" recordResult="false"
maxThreadRuntime="86400000" useDefaultValueForAging="false">
   <workflow>
      <steps>
         <step name="Dependencies Discvoery" failure-policy="mandatory">
            <module
type="java">com.hp.ucmdb.discovery.probe.agents.probemgr.accuratedependencie
s.processing.DependenciesDiscoveryWorkflowStep</module>
            <timeoutParking>
               <initialTimeout>180000</initialTimeout>
               <retriesThreshold>12</retriesThreshold>
               <multipleBy>2</multipleBy>
               <maxRetry>10</maxRetry>
               <timeoutThreshold>10800000</timeoutThreshold>
            </timeoutParking>
         </step>
      </steps>
```

```
      <finalStep />
      <libraryScripts />
   </workflow>
</params>
```

7. Prepare a script that will handle the results of the dependency signature search. For more information, see "Write a Jython Script" on page 123.

8. Add the following step to the workflow adapter:

```
<step name="Default Search Result Awaiting" failure-policy="mandatory">
   <module type="jython">[Your Jython Script Name]</module>
   <noParking />
</step>
```

9. In the Input pane, click **Select CI Type**  and select the provider CI type for this adapter.

10. Click **Edit Input Query**  and prepare the input TQL query. For more information, see "Define an Input TQL Query and Destination Data" on the next page.

11. In the Discovered CITs pane, click  and add the provider type, all possible consumer types, and the provider-consumer link type.

12. When you are finished, click **Save**.

Define a Consumer-Provider Adapter

The search functionality is driven by discovery adapters. Each adapter handles the search for a specific set of connection strings of a provider type. The adapter's input TQL query is responsible for fetching all CIs where such connection strings can be found, and injects values into the connection string variables and concepts using the trigger's destination data. For more information, see "Developing Jython Adapters" on page 41.

These adapters are of the "Workflow Adapter" type, and execute several steps that perform the search to discover the consumer-provider relationships:

1. The adapter runs the logic of composing concrete search expression for the scope's configuration document filter, by injecting the connection strings' values into the dependency signature global variables, and by instantiating concepts.

2. The adapter submits the concrete search expression of the scope to the UCMDB's Solr engine running in the UCMDB server.
3. The adapter executes a TQL query to UCMDB to fetch all required information (such as configuration documents, deployable descriptors, and so on)so the dependency signature search can be executed.
4. The adapter downloads and stores the content of the required configuration documents to the Data Flow Probe.
5. The adapter then executes the dependency searches that are relevant for the provider and the consumers that were found by the scope in steps 2 and 3.
6. A Jython script then reports the results of the dependency search.

Define an Input TQL Query and Destination Data

To compose a search expression from a dependency signature scope and conditions, each variable and concept that is mentioned in the search expression must be replaced with a concrete connection string (returned by the TQL query). Such a replacement is done based on adapter-specific mappings that you define.

The input TQL query for a dependency mapping adapter must define:

- the trigger CIs to find consumers for the provider (the SOURCE query node)
- all the connection strings that are required for the specific provider

These TQL queries should return all the connection strings that are scattered throughout the topology of CIs and realize the provider that is delivering a service. For example, if the provider is an Oracle RAC, the TQL query should return all the CIs that include any connection string a consumer needs to connect to the RAC and consume its services. Therefore, the layout definition of the TQL query should mention any attribute storing such a connection string. For example, the TQL query for a RAC should return the IP address and port through which each instance of the RAC is accessible, as well as the RAC-Service-Name.

Variables and concepts must be mapped to attributes of query nodes in the input TQL query using the adapter's destination data definition. Each pattern element should have a unique name to allow correct mapping.

As for any adapter, the input TQL query for a search adapter will be executed for any trigger that matches the trigger TQL query for the job associated with this adapter.

When using the adapter for service discovery, you might want to limit the trigger not only to triggers that were discovered by the server (which occurs automatically), but also to triggers that are connected to the business service CI (by some path). To accomplish this, define the input TQL query so that it contains a path (usually some compound link) between the trigger and the business service CI, and name the query node of the business service CI “SERVICE” (case-sensitive). Then, when the framework sees a “SERVICE” query node, it will automatically limit the results only to the service (or services) to which that trigger CI belongs.

Specify Variable Values

Each search adapter needs to set values for all global variables and concepts that appear in search conditions in the dependency signature for dependencies that the adapter is supposed to discover. Those values represents the provider’s connection string (for details, see "Variables and Concepts" on page 80), and will be used to find consumers for the provider (the trigger).

Note: If one of the mapping attributes is empty, leave it as shown in the following example:

```
CONTEXT_ROOT = ${WEB_MODULE.j2eemanagedobject_contextroot:}
```

In this example, CONTEXT_ROOT is the attribute that will receive an empty value in the dependancy signature and will be ignored.

To set the value of a variable:

1. Add a destination data value that has the exact name of the variable (case sensitive).
2. Set the destination data value to a hard-coded value or a variable.

For more information, see "Developing Jython Adapters" on page 41.

Specify Concept Variable Values

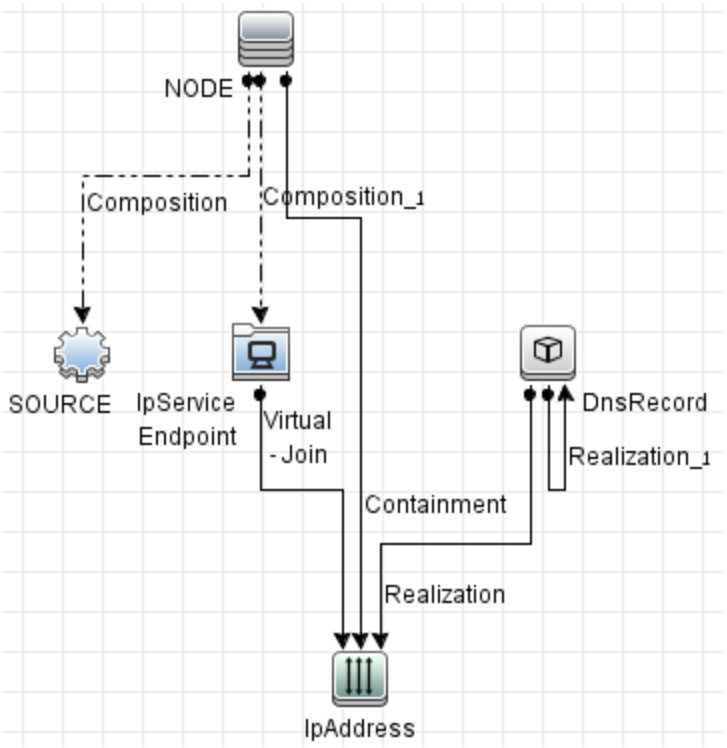
A concept should be instantiated to represent a set of inseparable, tightly-coupled connection strings. The input TQL query might return multiple inseparable sets of connection strings. For example, a running-software instance might listen on multiple IP:Port pairs. For each such set, a separate concept instance should be instantiated. The concept’s key attribute is used to separate concept instances. The key refers to a pattern element of the TQL query. For each CI instance returned for the key pattern element, a new concept instance is created. Each such CI instance is referred to as a **key CI instance**.

Note: TQL queries cannot contain multiple mapping definitions for the same concept.

The connection strings for each concept instance should be taken from its matching key CI instance and from the CIs connected to that key CI. Here is an example of a dependency signature file that contains this concept definition:

```
<Concept name="IpEndpoint">
   <Properties>
      <KeyProperty name="PORT"/>
      <Property name="IPADDRESS"/>
      <Property name="DNS"/>
   </Properties>
</Concept>
```

The adapter's input TQL query is as follows:



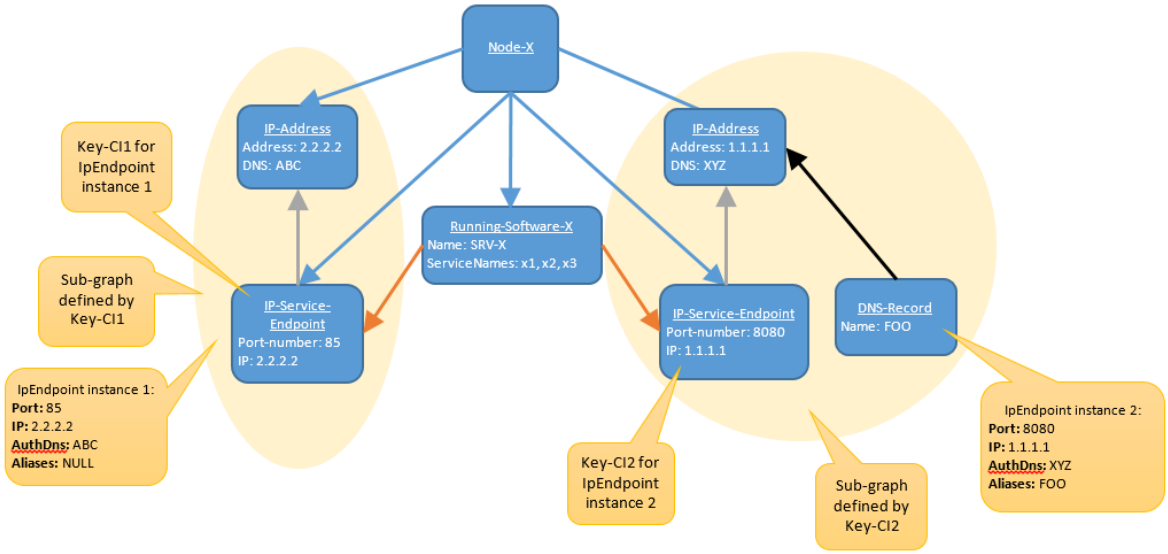
To instantiate concepts in the adapter, similar to variables, each concept variable needs to be mapped to a destination data variable. For more information, see "Specify Variable Values" on the previous page.

The adapter's destination data is as follows:

```
IpEndpoint.PORT = SOURCE.NODE.IpServiceEndpoint.name
IpEndpoint.IPADDRESS = SOURCE.NODE.IpServiceEndpoint.IpAddress.name
IpEndpoint.DNS = SOURCE.NODE.IpServiceEndpoint.IpAddress.DnsRecord.name
```

So, the **IpServiceEndpoint** query element is the key CI for the **IpEndpoint** concept in this adapter.

In the example TQL query graphic (representing the result of the TQL query for Running Software X), it is possible to recognize two key CIs. The connection strings to fill the attributes for each **IpEndpoint** concept instance are taken from the sub-graph of each key CI. This way, each concept instance represents a different set of inseparable connection strings.



Unlike variables, when mapping concept variables, you must mention the path in the input TQL query from the query node of the trigger (always named SOURCE) to the query node to which the variable will be bound. In addition, once the key variable is set, you must prefix all other concept variables with the path of the key variable.

In this example, you can see that:

- The name of the destination data variable is the name of the concept followed by the name of the concept variable.
- You can map multiple concepts in the same adapter by specifying each concept's name in the destination data variables.
- The path always starts with SOURCE.
- The paths of concept variables that are not the key variable will be prefixed with the path to the key variable.
- A path is comprised of only query node names, without links. However, at least one link must exist between two query node names specified in the path.

If a path contains one or more of the following items, the trigger will fail during its dispatching phase:

- A query node name that does not exist in the input TQL query.
- Two adjacent query node names that are not linked in the input TQL query.
- An attribute name that is not part of the resulting CI type.

If there are multiple resulting CIs for a query node that is not the key query node, the destination data variable will be a list of all the property values from these CIs. A list can only be created on destination data variables that contain a path that is not contained in any other destination data variable for the same concept. Otherwise, the trigger will fail during its dispatching phase. In the example shown above, the `IpEndpoint.IPADDRESS` destination data variable cannot contain a list, because its path is contained in the `IpEndpoint.DNS` destination data variable. `IpEndpoint.DNS` can contain a list, since its path is not contained in any other variable. This means that there can be only one result CI from the **IpAddress** query node that is connected to a result from the **IpServiceEndpoint** query node. This should be taken into consideration when you plan your TQL query and paths.

Having self-links in the TQL query will create a list in a similar way as having multiple resulting CIs in a query node that is not the key query node. Therefore, the same limitations exists for query nodes with self-links.

For more information, see "Variables and Concepts" on page 80.

Write a Jython Script

The final step in creating a dependency mapping adapter is to write a Jython script to report the results.

To access the results of the dependency search, you must retrieve them from the Workflow State using the following lines of codes:

```
workflowState = Framework.getWorkflowState()
searchResult = workflowState.getProperty
(DependenciesDiscoveryConsts.DEPENDENCIES_DISCOVERY_RESULT)
```

The searchResult variable is guaranteed to be non-null if the search step is successful. It is recommended to define the search step in the workflow as mandatory, so the Jython script step will not be executed in case the search step fails. The variable will contain an object of the type **com.hp.ucmdb.discovery.probe.agents.probemgr.accuratedependencies.search.ConsumerDeployableSearchResult**.

Use this object to do the following:

1. Review all of the consumer deployable components that were found by the dependency search. Remember: the search gets the provider's connection strings as input, and returns all the consumer deployable components that depend on the provider.

2. For each consumer, there might be more than one dependency signature.You should iterate through those as well. Each dependency signature might have different output variable values. In fact, the values of all variables that were used in a dependency signature (global, local, or concept variables) are available to the Jython script from the search results.

3. Create the Object State Holder (OSH) that contains the link between the consumer and the provider. It is possible to get the details of the provider from the search result object as well.

4. Add the OSH to the result vector (OSHV) and send the results to UCMDB.

Here is an Jython script snippet that performs the flow mentioned above:

```
# Get the search results object from the Workflow State
workflowState = Framework.getWorkflowState()
searchResult = workflowState.getProperty
(DependenciesDiscoveryConsts.DEPENDENCIES_DISCOVERY_RESULT)

# Prepare the OSHV that will contain the dependencies
oshv = ObjectStateHolderVector()
dependencyCount = 0

# Retrieve the provider OSH
providerServiceOsh = searchResult.getProviderDeployable().getDeployable()

# Loop through all the consumer deployable components
for index in range(0, searchResult.size()):
   deployable = searchResult.get(index)

   # Get the consumer deployable component's OSH
   deployableOsh = deployable.getDeployable()

   # Create an OSH for the dependency relationship between the consumer and
provider
   consumerProviderLink = modeling.createLinkOSH('consumer_provider',
deployableOsh, providerServiceOsh)

   references = []

   # Iterate through all of the dependencies that were found between the
consumer and the provider in deployable.getDependencies():
```

```
      # Extract the name of the dependency as it appears in the dependency
signature file
      dependencyName = dependency.getDependencyName()

      # Get the values of all the variables that were used by the dependency
      variables = dependency.getExportVariables()

      # Aggregate the values of the variable named REFERENCE
      # Note: The value of a variable can be a list if the variable contains
multiple values
      dependencyNames.append(dependencyName)
      for var in variables:
         varName = var.getName()
         values = var.getValues()
         if varName.lower() == REFERENCES:
            references += list(values)

   reference = references and ','.join(references)
   if reference:
      consumerProviderLink.setAttribute(REFERENCES, reference)

   # Add the link to the results OSHV
   oshv.add(consumerProviderLink)

# Send the result to the UCMDB
Framework.sendObjects(oshv)
```

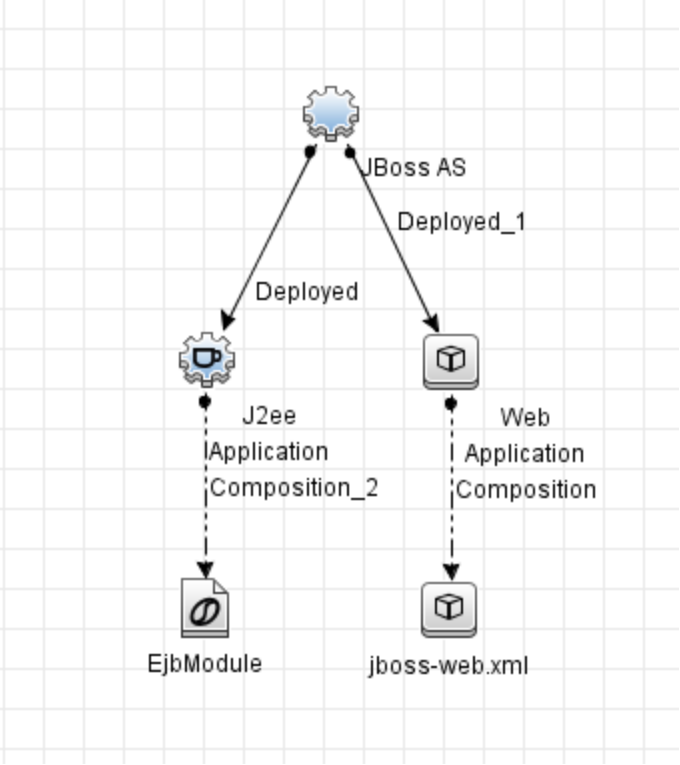
Note: The service border rules (a set of rules for the discovery rule engine) are executed based on information that is sent by the script. For that reason, it is important to send as much information as possible about the relationship, the consumer and the provider. It is recommended to create the relationship with the consumer and provider OSH that were retrieved from the search results object, as shown above. These OSH will contain all the necessary information for the service border rules (and other rule engine rules) to function properly.

Adapter Limitations

- Dependency mapping adapters are not supported for Data Flow Probes that are installed in separate mode.
- When the UCMDB server restarts, triggers for dependency mapping adapter jobs may fail due to timeout. The triggers should succeed during their next scheduled run.

Complete Example

This section will provide a complete example of how to develop a dependency signature and an adapter, which will find the dependencies between a J2ee application and a web application deployed in the same JBoss AS.



This section contains the following topics:

Development Workflow

The general flow for developing a dependency signature and an adapter is as follows:

1. Understand what is the provider CI type for the signature you are about to develop. In this example, the provider type will be the J2ee Application.

2. Decide on the consumer deployable components that are relevant for that provider that you want to cover. Since the signature is specific for the consumer's configuration documents, each

consumer will have its own dependency signatures.

In this example, we are selecting one consumer deployable component - a web application in JBoss.

3. Decide whether to add these signatures to an existing dependency signature file or to create a new one.

   There is no functional difference between the two methods. Select the option that will ease the maintenance of the dependency signatures. For example, if a consumer deployable component already exists in one of the dependency signature files, it might be easier to maintain if you add this new dependency to the existing deployable component. In this example, we will create a new dependency signature file to contain the new dependency signature.

4. Check if a new adapter is necessary or if the provider is already covered by an existing adapter. Remember that the trigger CI of the search adapters is the provider CI type. You need to check:

   a. Is there an existing search adapter that already has this CI type as the trigger CI?

      If not, you must create such an adapter. This is what we will do in this example.

   b. If a search adapter exists, are all the connection string variables and concepts that are required mapped to destination data?

      If not, you must update the existing adapter and add these variables to the destination data. Remember that the destination data is extracted from the input TQL query. This means that the input TQL query might need to change so it will bring the new information.

5. Prepare a job for the adapter, if one does not already exist.

6. Make sure the trigger TQL query also brings your providers as triggers.

Develop the Dependency Signatures

Develop dependency signatures between a JBoss J2EE Application and a Web Application.

1. Understand which connection strings are required to create the dependency between the consumer and the provider. In this example, the required connection string is the JNDI name of EJB from EJB modules contained in the provider J2EE application.

2. All of the connections strings should be defined as global variables or concepts. The values of these variables will be injected by the adapter. However, before defining these variables in the

dependency signature file (new or existing), check across all files to see if similar variables and concepts already exist. If they do exist, give your variables, concepts and concept variables the same names as the existing items. This will make developing the adapter easier, since the number of global variables that require injection will not grow. In this example, we will add these definitions to a new file, so the file will look like this:

```
<VariableDeclarations>
    <Variable name="EJB_JNDI_NAME"/>
</VariableDeclarations>
```

3. Analyze the configuration files and identify:

   - The location of each of the connection strings in each configuration file.
   - The type of each file: Properties, XML, or other text file.
   - The relationship between each of the files (if there are multiple files).

   In this example, JBoss web application defines the EJB reference in the **jboss-web.xml** configuration file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
    <!-- A reference to an EJB in the same server with a custom JNDI binding
-->
    <ejb-ref>
        <ejb-ref-name>ejb/BHome</ejb-ref-name>
        <jndi-name>someapp/ejbs/beanB</jndi-name>
    </ejb-ref>
    <!-- A reference to an EJB in an external server -->
    <ejb-ref>
        <ejb-ref-name>ejb/RemoteBHome</ejb-ref-name>
        <jndi-name>jnp://otherserver/application/beanB</jndi-name>
    </ejb-ref>
</jboss-web>
```

   The value in the `<ejb-ref-name>` section is the reference of the connection string. Since this is an XML file, we can use the following XPath expression to match the JNDI name of EJB:

```
//jboss-web/ ejb-ref/ ejb-ref-name[matches(., '^${EJB_JNDI_NAME}$')]
```

4. You must define a scope that contains a default search expression, which helps to locate the configuration file required by the dependency mapping. In this example, the keyword is the JNDI name itself. The following is the scope definition:

```
<ScopeDefinitions>
    <Scope name="JBoss_EJB_Same_Cell">
        <ConfigurationDocumentContentFilter>
            <Operator type="and">
                <Operand value="${EJB_JNDI_NAME}"/>
            </Operator>
        </ConfigurationDocumentContentFilter>
    </Scope>
</ScopeDefinitions>
```

The full dependency signature example looks like this:

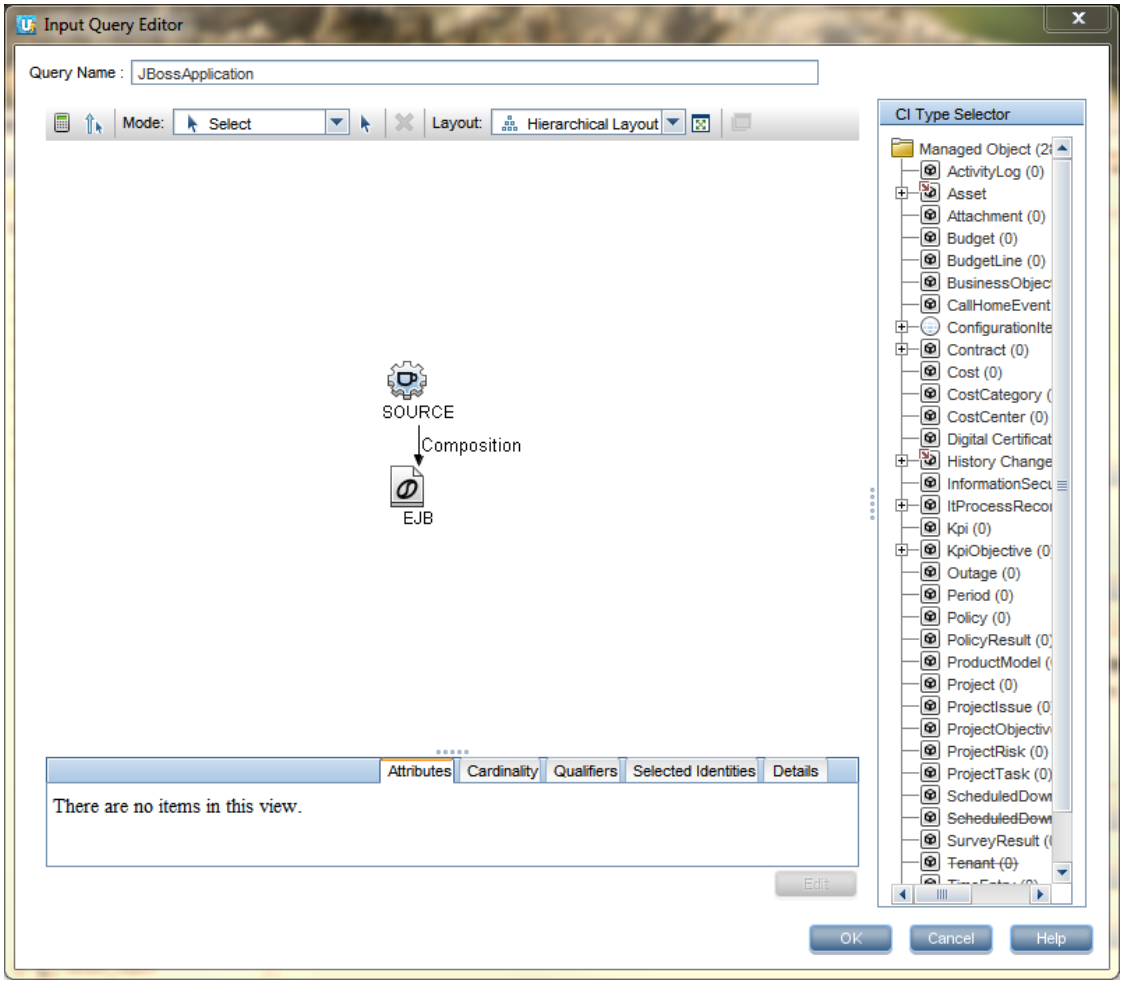
```
<?xml version="1.0"?>
<DependencySignatures xmlns="http://www.hp.com/ucmdb/1-0-0/Dependencies">
    <VariableDeclarations>
        <Variable name="EJB_JNDI_NAME"/>
    </VariableDeclarations>
    <Deployable name="JBoss J2EE Application to Web Application by JNDI" >
        <Descriptor cit="webapplication"/>
        <Dependency name="J2EE Application with Internal EJB"
providerCiType="j2eeapplication" scope="JBoss_EJB_Same">
            <XmlConfigurationDocument name="jboss-web.xml">
                <Condition>
                    <Operator type="or">
                        <XPathCondition>
                            <XPath>//ejb-ref/jndi-name[matches(., '^${EJB_JNDI_
NAME}$', 'i')]</XPath>
                        </XPathCondition>
                    </Operator>
                </Condition>
            </XmlConfigurationDocument>
        </Dependency>
    </Deployable>
    <ScopeDefinitions>
        <Scope name="JBoss_EJB_Same_Cell">
            <ConfigurationDocumentContentFilter>
                <Operator type="and">
                    <Operand value="${EJB_JNDI_NAME}"/>
                </Operator>
            </ConfigurationDocumentContentFilter>
        </Scope>
    </ScopeDefinitions>
</DependencySignatures>
```

Develop the Adapter

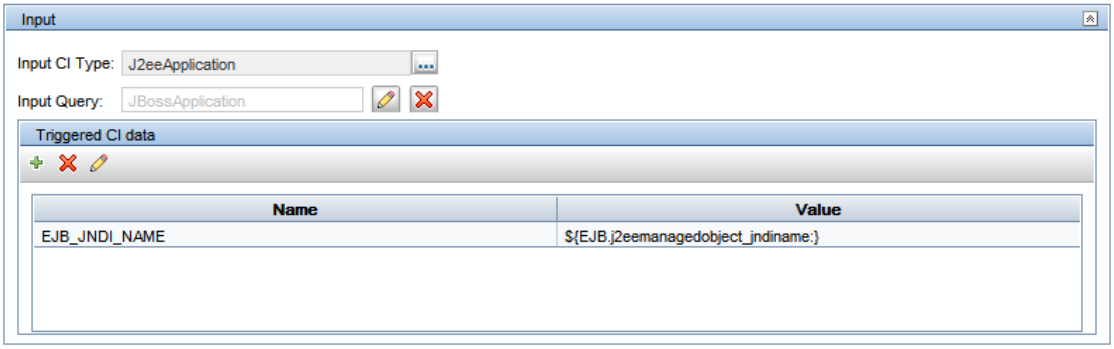
1. Create a new workflow adapter, as described in "Create an Dependency Search Adapter" on page 116.

2. Set the trigger CI type to the provider type. In this example, the J2EE Application is the provider.

3. Define input TQL query for the adapter to get the required CIs and their attributes. In this example, we need the trigger J2EEApplication CI together with EJBModule CIs, which belong to the provider J2EE application.



4. Map the global variables defined in the signatures for the required connection string variables and concepts with the trigger CI data. In this example, we map the j2eemanagedobject_jndiname attribute from the EJBModule CI to the EJB_JNDI_NAME destination data.



5. In the Workflow Steps section, paste the following workflow definition:

```
<workflow>
    <steps>
        <step name="Accurate Dependency Search" failure-policy="mandatory">
            <module type="jython">DependenciesDiscovery.py</module>
            <timeoutParking>
                <initialTimeout>60000</initialTimeout>
                <retriesThreshold>1</retriesThreshold>
                <multipleBy>1</multipleBy>
                <maxRetry>20</maxRetry>
                <timeoutThreshold>60000</timeoutThreshold>
            </timeoutParking>
        </step>
    </steps>
    <finalStep>
        <module type="jython">AccurateDependencyMapping.py</module>
    </finalStep>
    <libraryScripts />
</workflow>
```

You can change the timeout duration by adjusting the **timeoutParking** parameter.

6. Create a trigger TQL query for the new adapter.

7. Add the job definition in the Service Discovery Activity Type as follows:

```
<ServiceDiscoveryActivityType id="top-down" displayName="Top-down">
    <JobsDefinitions>
       …
        <job id=" JBoss Application to Web Application " displayName=" JBoss
Application to Web Application ">
            <patternId>JBossApplication2WebApplication</patternId>
            <triggers>
                <trigger>jboss_application_trigger</trigger>
```

```
            </triggers>
            <parameters/>
        </job>
        …
    </JobsDefinitions>
</ServiceDiscoveryActivityType>
```

Chapter 5: Developing Generic Database Adapters

This chapter includes:

Generic Database Adapter Overview

The purpose of the generic database adapter platform is to create adapters that can integrate with relational database management systems (RDBMS) and run TQL queries and population jobs against the database. The RDBMS supported by the generic database adapter are Oracle, Microsoft SQL Server, and MySQL.

This version of the database adapter implementation is based on a JPA (Java Persistence API) standard with the Hibernate ORM library as the persistence provider.

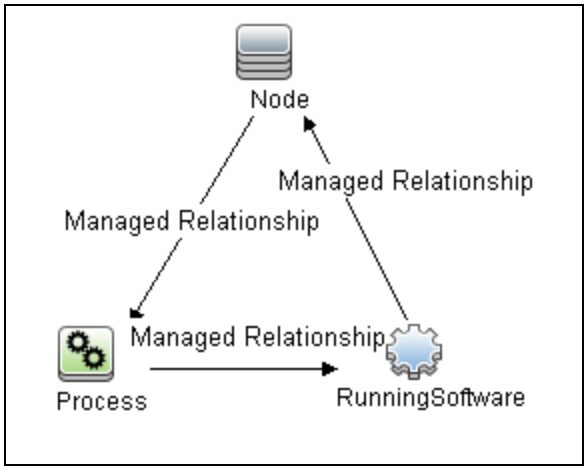
TQL Queries for the Generic Database Adapter

For population jobs, every required layout of a CI must be checked in the Layout Settings Dialog Box in the Modeling Studio. For details, see Query Node/Relationship Properties Dialog Box in the *HP Universal CMDB Modeling Guide*. It is important to note that a CI might require an attribute to be identified, and without those attributes the CI will fail to be added to UCMDB.

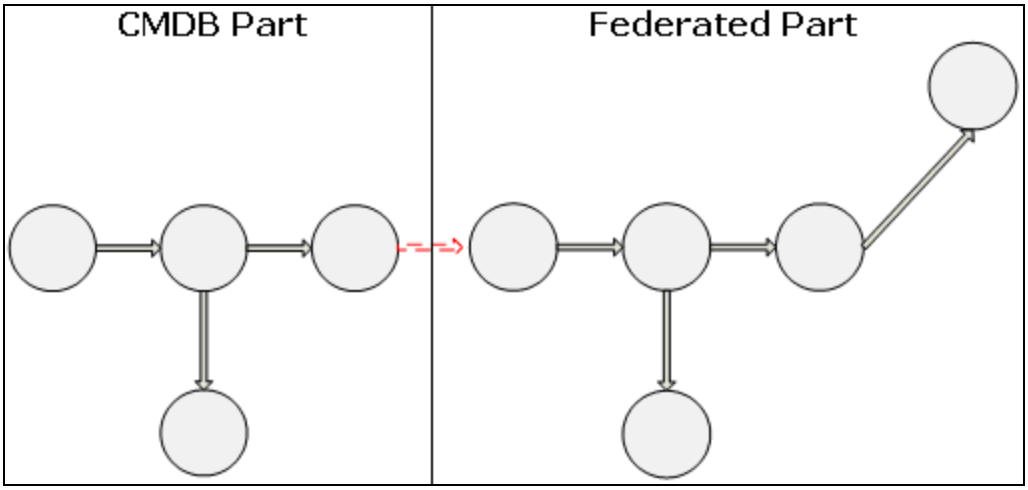
The following limitations exist on the TQL queries calculated by the Generic Database Adapter only:

- subgraphs are not supported
- compound relationships are not supported
- cycles or cycle parts are not supported

  The following TQL query is an example of a cycle:



- Function layout is not supported.
- 0..0 cardinality is not supported.
- The `Join` relationship is not supported.
- Qualifier conditions are not supported.
- To connect between two CIs, a relationship in the form of a table or foreign key must exist in the external database source.



Reconciliation

Reconciliation is carried out as part of the TQL calculation on the adapter side. For reconciliation to occur, the CMDB side is mapped to a federated entity called reconciliation CIT.

Mapping. Each attribute in the CMDB is mapped to a column in the data source.

Although mapping is done directly, transformation functions on the mapping data are also supported. You can add new functions through the Java code (for example, lowercase, uppercase). The purpose of these functions is to enable value conversions (values that are stored in the CMDB in one format and in the federated database in another format).

Note:

- To connect the CMDB and external database source, an appropriate association must exist in the database. For details, see "Prerequisites" on page 139.

- Reconciliation with the CMDB ID is also supported
- Reconciliation with the Global ID is also supported.

Hibernate as JPA Provider

Hibernate is an object-relational (OR) mapping tool, which enables mapping Java classes to tables over several types of relational databases (for example, Oracle and Microsoft SQL Server). For details, see "Functional Limitations" on page 216.

In an elementary mapping, each Java class is mapped to a single table. More advanced mapping enables inheritance mapping (as can occur in the CMDB database).

Other supported features include mapping a class to several tables, support for collections, and associations of types one-to-one, one-to-many, and many-to-one. For details, see "Associations" on page 138 below.

For our purposes, there is no need to create Java classes. The mapping is defined from the CMDB class model CITs to the database tables.

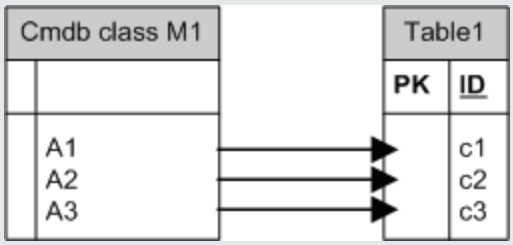
This section also includes the following topics:

Examples of Object-Relational Mapping

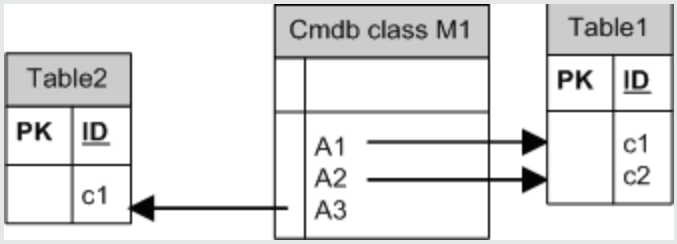
The following examples describe object-relational mapping:

Example of One CMDB Class Mapped to One Database Table:

Class `M1`, with attributes `A1`, `A2`, and `A3`, is mapped to table `1` columns `c1`, `c2`, and `c3`. This means that any `M1` instance has a matching row in table `1`.

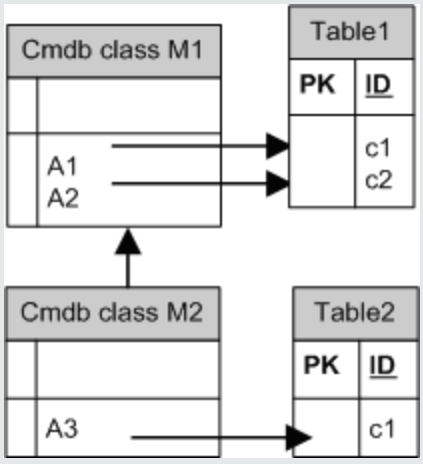


Example of One CMDB Class Mapped to Two Database Tables:



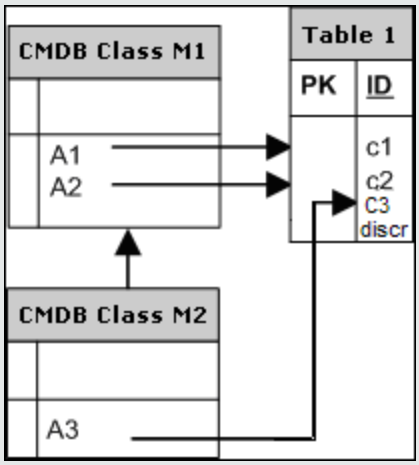
Example of Inheritance:

This case is used in the CMDB, where each class has its own database table.



Example of Single Table Inheritance with Discriminator:

An entire hierarchy of classes is mapped to a single database table, whose columns comprise a super-set of all attributes of the mapped classes. The table also contains an additional column (`Discriminator`), whose value indicates which specific class should be mapped to this entry.



Associations

There are three types of associations: one-to-many, many-to-one and many-to-many. To connect between the different database objects, one of these associations must be defined by using a foreign key column (for the one-to-many case) or a mapping table (for the many-to-many case).

Usability

As the JPA schema is very extensive, a streamlined XML file is provided to make it easier to define associations.

The use case for using this XML file is as follows: Federated data is modeled into one federated class. This class has many-to-one relations to a non-federated CMDB class. In addition, there is only one possible relation type between the federated class and the non-federated class.

Prepare for Adapter Creation

This task describes the preparations that are necessary for creating an adapter.

Note: You can view samples for the Generic DB adapter in the UCMDB API. Specifically, the DDMi Adapter sample contains a complicated **orm.xml** file, as well as the implementations for some plug-in interfaces.

This task includes the following steps:

- "Prerequisites" below
- "Create a CI Type" on page 141
- "Create a Relationship" on page 141

1. **Prerequisites**

   To validate that you can use the database adapter with your database, check the following:

   - The reconciliation classes and their attributes (also known as multinodes) exist in the database. For example, if the reconciliation is run by node name, verify that there is a table that contains a column with node names. If the reconciliation is run according to node `cmdb_id`, verify that there is a column with CMDB IDs that matches the CMDB IDs of the nodes in the CMDB. For details on reconciliation, see "Reconciliation" on page 135.

ID	NAME	IP_ADDRESS
31	BABA	16.59.33.60
33	ext3.devlab.ad	16.59.59.116
46	LABM1MAM15	16.59.58.188
72	cert-3-j2ee	16.59.57.100
102	labm1sun03.devlab.ad	16.59.58.45
114	LABM2PCOE73	16.59.66.79
116	CUT	16.59.41.214
117	labm1hp4.devlab.ad	16.59.60.182

   - To correlate two CITs with a relationship, there must be correlation data between the CIT tables. The correlation can be either by a foreign key column or by a mapping table. For example, to correlate between node and ticket, there must be a column in the ticket table that contains the node ID, a column in the node table with the ticket ID that is connected to it, or a mapping table whose `end1` is the node ID and `end2` is the ticket ID. For details on correlation data, see "Hibernate as JPA Provider" on page 136.

     The following table shows the foreign key `NODE_ID` column:

NODE_ID	CARD_ID	CARD_TYPE	CARD_NAME
2015	1	Serial Bus Controller	Intel 82801EB USB Universal Host Controller
3581	2	System	Intel 631xESB/6321ESB/3100 Chipset LPC
3581	3	Display	ATI ES1000
3581	4	Base System Peripheral	HP ProLiant iLO 2 Legacy Support Function

- Each CIT can be mapped to one or more tables. To map one CIT to more than one table, check that there is a primary table whose primary key exists in the other tables, and is a unique value column.

  For example, a ticket is mapped to two tables: `ticket1` and `ticket2`. The first table has columns `c1` and `c2` and the second table has columns `c3` and `c4`. To enable them to be considered as one table, both must have the same primary key. Alternatively, the first table primary key can be a column in the second table.

  In the following example, the tables share the same primary key called `CARD_ID`:

CARD_ID	CARD_TYPE	CARD_NAME
1	Serial Bus Controller	Intel 82801EB USB Universal Host Controller
2	System	Intel 631xESB/6321ESB/3100 Chipset LPC
3	Display	ATI ES1000
4	Base System Peripheral	HP ProLiant iLO 2 Legacy Support Function

CARD_ID	CARD_VENDOR
1	Hewlett-Packard Company
2	(Standard USB Host Controller)
3	Hewlett-Packard Company
4	(Standard system devices)
5	Hewlett-Packard Company

2. **Create a CI Type**

   In this step you create a CIT that represents the data in the RDBMS (the external data source).

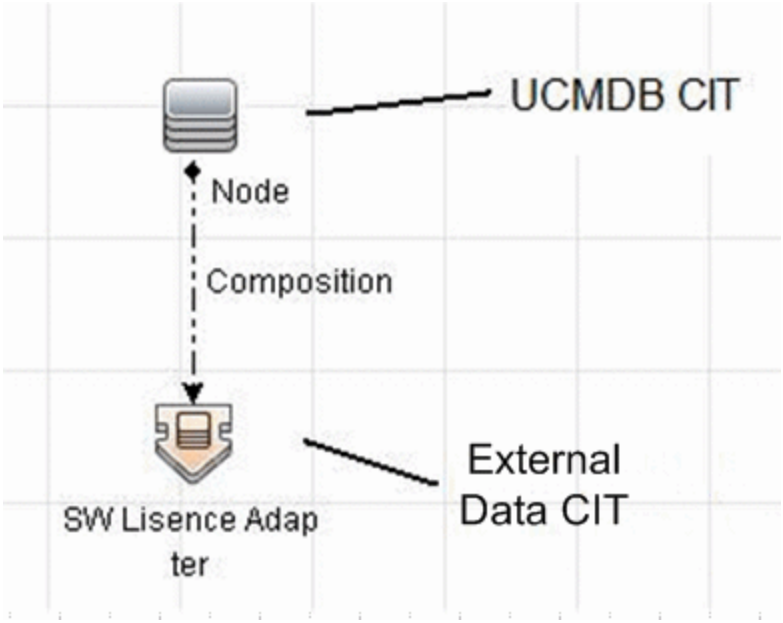
   a. In UCMDB, access the CI Type Manager and create a new CI Type. For details, see How to Create a CI Type in the *HP Universal CMDB Modeling Guide*.

   b. Add the necessary attributes to the CIT, such as last access time, vendor, and so on. These are the attributes that the adapter will retrieve from the external data source and bring into CMDB views.

3. **Create a Relationship**

   In this step you add a relationship between the UCMDB CIT and the new CIT that represents the data from the external data source.

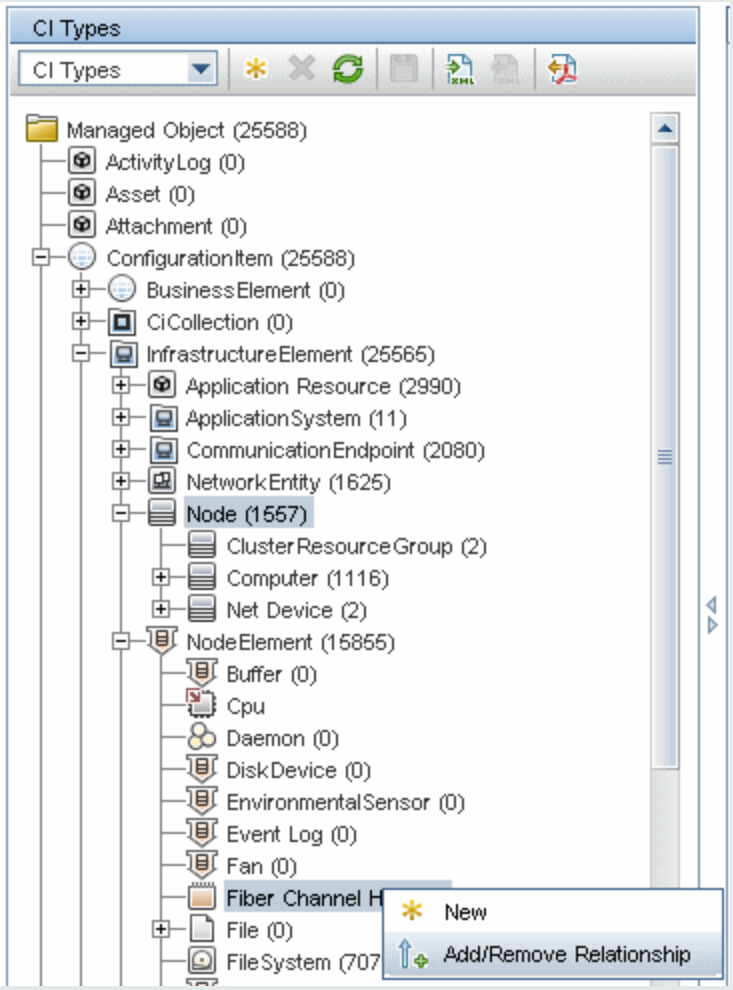
   Add appropriate, valid relationships to the new CIT. For details, see Add/Remove Relationship Dialog Box in the *HP Universal CMDB Modeling Guide*.



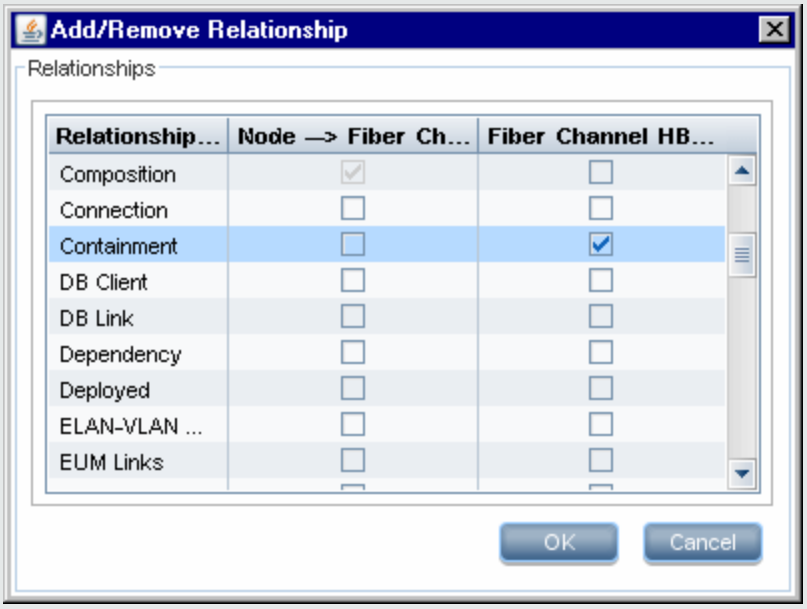
Note: At this stage, you cannot yet view the federated data or populate the external data, as you have not yet defined the method for bringing in the data.

Example of Creating a Containment Relationship:

a. In the CIT Manager, select the two CITs:



b. Create a **Containment** relationship between the two CITs:



Prepare the Adapter Package

In this step, you locate and configure the Generic DB adapter package.

1. Locate the **db-adapter.zip** package in the **C:\hp\UCMDB\UCMDBServer\content\adapters** folder.
2. Extract the package to a local temporary directory.
3. Edit the adapter XML file:
   - Open the **discoveryPatterns\db_adapter.xml** file in a text editor.
   - Locate the **adapter id** attribute and replace the name:

```
<pattern id="MyAdapter" displayLabel="My Adapter"
xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery
Pattern Description"
         schemaVersion="9.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" displayName="UCMDB API Population">
```

If the adapter supports population, the following capability should be added to the **<adapter-capabilities>** element:

```
<support-replicatioin-data>
    <source>
        <changes-source>
    </source>
</support-replicatioin-data>
```

The display label or ID appears in the list of adapters in the Integration Point pane in HP Universal CMDB.

When creating a Generic DB Adapter it is not necessary to edit the **changes-source** tag in the **support-replicatioin-data** tag. If the **FcmdbPluginForSyncGetChangesTopology** plug-in is implemented, the changed topology from the last run will be returned. If the plug-in is not implemented, the full topology will be returned and the auto-delete will be performed according to the returned CIs.

For details about populating the CMDB with data, see "Integration Studio Page" in the *HP Universal CMDB Data Flow Management Guide*.

- If the adapter is using the mapping engine from version 8.x (meaning that it is not using the new reconciliation mapping engine), replace the following element:

  ```
  <default-mapping-engine>
  ```

  with:

  ```
  <default-mapping-engine>com.hp.ucmdb.federation.
  mappingEngine.AdapterMappingEngine</default-mapping-engine>
  ```

  To revert to the new mapping engine, return the element to the following value:

  ```
  <default-mapping-engine>
  ```

- Locate the **category** definition:

  ```
  <category>Generic</category>
  ```

  Change the **Generic** category name to the category of your choice.

  > **Note:** Adapters whose categories are specified as **Generic** are not listed in the Integration Studio when you create a new integration point.

- The connection to the database can be described using a user name (schema), password, database type, database host machine name, and database name or SID.

For this type of connection, parameters have the following elements in the **parameter** section of the adapter's XML file:

```
<parameters>
    <!--The description attribute may be written in simple text or HTML.-
->
    <!--The host attribute is treated as a special case by UCMDB-->
    <!--and will automatically select the probe name (if possible)-->
    <!--according to this attribute's value.-->
    <!--Display name and description may be overwritten by I18N values-->
        <parameter name="host" display-name="Hostname/IP" type="string"
description="The host name or IP address of the remote machine"
mandatory="false" order-index="10" />
        <parameter name="port" display-name="Port" type="integer"
description="The remote machine's connection port" mandatory="false"
order-index="11" />
        <parameter name="dbtype" display-name="DB Type" type="string"
description="The type of database" valid-
values="Oracle;SQLServer;MySQL;BO" mandatory="false" order-
index="13">Oracle</parameter>
        <parameter name="dbname" display-name="DB Name/SID" type="string"
description="The name of the database or its SID (in case of Oracle)"
mandatory="false" order-index="13" />
        <parameter name="credentialsId" display-name="Credentials ID"
type="integer" description="The credentials to be used" mandatory="true"
order-index="12" />
</parameters>
```

Note: This is the default configuration. Therefore, the **db_adapter.xml** file already contains this definition.

There are situations in which the connection to the database cannot be configured in this way. For example, connecting to Oracle RAC or connecting using a database driver other than the one supplied with the CMDB.

For these situations, you can describe the connection using user name (schema), password, and a connection URL string.

To define this, edit the adapter's XML parameters section as follows:

```
<parameters>
    <!--The description attribute may be written in simple text or HTML.-->
    <!--The host attribute is treated as a special case by CMDBRTSM-->
```

```
    <!--and will automatically select the probe name (if possible)-->
    <!--according to this attribute's value.-->
    <!--Display name and description may be overwritten by I18N values-->
        <parameter name="url" display-name="Connection String" type="string"
description="The connection string to connect to the database"
mandatory="true" order-index="10" />
        <parameter name="credentialsId" display-name="Credentials ID"
type="integer" description="The credentials to be used" mandatory="true"
order-index="12" />
</parameters>
```

An example of a URL that connects to an Oracle RAC using the out-of-the- box Data Direct driver is: **jdbc:mercury:oracle://labm3amdb17:1521;ServiceName=RACQA;AlternateServers=(labm3amdb18:1521);LoadBalancing=true**.

4. In the temporary directory, open the **adapterCode** folder and rename **GenericDBAdapter** to the value of **adapter id** that was used in the previous step.

   This folder contains the adapter's configuration, for example, the adapter name, the queries and classes in the CMDB, and the fields in the RDBMS that the adapter supports.

5. Configure the adapter as required. For details, see "Configure the Adapter – Minimal Method" below.

6. Create a `*.zip` file with the same name as you gave to the **adapter id** attribute, as described in the step "Edit the adapter XML file:" on page 143.

   Note: The **descriptor.xml** file is a default file that exists in every package.

7. Save the new package that you created in the previous step. The default directory for adapters is: **C:\hp\UCMDB\UCMDBServer\content\adapters**.

Configure the Adapter – Minimal Method

The simplified (minimal) method is a method for creating the **simplifiedConfiguration.xml** mapping file that is used by the adapter. This method enables a basic population or federation of a single CIT.

The instructions provided in this section describes a method of mapping the class model for certain CI Types in the CMDB to an RDBMS.

All of the configuration files mentioned in this section are located in the **db-adapter.zip** package in the **C:\hp\UCMDB\UCMDBServer\content\adapters** folder that you extracted in "Prepare the Adapter Package" on page 143.

Note: The **orm.xml** file that is automatically generated as a result of running this method is a good example that you can use when working with the advanced method.

You would use this minimal method when you need to:

- Federate/populate a single node such as a node attribute.
- Demonstrate the Generic Database Adapter capabilities.

This method:

- Supports one-node federation/population only
- Supports many-to-one virtual relationships only

Configure the adapter.conf File

To change the settings in the `adapter.conf` file so that the adapter uses the simplified configuration method:

1. Open the **adapter.conf** file in a text editor.
2. Locate the following line: **use.simplified.xml.config=<true/false>**.
3. Change it to **use.simplified.xml.config=true**.

Example: Populating a Node and IP Address using the Simplified Method

This example demonstrates populating a **Node** related by a containment link to **IP Address** into UCMDB. The RDBMS has a table named **simpleNode** that contains data on the computer name, computer node, and the IP address of the computer.

The content of the **simpleNode** table is shown below:

	host_id	host_name	note	ip_address
	1	Comp1	Test Computer 1	12.33.211.52
	2	Comp2	Test Computer 2	12.33.211.53
	3	Comp3	Test Computer 3	12.33.211.54
	4	Comp4	Test Computer 4	12.33.211.55

The population is performed in three stages, as follows:

1. "Create the simplifiedConfiguration.xml" below
2. "Create the TQL" on page 150
3. "Create an Integration Point" on page 151

Create the simplifiedConfiguration.xml

Create the **simplifiedConfiguration.xml** as follows:

1. Create a **cmdb-class** entity as follows:

   ```
   <cmdb-class cmdb-class-name="node" default-table-name="simpleNode">
   ```

   The CI Type is **node** and the RDBMS table name is **simpleNode**.

2. Set the primary-key of the table as follows:

   ```
   <primary-key column-name="host_id"/>
   ```

   This primary key is equivalent to entity id in the **orm.xml** file.

3. Set the **reconciliation-by-two-nodes** rule as follows:

   ```
   <reconciliation-by-two-nodes connected-node-cmdb-class-name="ip_address" cmdb-link-type="containment">
   ```

   This tag defines the relation between the **Node** and the **IpAddress** CI types. The relation type is Containment link. The reconciliation is done by the two connected CI Types. The attribute mapping of the connected node (in this case IpAddress) is defined in the **connected-node** attribute.

4. Add the **or** condition between the reconciliation attributes as follows:

   ```
   <or is-ordered="true">
   ```

This tag defines an OR relationship between the reconciliation attributes, meaning the first reconciliation attribute that is **true** sets the whole reconciliation rule to **true**.

5. Add the following attributes:

   ```
   <attribute cmdb-attribute-name="name" column-name="host_name" ignore-
   case="true"/>
   ```

   This tag sets a mapping between the **node.name** in the UCMDB to the column **host_name** in the **simpleNode** table.

   Do the same with **data_note** attribute:

   ```
   <attribute cmdb-attribute-name="data_note" column-name="note" ignore-
       case="true"/>
   ```

   Add the connected node attribute:

   ```
   <connected-node-attribute cmdb-attribute-name="name" column-name="ip_address"/>
   ```

   This tag sets a mapping between the **ip_address.name** to the column **ip_address** in the **simpleNode** table.

6. Close the opened tag by order:

   ```
   </or>
   ```

   ```
   </reconciliation-by-two-nodes>
   ```

   ```
   </cmdb-class>
   ```

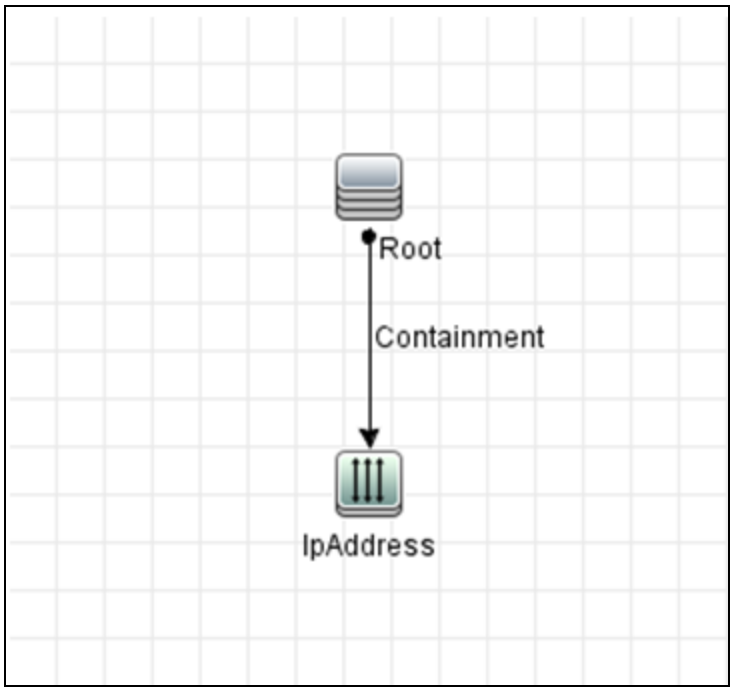
The contents of the simplifiedConfiguration.xml file now appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-db-adapter-config xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:noNamespaceSchemaLocation="../META-
CONF/simplifiedConfiguration.xsd">
   <cmdb-class cmdb-class-name="node" default-table-name="simpleNode">
       <primary-key column-name="host_id"/>
   <reconciliation-by-two-nodes connected-node-cmdb-class-name="ip_address" cmdb-
link-type="containment">
       <or is-ordered="true">
       <attribute cmdb-attribute-name="name" column-name="host_name" ignore-
case="true"/>
       <attribute cmdb-attribute-name="data_note" column-name="note" ignore-
case="true"/>
          <connected-node-attribute cmdb-attribute-name="name" column-name="ip_
address"/>
       </or>
       </reconciliation-by-two-nodes>
```

```
    </cmdb-class>
</generic-db-adapter-config>
```

Create the TQL

The TQL is a **node** connected by a containment link to **ip_address**. The node should be marked as **root**, as shown below.



To create the TQL:

1. Go to **Modeling > Modeling Studio**.
2. Click the **New** button and create a new query.
3. Go to the CI Types tab and drag the Node CI Type and IpAddress CI Type to the TQL screen.
4. Connect **Node** and **IpAddress** with a Containment relationship.
5. Right-click on the **Node** element and choose Query Node Properties.
6. Change **Element name** to **Root**.
7. Go to the **Element Layout** tab. Select **Specific Attributes** as the Attributes condition. Choose **Name** and **Note** from the Available Attributes window and move them to the Specific Attributes window.

8. Right-click on the **IpAddress** element and choose Query Node Properties.
9. Go to the **Element Layout** tab. Select **Specific Attributes** as the Attributes condition. Choose **Name** from the Available Attributes window and move it to the Specific Attributes window.
10. Save the TQL.

Create an Integration Point

Create the Integration Point as follows:

1. Go to **Data Flow Management > Integration Studio**, and click the **New Integration Point** button.
2. Insert the details of the Integration Point and click **OK**.
3. In the Population tab, select the **New Integration Job** button, and add the TQL previously created.
4. Save the Integration Point and click the **Run Full Synchronization** button.

Configure the Adapter – Advanced Method

These configuration files are located in the **db-adapter.zip** package in the **C:\hp\UCMDB\UCMDBServer\content\adapters** folder that you extracted when preparing the adapter package. For details, see "Prepare the Adapter Package" on page 143.

This task includes the following steps:

- "Configure the orm.xml File" below
- "Configure the reconciliation_rules.txt File " on page 155

Configure the orm.xml File

In this step, you map the CITs and relationships in the CMDB to the tables in the RDBMS.

1. Open the **orm.xml** file in a text editor.

   This file, by default, contains a template that you use to map as many CITs and relationships as needed.

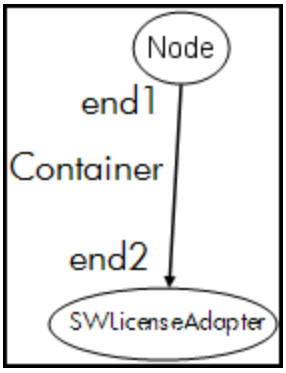
   Note: Do not edit the **orm.xml** file in any version of Notepad from Microsoft Corporation. Use

Notepad++, UltraEdit, or some other third-party text editor.

2. Make changes to the file according to the data entities to be mapped. For details, see the following examples.

   The following types of relationships may be mapped in the **orm.xml** file:

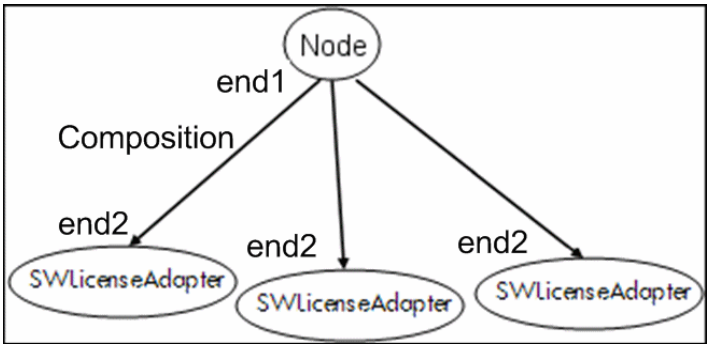
   - One to one:



   The code for this type of relationship is:

```
<one-to-one name="end1" target-entity="node">
        <join-column name="Device_ID" >
</one-to-one>
<one-to-one name="end2" target-entity="sw_sub_component">
        <join-column name="Device_ID" >
        <join-column name="Version_ID" >
</one-to-one>
```

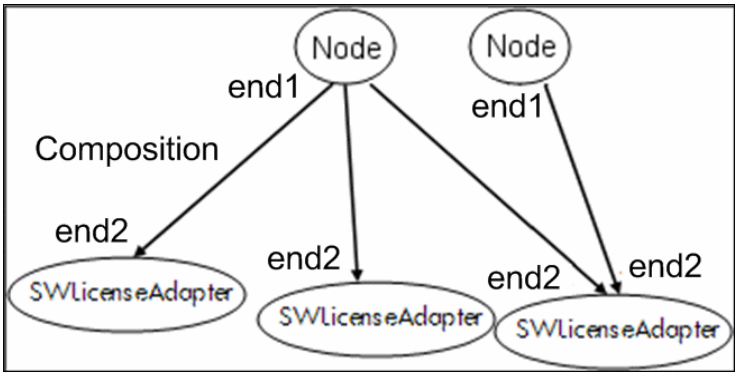
   - Many to one:



   The code for this type of relationship is:

```
<many-to-one name="end1" target-entity="node">
        <join-column name="Device_ID" >
</many-to-one>
<one-to-one name="end2" target-entity="sw_sub_component">
        <join-column name="Device_ID" >
        <join-column name="Version_ID" >
</one-to-one>
```

- Many to many:



The code for this type of relationship is:

```
<many-to-one name="end1" target-entity="node">
        <join-column name="Device_ID" >
</many-to-one>
<many-to-one name="end2" target-entity="sw_sub_component">
        <join-column name="Device_ID" >
        <join-column name="Version_ID" >
</many-to-one>
```

For details about naming conventions, see "Naming Conventions" on page 179.

Example of Entity Mapping Between the Data Model and the RDBMS:

Note: Attributes that do not have to be configured are omitted from the following examples.

- The class of the CMDB CIT:

```
<entity class="generic_db_adapter.node">
```

- The name of the table in the RDBMS:

```
<table name="Device" />
```

- The column name of the unique identifier in the RDBMS table:

```
<column name="Device ID" />
```

- The name of the attribute in the CMDB CIT:

```
<basic name="name">
```

- The name of the table field in the external data source:

```
<column name="Device_Name" />
```

- The name of the new CIT you created in "Create a CI Type" on page 141:

```
<entity class="generic_db_adapter.MyAdapter">
```

- The name of the corresponding table in the RDBMS:

```
<table name="SW_License" />
```

- The unique identity in the RDBMS:

- The attribute name in the CMDB CIT and the name of the corresponding attribute in the RDBMS:

Example of Relationship Mapping Between the Data Model and the RDBMS:

- The class of the CMDB relationship:

```
<entity class="generic_db_adapter.node_containment_MyAdapter">
```

- The name of the RDBMS table where the relationship is performed:

```
<table name="MyAdapter" />
```

- The unique ID in the RDBMS:

```
<id name="id1">
        <column updatable="false" insertable="false"
name="Device_ID">
        <generated-value strategy="TABLE" />
</id>
<id name="id2">
        <column updatable="false" insertable="false"
name="Version_ID">
        <generated-value strategy="TABLE" />
</id>
```

- The relationship type and the CMDB CIT:

```
<many-to-one target-entity="node" name="end1">
```

- The primary key and foreign key fields in the RDBMS:

```
<join-column updatable="false" insertable="false"
referenced-column-name="[column_name]" name="Device_ID" />
```

Configure the reconciliation_rules.txt File

In this step you define the rules by which the adapter reconciles the CMDB and the RDBMS (only if Mapping Engine is used, for backward compatibility with version 8.x):

1. Open **META-INF\reconciliation_rules.txt** in a text editor.

2. Make changes to the file according to the CIT you are mapping. For example, to map a node CIT, use the following expression:

```
multinode[node] ordered expression[^name]
```

Note:

- If the data in the database is case sensitive, do not delete the control character (**^**).
- Check that each opening square bracket has a matching closing bracket.

For details, see "The reconciliation_rules.txt File (for backwards compatibility)" on page 190.

Implement a Plug-in

This task describes how to implement and deploy a Generic DB Adapter with plug-ins.

Note: Before writing a plug-in for an adapter, make sure you have completed all the necessary steps in "Prepare the Adapter Package" on page 143.

1. Option 1 – Write a Java based plug-in

   a. Copy the following jar files from the UCMDB server installation directory to your development class path:

      - Copy the **db-interfaces.jar** file and **db-interfaces-javadoc.jar** file from the **tools\adapter-dev-kit\db-adapter-framework** folder.
      - Copy the **federation-api.jar** file and **federation-api-javadoc.jar** file from the **\tools\adapter-dev-kit\SampleAdapters\production-lib** folder.

        Note: More information about developing a plug-in can be found in the **db-interfaces-javadoc.jar** and **federation-api-javadoc.jar** files and in the online documentation at:

        - **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DBAdapterFramework_JavaAPI\index.html**
        - **C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\Federation_JavaAPI\index.html**

   b. Write a Java class implementing the plug-in's Java interface. The interfaces are defined in the **db-interfaces.jar** file. The table below specifies the interface that must be implemented for each plug-in:

Plug-in Type	Interface Name	Method
Synchronize Full Topology	FcmdbPluginForSyncGetFullTopology	getFullTopology
Synchronize Changes	FcmdbPluginForSyncGetChangesTopology	getChangesTopology
Synchronize Layout	FcmdbPluginForSyncGetLayout	getLayout
Retrieve Supported Queries	FcmdbPluginForSyncGetSupportedQueries	getSupportedQueries
Alter TQL query definition and results	FcmdbPluginGetTopologyCmdbFormat	getTopologyCmdbFormat
Alter layout request for CIs	FcmdbPluginGetCIsLayout	getCisLayout
Alter layout request for links	FcmdbPluginGetRelationsLayout	getRelationsLayout
Push Back IDs	FcmdbPluginPushBackIds	getPushBackIdsSQL

The plug-in's class must have a public default constructor. Also, all of the interfaces expose a method called initPlugin. This method is guaranteed to be called before any other method and is used to initialize the adapter with the containing adapter's environment object.

If **FcmdbPluginForSyncGetChangesTopology** is implemented, there are two different ways to report the changes:

- **Report the entire root topology at all times.** According to this topology, the auto-delete function finds which CIs should be removed. In this case, the auto-delete function should be enabled by using the following:

```
<autoDeleteCITs isEnabled="true">
                <CIT>link</CIT>
                <CIT>object</CIT>
            </autoDeleteCITs>
```

- **Report each CI instance that was removed/updated.** In this case the auto-delete mechanism should be disabled by using the following:

```
<autoDeleteCITs isEnabled="false">
                <CIT>link</CIT>
                <CIT>object</CIT>
            </autoDeleteCITs>
```

c. Make sure you have the Federation SDK JAR and the Generic DB Adapter JARs in your class path before compiling your Java code. The Federation SDK is the **federation_api.jar** file, which can be found in the **C:\hp\UCMDB\UCMDBServer\lib** directory.

d. Pack your class into a jar file and put it under the adapterCode\<Your Adapter Name> folder in the adapter package, prior to deploying it.

2. Option 2 – Write a Groovy based plug-in

   a. Create a Groovy code file (MyPlugin.groovy) in the Adapter Management Menu, under the adapter package configuration files.

   b. In the Groovy class, implement the appropriate interfaces. The interfaces are defined in the db-interfaces.jar file, see the table above.

3. The plug-ins are configured using the **plugins.txt** file, located in the **\META-INF** folder of the adapter.

   The following is an example of the file from the DDMi adapter:

```
# mandatory plugin to sync full topology
[getFullTopology]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# mandatory plugin to sync changes in topology
[getChangesTopology]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# mandatory plugin to sync layout
[getLayout]
com.hp.ucmdb.adapters.ed.plugins.replication.EDReplicationPlugin
# plugin to get supported queries in sync. If not defined return all tqls
names
[getSupportedQueries]
# internal not mandatory plugin to change tql definition and tql result
[getTopologyCmdbFormat]
# internal not mandatory plugin to change layout request and CIs result
[getCisLayout]
# internal not mandatory plugin to change layout request and relations
result
```

```
[getRelationsLayout]
# internal not mandatory plugin to change action on pushBackIds
[pushBackIds]
```

Legend:

- A comment line.

[<Adapter Type>] – Start of the definition section for a specific adapter type.

There can be an empty line under each [<Adapter Type>], meaning that there is no plug-in class associated, or the fully qualified name of your plug-in class can be listed.

4. Pack your adapter with the new jar file and the updated **plugins.xml** file. The remainder of the files in the package should be the same as in any adapter based on the Generic DB adapter.

Deploy the Adapter

1. In UCMDB, access the Package Manager. For details, see "Package Manager Page" in the *HP Universal CMDB Administration Guide*.

2. Click the **Deploy Packages to Server (from local disk)** icon  and browse to your adapter package. Select the package and click **Open**, then click **Deploy** to display the package in the Package Manager.

3. Select your package in the list and click the **View package resources** icon  to verify that the package contents are recognized by Package Manager.

Edit the Adapter

Once you have created and deployed the adapter, you can then edit it within UCMDB. For details, see "Adapter Management" in the *HP Universal CMDB Data Flow Management Guide*.

Create an Integration Point

In this step you check that the federation is working. That is, that the connection is valid and that the XML file is valid. However, this check does not verify that the XML is mapping to the correct fields in the RDBMS.

1. In UCMDB, access the Integration Studio (**Data Flow Management > Integration Studio**).
2. Create an integration point. For details, see New Integration Point/Edit Integration Point Dialog Box in the *HP Universal CMDB Data Flow Management Guide*.

   The Federation tab displays all CITs that can be federated using this integration point. For details, see Federation Tab in the *HP Universal CMDB Data Flow Management Guide*.

Create a View

In this step you create a view that enables you to view instances of the CIT.

1. In UCMDB, access the Modeling Studio (**Modeling > Modeling Studio**).
2. Create a view. For details, see How to Create a Pattern View in the *HP Universal CMDB Modeling Guide*.

Calculate the Results

In this step you check the results.

1. In UCMDB, access the Modeling Studio (**Modeling > Modeling Studio**).
2. Open a view.
3. Calculate results by clicking the **Calculate Query Result Count**  button.
4. Click the **Preview** button to view the CIs in the view.

View the Results

In this step you view the results and debug problems in the procedure. For example, if nothing is shown in the view, check the definitions in the **orm.xml** file; remove the relationship attributes and reload the adapter.

1. In UCMDB, access the IT Universe Manager (**Modeling > IT Universe Manager**).
2. Select a CI. The Properties tab displays the results of the federation.

View Reports

In this step you view Topology reports. For details, see Topology Reports Overview in the *HP Universal CMDB Modeling Guide*.

Enable Log Files

To understand the calculation flows, adapter lifecycle, and to view debug information, you can consult the log files. For details, see "Adapter Log Files" on page 213.

Use Eclipse to Map Between CIT Attributes and Database Tables

Caution: This procedure is intended for users with an advanced knowledge of content development. For any questions, contact HP Software Support.

This task describes how to install and use the JPA plug-in, provided with the J2EE edition of Eclipse, to:

- Enable graphical mapping between CMDB class attributes and database table columns.
- Enable manual editing of the mapping file (**orm.xml**), while providing correctness. The correctness check includes a syntax check as well as verification that the class attributes and mapped database table columns are stated correctly.
- Enable deployment of the mapping file to the CMDB server and to view the errors, as a further correctness check.
- Define a sample query on the CMDB server and run it directly from Eclipse, to test the mapping file.

Version 1.1 of the plug-in is compatible with UCMDB version 9.01 or later and Eclipse IDE for Java EE Developers, version 1.2.2.20100217-2310 or later.

This task includes the following steps:

- "Prerequisites" on the next page
- "Installation" on the next page

1. **Prerequisites**

   Install the latest update for **Java Runtime Environment (JRE) 6** on the machine where you will run Eclipse from the following site:
   http://java.sun.com/javase/downloads/index.jsp.

2. **Installation**

   a. Download and extract **Eclipse IDE for Java EE Developers** from **http://www.eclipse.org/downloads** to a local folder, for example, **C:\Program Files\eclipse**.

b. Copy **com.hp.plugin.import_cmdb_model_1.0.jar** from **C:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\bin** to **C:\Program Files\Eclipse\plugins**.

c. Launch **C:\Program Files\Eclipse\eclipse.exe**. If a message is displayed that the Java virtual machine is not found, launch **eclipse.exe** with the following command line:

```
"C:\Program Files\eclipse\eclipse.exe" -vm "<JRE installation folder>\bin"
```

3. **Prepare the Work Environment**

In this step, you set up the workspace, database, connections, and driver properties.

a. Extract the file **workspaces_gdb.zip** from **C:\hp\UCMDB\UCMDBServer\tools\db-adapter-eclipse-plugin\workspace** into **C:\Documents and Settings\All Users.**

> **Note:** You must use the exact folder path. If you unzip the file to the wrong path or leave the file unzipped, the procedure will not work.

b. In Eclipse, choose **File > Switch Workspace > Other**:

If you are working with:

- SQL Server, select the following folder: **C:\Documents and Settings\All Users\workspace_gdb_sqlserver**.
- MySQL, select the following folder: **C:\Documents and Settings\All Users\workspace_gdb_mysql**.
- Oracle, select the following folder: **C:\Documents and Settings\All Users\workspace_gdb_oracle**.

c. Click **OK**.

d. In Eclipse, display the Project Explorer view and select **<Active project> > JPA Content > persistence.xml > <active project name> > orm.xml**.

e. In the Data Source Explorer view (the bottom left pane), right-click the database connection and select the **Properties** menu.

f. In the **Properties for <Connection name>** dialog box, select **Common** and select the **Connect every time the workbench is started** check box. Select **Driver Properties** and fill in the

connection properties. Click **Test Connection** and verify that the connection is working. Click **OK**.

g. In the Data Source Explorer view, right-click the database connection and click **Connect**. A tree containing the database schemas and tables is displayed under the database connection icon.

4. **Create an Adapter**

Create an adapter using the guidelines in "Step 1: Create an Adapter" on page 30.

5. **Configure the CMDB Plug-in**

a. In Eclipse, click **UCMDB > Settings** to open the **CMDB Settings** dialog box.

b. If not already selected, select the newly created JPA project as the Active project.

c. Enter the CMDB host name, for example, **localhost** or **labm1.itdep1**. There is no need to include the port number or `http://` prefix in the address.

d. Fill in the user name and password for accessing the CMDB API, usually **admin/admin**.

e. Make sure that the **C:\hp** folder on the CMDB server is mapped as a network drive.

f. Select the base folder of the relevant adapter under **C:\hp**. The base folder is the one that contains the **dbAdapter.jar** file and the **META-INF** subfolder. Its path should be **C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase\<adapter name>**. Verify that there is no backslash (****) at the end.

6. **Import the UCMDB Class Model**

In this step, you select the CITs to be mapped as JPA entities.

a. Click **UCMDB > Import CMDB Class Model** to open the **CI Types Selection** dialog box.

b. Select the CI types that you intend to map as JPA entities. Click **OK**. The CI types are imported as Java classes. Verify that they appear under the **src** folder of the active project.

7. **Build the ORM File – Map UCMDB Classes to Database Tables**

In this step, you map the Java classes (that you imported in the previous step) to the database tables.

a. Make sure the DB connection is connected. Right-click the active project (called myProject by default) in Project Explorer. Select the JPA view, select the **Override default schema from connection** check box, and select the relevant database schema. Click **OK**.

b. Map a CIT: In the JPA Structure view, right-click the **Entity Mappings** branch and select **Add Class**. The **Add Persistent Class** dialog box opens. Do not change the **Map as** field (**Entity**).

c. Click **Browse** and select the UCMDB class to be mapped (all UCMDB classes belong to the **generic_db_adapter** package).

d. Click **OK** in both dialog boxes. The selected class is displayed under the **Entity Mappings** branch in the JPA Structure view.

> **Note:** If the entity appears without an attribute tree, right-click the active project in the Project Explorer view. Choose **Close** and then **Open**.

e. In the JPA Details view, select the primary database table to which the UCMDB class should be mapped. Leave all other fields unchanged.

8. **Map IDs**

According to JPA standards, each persistent class must have at least one ID attribute. For UCMDB classes, you can map up to three attributes as IDs. Potential ID attributes are called **id1**, **id2**, and **id3**.

To map an ID attribute:

a. Expand the corresponding class under the **Entity Mappings** branch in the JPA Structure view, right-click the relevant attribute (for example, **id1**), and select **Add Attribute to XML and Map...**.

b. The **Add Persistent Attribute** dialog box opens. Select **Id** in the **Map as** field and click **OK**.

c. In the JPA Details view, select the database table column to which the ID field should be mapped.

9. **Map Attributes**

In this step, you map attributes to the database columns.

a. Expand the corresponding class under the **Entity Mappings** branch in the JPA Structure view, right-click the relevant attribute (for example, **host_hostname**), and select **Add Attribute to XML and Map...**.

b. The **Add Persistent Attribute** dialog box opens. Select **Basic** in the **Map as** field and click **OK**.

c. In the JPA Details view, select the database table column to which the attribute field should be mapped.

10. **Map a Valid Link**

Perform the steps described above in the step "Build the ORM File – Map UCMDB Classes to Database Tables" on page 164 for mapping a UCMDB class denoting a valid link. The name of each such class takes the following structure: **<end1 entity name>_<link name>_<end 2 entity name>**. For example, a **Contains** link between a host and a location is denoted by a Java class whose name is **generic_db_adapter.host_contains_location**. For details, see "The reconciliation_rules.txt File (for backwards compatibility)" on page 190.

a. Map the ID attributes of the link class as described in "Map IDs" on the previous page. For each ID attribute, expand the **Details** check box group in the JPA Details view and clear the **Insertable** and **Updateable** check boxes.

b. Map the **end1** and **end2** attributes of the link class as follows: For each of the **end1** and **end2** attributes of the link class:

- Expand the corresponding class under the **Entity Mappings** branch in the JPA Structure view, right-click the relevant attribute (for example, **end1**), and select **Add Attribute to XML and Map...**.
- In the **Add Persistent Attribute** dialog box, select **Many to One** or **One to One** in the **Map as** field.
- Select **Many to One** if the specified **end1** or **end2** CI can have multiple links of this type. Otherwise, select **One to One**. For example, for a **host_contains_ip** link the **host** end should be mapped as **Many to One**, since one host can have multiple IPs, and the **ip** end should be mapped as **One to One**, since one IP can have only a single host.
- In the JPA Details view, select **Target entity**, for example, **generic_db_adapter.host**.
- In the **Join Columns** section of the JPA Details view, check **Override Default**. Click **Edit**. In the **Edit Join Column** dialog box, select the foreign key column of the link database table that points to an entry in the **end1**/**end2** target entity's table. If the referenced column

name in the **end1/end2** target entity's table is mapped to its ID attribute, leave the **Referenced Column Name** unchanged. Otherwise, select the name of the column to which the foreign key column points. Clear the **Insertable** and **Updatable** check boxes and click **OK**.

- If the **end1/end2** target entity has more than one ID, click the **Add** button to add additional join columns and map them in the same way as described in the previous step.

11. **Build the ORM File – Use Secondary Tables**

JPA enables a Java class to be mapped to more than one database table. For example, **Host** can be mapped to the **Device** table to enable persistence of most of its attributes and to the **NetworkNames** table to enable persistence of **host_hostName**. In this case, **Device** is the primary table and **NetworkNames** is the secondary table. Any number of secondary tables can be defined. The only condition is that there must be a one-to-one relationship between the entries of the primary and secondary tables.

12. **Define a Secondary Table**

Select the appropriate class in the JPA Structure view. In the **JPA Details** view, access the **Secondary Tables** section and click **Add**. In the **Add Secondary Table** dialog box, select the appropriate secondary table. Leave the other fields unchanged.

If the primary and the secondary table do not have the same primary keys, configure the join columns in the **Primary Key Join Columns** section of the **JPA Details** view.

13. **Map an Attribute to a Secondary Table**

You map a class attribute to a field of a secondary table as follows:

a. Map the attribute as described above in "Map Attributes" on page 165.

b. In the **Column** section of the JPA Details view, select the secondary table name in the **Table** field, to replace the default value.

14. **Use an Existing ORM File as a Base**

To use an existing **orm.xml** file as a basis for the one you are developing, perform the following steps:

a. Verify that all CITs mapped in the existing **orm.xml** file are imported into the active Eclipse project.

b. Select and copy all or part of the entity mappings from the existing file.

c. Select the **Source** tab of the **orm.xml** file in the Eclipse JPA perspective.

d. Paste all copied entity mappings under the **<entity-mappings>** tag of the edited **orm.xml** file, beneath the **<schema>** tag. Make sure that the schema tag is configured as described above in the step "Build the ORM File – Map UCMDB Classes to Database Tables" on page 164. All pasted entities now appear in the JPA Structure view. From now on, mappings can be edited both graphically and manually through the xml code of the **orm.xml** file.

e. Click **Save**.

15. Importing an Existing ORM File from an Adapter

If an adapter already exists, the Eclipse Plug-in can be used to edit its ORM file graphically. Import the **orm.xml** file into Eclipse, edit it using the plug-in and then deploy it back to the UCMDB machine. To import the ORM file, press the button on the Eclipse toolbar. A confirmation dialog is displayed. Click **OK**. The ORM file is copied from the UCMDB machine to the active Eclipse project and all relevant classes are imported from the UCMDB class model.

If the relevant classes do not appear in the JPA Structure view, right-click the active project in the Project Explorer view, choose **Close** and then **Open**.

From now on, the ORM file can be edited graphically using Eclipse, and then deployed back to the UCMDB machine as described below in "Deploy the ORM File to the CMDB" on the next page.

16. Check the Correctness of the orm.xml File – Built-in Correctness Check

The Eclipse JPA plug-in checks if any errors are present and marks them in the **orm.xml** file. Both syntax (for example, wrong tag name, unclosed tag, missing ID) and mapping errors (for example, wrong attribute name or database table field name) are checked. If there are errors, their description appears in the **Problems** view.

17. Create a New Integration Point

If no integration point exists in the CMDB for this adapter, you can create it in the Integration Studio. For details, see Integration Studio in the *HP Universal CMDB Data Flow Management Guide*.

Fill in the integration point name in the dialog box that opens. The **orm.xml** file is copied to the adapter folder. An integration point is created with all the imported CI types as its supported classes, except for multinode CITs, if they are configured in the **reconciliation_rules.txt** file. For details, see "The reconciliation_rules.txt File (for backwards compatibility)" on page 190.

18. **Deploy the ORM File to the CMDB**

    Save the **orm.xml** file and deploy it to the UCMDB server by clicking **UCMDB > Deploy ORM**. The **orm.xml** file is copied to the adapter folder and the adapter is reloaded. The operation result is shown in an **Operation Result** dialog box. If any error occurs during the reload process, the Java exception stack trace is displayed in the dialog box. If no integration point has yet been defined using the adapter, no mapping errors are detected upon deployment.

19. **Run a Sample TQL Query**

    a. Define a query (not a view) in the Modeling Studio. For details, see Modeling Studio in the *HP Universal CMDB Modeling Guide*.

    b. Create an integration point using the adapter that you created in the step "Create a New Integration Point" on the previous page. For details, see New Integration Point/Edit Integration Point Dialog Box in the *HP Universal CMDB Data Flow Management Guide*.

    c. During the creation of the adapter, verify that the CI types that should participate in the query are supported by this integration point.

    d. When configuring the CMDB plug-in, use this sample query name in the Settings dialog box. For details, see the step above "Configure the CMDB Plug-in" on page 164.

    e. Click the **Run TWL** button to run a sample TQL and verify whether it returns the required results using the newly created **orm.xml** file.

Adapter Configuration Files

The files discussed in this section are located in the **db-adapter.zip** package in the **C:\hp\UCMDB\UCMDBServer\content\adapters** folder.

This section describes the following configuration files:

- "The reconciliation_rules.txt File (for backwards compatibility)" on page 190
- "The transformations.txt File" on page 192
- "The discriminator.properties File" on page 193
- "The replication_config.txt File" on page 194
- "The fixed_values.txt File" on page 194
- "The Persistence.xml File" on page 195

General Configuration

- **adapter.conf.** The adapter configuration file. For details, see "The adapter.conf File" on the next page.

Simple Configuration

- **simplifiedConfiguration.xml**. Configuration file that replaces **orm.xml**, **transformations.txt**, and **reconciliation_rules.txt** with less capabilities. For details, see "The simplifiedConfiguration.xml File" on page 173.

Advanced Configuration

- **orm.xml**. The object-relational mapping file in which you map between CMDB CITs and database tables. For details, see "The orm.xml File" on page 175.
- **reconciliation_rules.txt**. Contains the reconciliation rules. For details, see "The reconciliation_rules.txt File (for backwards compatibility)" on page 190.
- **transformations.txt.** Transformations file in which you specify the converters to apply to convert from the CMDB value to the database value, and vice versa. For details, see "The transformations.txt File" on page 192.
- **Discriminator.properties.** This file maps each supported CI type to a comma-separated list of possible corresponding values. For details, see "The discriminator.properties File" on page 193.
- **Replication_config.txt.** This file contains a comma-separated list of CI and relationship types whose property conditions are supported by the replication plug-in. For details, see "The replication_

config.txt File" on page 194.

- **Fixed_values.txt.** This file enables you to configure fixed values for specific attributes of certain CITs. For details, see "The fixed_values.txt File" on page 194.

Hibernate Configuration

- **persistence.xml.** Used to override out-of-the-box Hibernate configurations. For details, see "The Persistence.xml File" on page 195.

Enabling Temporary Table Support for the Adapter

Enabling temporary tables allows the adapter to work more efficiently with the remote database, thus reducing stress on the database and network and also enhancing performance.

To enable temporary table support in General Database Adapter, the following conditions must be met:

- The credentials given to connect to the database, include permission to create, modify, and delete temporary tables.
- Configure the following settings in the adapter.conf configuration file:

  temp.tables.enabled=true

  performance.enable.single.sql=true

Note: Temporary tables are only supported for Microsoft SQL and Oracle.

The adapter.conf File

This file contains the following settings:

- **use.simplified.xml.config=false.true**: uses simplifiedConfiguration.xml.

  Note: Using this file means that `orm.xml`, `transformations.txt`, and `reconciliation_rules.txt` are replaced with fewer capabilities.

- **dal.ids.chunk.size=300.** Do not change this value.
- **dal.use.persistence.xml=false**. **true**: the adapter reads the Hibernate configuration from

`persistence.xml`.

> **Note:** It is not recommended to override the Hibernate configuration.

- **performance.memory.id.filtering=true**. When the GDBA executes TQLS, in some cases a large number of IDs may be retrieved and sent back to the database using SQL. To avoid this excessive work and improve performance, the GDBA attempts to read the entire view/table and filters the results in-memory.
- **id.reconciliation.cmdb.id.type=string/bytes**. When mapping the Generic DB adapter using ID Reconciliation, you can either map the **cmdb_id** to a **string** or **bytes/raw** column type by changing the **META-INF/ adapter.conf** property.
- **performance.enable.single.sql=true**. This is an optional parameter. If it does not appear in the file, its default value is **true**. When **true**, the Generic Database Adapter tries to generate a single SQL statement for each query being executed (either for population or a federated query). Using a single SQL statement improves the performance and memory consumption of the Generic Database Adapter. When **false**, the Generic Database Adapter generates multiple SQL statements, which may take longer and consume more memory than a single one. Even when this attribute is set to **true**, the adapter does not generate a single SQL statement in the following scenarios:
  - The database the adapter connects to is not on an Oracle or SQL Server.
  - The TQL being executed contains a cardinality condition other than 0..* and 1..* (for example, if there is a cardinality condition like 2..* or 0..2).
- **in.expression.size.limit=950** (default). This parameter splits the 'IN' expression of the executed SQL, when the size limit of the list of arguments is reached.
- **stringlist.delimiter.of.<CIT Name>.<Attribute Name>=<delimiter>**. In order to map a string list attribute to a database column in the generic database adapter, the attribute needs to be mapped to a string column that contains a list of concatenated values. For example, to map the attribute **policy_category** with the CI Type **policy**, and the string column contains a list of values: value1##value2##value3 (that define a list of 3 values value1, value2, value3), use the setting: **stringlist.delimiter.of.policy.policy_category=##.**
- **temp.tables.enabled=true**. Enables using temporary tables to improve performance. Only available when **performance.enable.single.sql** is enabled (only supported in Microsoft SQL and Oracle). Certain permissions in the database server may be required.
- **temp.tables.min.value=50**. Defines the number of condition values (or IDs) that are needed to use

temporary tables.

The simplifiedConfiguration.xml File

This file is used for simple mapping of UCMDB classes to database tables. To access the template for editing the file, go to **Adapter Management > db-adapter > Configuration files**.

This section includes the following topics:

The simplifiedConfiguration.xml File Template

- The **CMDB-class-name** property is the multinode type (the node to which federated CITs connect in the TQL):

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="[table_name]">
        <primary-key column-name="[column_name]"/>
```

- **reconciliation-by-two-nodes**. Reconciliation can be done using one node or two nodes. In this case example, reconciliation uses two nodes.
- **connected-node-CMDB-class-name.** The second class type needed in the reconciliation TQL.
- **CMDB-link-type.** The relationship type needed in the reconciliation TQL.
- **link-direction.** The direction of the relationship in the reconciliation TQL (from `node` to `ip_address` or from `ip_address` to `node`):

```
<reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address" CMDB-
link-type="containment" link-direction="main-to-connected">
```

  The reconciliation expression is in the form of ORs and each OR includes ANDs.

- **is-ordered.** Determines if reconciliation is done in order form or by a regular `OR` comparison.

```
<or is-ordered="true">
```

If the reconciliation property is retrieved from the main class (the multinode), use the **attribute** tag, otherwise use the **connected-node-attribute** tag.

- **ignore-case.true**: when data in the UCMDB class model is compared with data in the RDBMS, case does not matter:

  ```
  <attribute CMDB-attribute-name="name" column-name="[column_name]" ignore-
  case="true" />
  ```

  The column name is the name of the foreign key column (the column with values that point to the multinode primary key column).

  If the multinode primary key column is composed of several columns, there needs to be several foreign key columns, one for each primary key column.

  ```
  <foreign-primary-key column-name="[column_name]" CMDB-class-primary-key-column="
  [column_name]" />
  ```

  If there are few primary key columns, duplicate this column.

  ```
  <primary-key column-name="[column_name]"/>
  ```

- The **from-CMDB-converter** and **to-CMDB-converter** properties are Java classes that implement the following interfaces:

  - `com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerFromExternalDB`
  - `com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerToExternalDB`

  Use these converters if the value in the CMDB and in the database are not the same.

  In this example, `GenericEnumTransformer` is used to convert the enumerator according to the XML file that is written inside the parenthesis (**generic-enum-transformer-example.xml**):

  ```
  <attribute CMDB-attribute-name="[CMDB_attribute_name]" column-name="[column_
  name]" from-CMDB-converter="com.mercury.topaz.fcmdb.
  adapters.dbAdapter.dal.transform.impl. GenericEnumTransformer
  (generic-enum-transformer-example.xml)" to-CMDB-onverter="com.
  mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
  GenericEnumTransformer(generic-enum-transformer-example.xml)" />
  ```

```
<attribute CMDB-attribute-name="[CMDB_attribute_name]" column-name="[column_
name]" />

<attribute CMDB-attribute-name="[CMDB_attribute_name]" column-name="[column_
name]" />

 </class>

</generic-DB-adapter-config>
```

Limitations

- Can be used to map only TQL queries containing one node (in the database source). For example, you can run a `node > ticket` and a `ticket` TQL query. To bring the hierarchy of nodes from the database, you must use the advanced **orm.xml** file.
- Only one-to-many relations are supported. For example, you can bring one or more tickets on each node. You cannot bring tickets that belong to more than one node.
- You cannot connect the same class to different types of CMDB CITs. For example, if you define that `ticket` is connected to `node`, it cannot be connected to `application` as well.

The orm.xml File

This file is used for mapping CMDB CITs to database tables.

A template to use for creating a new file is located in the **C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase\GenericDBAdapter\META-INF** directory.

To edit the XML file for a deployed adapter, navigate to **Adapter Management > db-adapter > Configuration files**.

This section includes the following topics:

- "The orm.xml Schema" on page 180
- "Example of Creating the orm.xml File" on page 185
- "Configuring a Specific orm.xml for each Remote Product Version" on page 189

The orm.xml File Template

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_1_0.xsd">
    <description>Generic DB adapter orm</description>
```

Do not change the package name.

```
    <package>generic_db_adapter</package>
```

entity. The CMDB CIT name. This is the multinode entity.

Make sure that **class** includes a **generic_db_adapter.** prefix.

```
    <entity class="generic_db_adapter.node">
        <table name="[table_name]" />
```

Use a secondary table if the entity is mapped to more than one table.

```
        <secondary-table name="" />
        <attributes>
```

For a single table inheritance with discriminator, use the following code:

```
        <inheritance strategy="SINGLE_TABLE" />
        <discriminator-value>node</discriminator-value>
        <discriminator-column name="[column_name]" />
```

Attributes with tag **id** are the primary key columns. Make sure that the naming convention for these primary key columns are **idX** (id1, id2, and so on) where **X** is the column index in the primary key.

```
            <id name="id1">
```

Change only the column name of the primary key.

```
                <column updatable="false" insertable="false" name="[column_name]
" />
                <generated-value strategy="TABLE" />
            </id>
```

basic. Used to declare the CMDB attributes. Make sure to edit only **name** and **column_name** properties.

```
            <basic name="name">
                <column updatable="false" insertable="false" name="[column_name]
" />
            </basic>
```

For a single table inheritance with discriminator, map the extending classes as follows:

```
    <entity name="[cmdb_class_name]" class="generic_db_adapter.nt" name="nt">

        <discriminator-value>nt</discriminator-value>
        <attributes>
    </entity>
    <entity class="generic_db_adapter.unix" name="unix">
        <discriminator-value>unix</discriminator-value>
        <attributes>
    </entity>
    <entity name="[CMDB_class_name]" class="generic_db_adapter.[CMDB[cmdb_class_
name]">
        <table name="[default_table_name]" />
        <secondary-table name="" />
        <attributes>
            <id name="id1">
                <column updatable="false" insertable="false" name="[column_name]
" />
                <generated-value strategy="TABLE" />
            </id>
            <id name="id2">
                <column updatable="false" insertable="false" name="[column_name]
" />
                <generated-value strategy="TABLE" />
            </id>
            <id name="id3">
                <column updatable="false" insertable="false" name="[column_name]
" />
                <generated-value strategy="TABLE" />
            </id>
```

The following example shows a CMDB attribute name with no prefix:

```
            <basic name="[CMDB_attribute_name]">
```

```
                <column updatable="false" insertable="false" name="[column_name]
" />
            </basic>
            <basic name="[CMDB_attribute_name]">
                <column updatable="false" insertable="false" name="[column_name]
" />
            </basic>
            <basic name="[CMDB_attribute_name]">
                <column updatable="false" insertable="false" name="[column_name]
" />
            </basic>
        </attributes>
    </entity>
```

This is a relationship entity. The naming convention is **end1Type_linkType_end2Type**. In this example **end1Type** is **node** and the **linkType** is **composition**.

```
    <entity name="node_composition_[CMDB_class_name]" class="generic_db_
adapter.node_composition_[CMDB_class_name]">
        <table name="[default_table_name]" />
        <attributes>
            <id name="id1">
                <column updatable="false" insertable="false" name="[column_name]
" />
                <generated-value strategy="TABLE" />
            </id>
```

The target entity is the entity that this property is pointing to. In this example, **end1** is mapped to **node** entity.

many-to-one. Many relationships can be connected to one node.

join-column. The column that contains **end1** IDs (the target entity IDs).

referenced-column-name. The column name in the target entity (**node**) that contain the IDs that are used in the join column.

```
            <many-to-one target-entity="node" name="end1">
                <join-column updatable="false" insertable="false" referenced-
column-name="[column_name]" name="[column_name]" />
            </many-to-one>
```

one-to-one. One relationship can be connected to one **[CMDB_class_name]**.

```
            <one-to-one target-entity="[CMDB_class_name]" name="end2">
                <join-column updatable="false" insertable="false" referenced-
column-name="" name="[column_name]" />
```

```
            </one-to-one>
        </attributes>
    </entity>
</entity-mappings>
```

node attribute. This is an example of how to add a node attribute.

```
<entity class="generic_db_adapter.host_node">
  <discriminator-value>host_node</discriminator-value>
    <attributes/>
</entity>
<entity class="generic_db_adapter.nt">
  <discriminator-value>nt</discriminator-value>
    <attributes>
      <basic name="nt_servicepack">
        <column updatable="false" insertable="false" name="specific_type_value"/>
      </basic>
    </attributes>
</entity>
```

Multiple ORM files

Multiple mapping files are supported. Each mapping file name should end with **orm.xml**. All mapping files should be placed under the META-INF folder of the adapter.

Naming Conventions

- In each entity, the class property must match the name property with the prefix of `generic_db_adapter`.
- Primary key columns must take names of the form **idX** where **X** = **1**, **2**, ..., according to the number of primary keys in the table.
- Attribute names must match class attribute names even as regards case.
- The relationship name takes the form `end1Type_linkType_end2Type`.

- CMDB CITs, which are also reserved words in Java, should be prefixed by **gdba_**. For example, for the CMDB CIT **goto**, the ORM entity should be named **gdba_goto**.

Using Inline SQL Statements Instead of Table Names

You can map entities to inline `select` clauses instead of to database tables. This is equivalent to defining a view in the database and mapping an entity to this view. For example:

```
    <entity class="generic_db_adapter.node">
        <table name="(select d.id as id1, d.name as name , d.os as host_os from
Device d)" />
```

In this example, the node attributes should be mapped to columns id1, name, and host_os, rather than id, name, and os.

The following limitations apply:

- The inline SQL statement is available only when using Hibernate as the JPA provider.
- Round brackets around the inline SQL select clause are mandatory.
- The **<schema>** element should not be present in the **orm.xml** file. In the case of Microsoft SQL Server 2005, this means that all table names should be prefixed with `dbo.`, rather than defining them globally by `<schema>dbo</schema>`.

The orm.xml Schema

The following table explains the common elements of the **orm.xml** file. The complete schema can be found at http://java.sun.com/xml/ns/persistence/orm_1_0.xsd. The list is not complete, and it mainly explains the specific behavior of the standard Java Persistence API for the Generic Database Adapter.

Element Name and Path	Description	Attributes
entity-mappings	The root element for the entity mapping document. This element should be exactly the same as the one given in the GDBA sample files.	

Element Name and Path	Description	Attributes
description (entity-mappings)	A free text description of the entity mapping document. (Optional)	
package (entity-mappings)	The name of the Java package that will contain the mapping classes. Should always contain the text `generic_db_adapter`.	1. **Name:** name **Description:** The name of the UCMDB CI type to which this entity is mapped. If this is entity is mapped to a link in the CMDB, the name of the entity should be in the format `<end_1>_<link_name>_<end_2>`. For example, `node_composition_cpu` defines an entity that will be mapped to the composition link between a node and a CPU. If the name of the CI type is the same as the name of the Java class without the package prefix, this field can be omitted. **Is required?:** Optional **Type:** String 2. **Name:** class **Description:** The fully qualified name of the Java class that will be created for this DB entity. The name of the Java class' package should be the same as the name given in the `package` element. You may not use Java reserved words, such as interface or switch, as the class name. Instead, add the prefix `gdba_` to the name (so interface will be `generic_db_adapter.gdba_interface`. **Is required?:** Required **Type:** String

Element Name and Path	Description	Attributes
table (entity-mappings>entity)	This element defines the primary table of the DB entity. Can only appear once. Required.	**Name:** name **Description:** The name of the primary table. If the name of the table does not contain the schema to which it belongs, the table will be searched only in the schema of the user that was used to create the integration point. This can also be any valid SELECT statement. If this is a SELECT statement, it must be encapsulated with parentheses. **Is required?:** Required **Type:** String
secondary-table (entity-mappings > entity)	This element may be used to define a secondary table for the DB entity. This table must be connected to the primary table with a 1-to-1 relationship. You may define more than one secondary table. Optional.	**Name:** name **Description:** The name of the secondary table. If the name of the table does not contain the schema to which it belongs, the table will be searched only in the schema of the user that was used to create the integration point. This can also be any valid SELECT statement. If this is a SELECT statement, it must be encapsulated with parentheses. **Is required?:** Required **Type:** String
primary-key-join-column (entity-mappings > entity > secondary-table)	If the secondary table and primary table are not connected using fields with the same name, this element defines the name of the primary key field in the secondary table that needs to be connected to the primary key field of the primary table.	**Name:** name **Description:** The name of the primary key field in the secondary table. If this element does not exist, it is assumed that the primary key field has the same name as the primary key field of the primary table. **Is required?:** Optional **Type:** String

Element Name and Path	Description	Attributes
inheritance (entity-mappings > entity)	If the current entity is the parent entity for a family of DB entities, then use this element to mark it as such. Optional.	**Name:** strategy **Description:** Defines the way the inheritance is implemented in your DB. **Is required?:** Required **Type:** One of the following values: • SINGLE_TABLE: This entity and all child entities exist in the same table. • JOINED: The child entities are in joined tables. • TABLE_PER_CLASS: Each entity is completely defined by a separate table.
discriminator-column (entity-mappings > entity)	If the inheritance is of type SINGLE_TABLE, this element is used to define the name of the field used to determine the type of entity for each row.	**Name:** name **Description:** The name of the discriminator column. **Is required?:** Required **Type:** String
discriminator-value (entity-mappings > entity)	This element defines the type of the specific entity in the inheritance tree. This name needs to be the same as the name defined in the **discriminator.properties** file for the value group of this specific entity type.	
attributes (entity-mappings > entity)	The root element for all of the attribute mappings for an entity.	

Element Name and Path	Description	Attributes
id (entity-mappings > entity attributes)	This element defines the key field for the entity. There must be at least one id field defined. If more than one id element exists, its fields create a compound key for the entity. You should try and avoid compound keys for CI entities (not for links).	**Name:** name **Description:** A string of type idX, where X is a number between 1 and 9. The first id should be marked as id1, the second as id2 and so on. This is NOT the name of the key attribute in UCMDB. **Is required?:** Required **Type:** String
basic (entity-mappings > entity attributes)	This element defines a mapping between a field in the table, which is not part of the table's primary key, and a UCMDB attribute.	**Name:** name **Description:** The name of the UCMDB attribute to which the field is mapped. This attribute must exist in the UCMDB CI type to which the current entity is mapped. **Is required?:** Required **Type:** String
column (entity-mappings > entity > attributes >id -OR- (entity-mappings > entity > attributes > basic)	Defines the name of the column in the table for basic mapping or an id field.	1. **Name:** name **Description:** The name of the field. **Is required?:** Required **Type:** String 2. **Name:** table **Description:** The name of the table to which the field belongs. This must be either the primary table or one of the secondary tables defined for the entity. If this attribute is omitted, it is assumed that the field belongs to the primary table. **Is required:** Optional **Type:** String

Element Name and Path	Description	Attributes
one-to-one (entity-mappings > entity > attributes)	Defines a column whose value is in another table, and the two tables are connected using a one-to-one relationship. This element is only supported for link entity mappings and not for other CI types. This is the only way to define a mapping between a table and a UCMDB link.	1. **Name:** name **Description:** Which of the two ends this field represents. **Is required?:** Required **Type:** Either `end1` or `end2` 2. **Name:** target-entity **Description:** The name of the entity to which the end refers. **Is required?:** Required **Type:** One of the entity names defined in the entity mapping document.
join-column (entity-mappings > entity attributes > one-to-one)	Defines the way to join the target-entity defined in the parent one-to-one element and the current entity.	1. **Name:** name **Description:** The name of the field in the current table that will be used to perform the one-to-one join. **Is required?:** Required **Type:** String 2. **Name:** name **Description:** The name of a field in the joint entity by which to perform the join. If this attribute is omitted, it is assumes that the joint table has a column with the same name as the field defined in the name attribute. **Is required?:** Optional **Type:** String

Example of Creating the orm.xml File

The example presented here demonstrates how to create the **orm.xml file**. In this example, SQL tables in a remote database are mapped to CI types in UCMDB.

Given tables with the following format in the remote database, populate the **Hosts** table with nodes, the **IP_Addresses** table with IP Addresses, and create links between the nodes and IP Addresses as follows:

Hosts Table

host_name	host_id
Test1	1
Test2	2
Test3	3

IP_Addresses Table

ip_address	ip_id
10.1.1.1	1
10.2.2.2	2
10.3.3.2	3
10.4.4.4	4

Host_IP_Link Table (links between Nodes and IP Addresses)

host_id	ip_id
1	1
2	2
2	3
3	4

The primary key for the **Hosts** table is the **host_id** field and the primary key in the **IP_Addresses Table** table is the **ip_id** field. In the **Host_IP_Link** table the **host_id** and the **ip_id** are foreign keys from the **Hosts Table** and **IP_Addresses Table**.

According to the tables above, create the **orm.xml** file according to the following steps. The entities used in this example are **node**, **ip_address**, and **node_containment_ip_address**

1. Create the **node** entity by mapping the **host_id** from the **Hosts** table as follows:

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
```

```
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
       /orm_1_0.xsd">
    <description>test_integration</description>
    <package>generic_db_adapter</package>
    <entity class="generic_db_adapter.node">
       <table name="Hosts"/>
       <attributes>
        <id name="id1">
                <column updatable="false" insertable="false" name=
                       "host_id"/>
                <generated-value strategy="TABLE"/>
        </id>
        <basic name="name">
                <column updatable="false" insertable="false" name=
                       "host_name"/>
        </basic>
       </attributes>
    </entity>
```

The entity **class** must be a CI Type that already exists in UCMDB. The table **name** is the table within the database that contains both an ID and the Host information. The ID attribute is required to identify specific hosts and is used later in the mapping. In this example, the **name** attribute of this entity is populated with the **host_name** column in the hosts table.

2. For the next entity, map the IP Addresses from the Interfaces table:

```
<entity name="ip_address" class="generic_db_adapter.ip_address">
    <table name="IP_Addresses"/>
    <attributes>
       <id name="id1">
        <column insertable="false" updatable="false" name="ip_id"/>
        <generated-value strategy="TABLE"/>
       </id>
```

```
        <basic name="name">
         <column updatable="false" insertable="false" name="ip_address"/>
        </basic>
    </attributes>
</entity>
```

3. Next the link between the Node and the IP Address must be created by means of the mapping table, and reference the **ip_id** field (though it could reference both the **host_id** and **ip_id** fields if desired).

```
<entity name="node_containment_ip_address"
        class="generic_db_adapter.node_containment_ip_address">
    <table name="Host_IP_Link"/>
    <attributes>
       <id name="id1">
        <column updatable="false" insertable="false" name="ip_id"/>
        <generated-value strategy="TABLE"/>
       </id>
       <many-to-one target-entity="node" name="end1">
        <join-column name="host_id"/>
       </many-to-one>
       <one-to-one target-entity="ip_address" name="end2">
        <join-column name="ip_id"/>
       </one-to-one>
    </attributes>
</entity>
```

The entity name for the container has the format: [end1 CIT]_[link CIT]_[end2 CIT]. So for this example, since the link CI Type is **containment**, the entity name for the container is: **node_containment_ip_address** and the entity class is **generic_db_adapter.node_containment_ip_address**. The ID is required in this block of code, and while this example works with a single ID of the Interface, both columns could reference id1 and id2. The code for that would be:

```
<id name="id1">
    <column updatable="false" insertable="false" name="ip_id"/>
    <generated-value strategy="TABLE"/>
</id>
<id name="id2">
    <column updatable="false" insertable="false" name="host_id"/>
    <generated-value strategy="TABLE"/>
</id>
```

The two ends of this link are 'many-to-one' and 'one-to-one', meaning each IP Address will be linked to 1 node, but a node may be linked to many IP Addresses. The columns included are from the Links table and reference the Hosts and Interfaces tables.

Configuring a Specific orm.xml for each Remote Product Version

It is possible to configure a specific **orm.xml** file so that the adapter uses a specific **orm.xml** for a given remote product version. For example, if the remote data store has two product versions x and y, for each version there can be a different mapping of entities.

To configure a specific orm.xml file per remote product version:

1. Add a parameter to the **adapter.xml** file called **version** and specify possible version values as **valid-values**.
2. In the adapter package, under the META-INF folder, create a folder called **VersionOrm**.
3. In the **VersionOrm** folder, create an **orm.xml** file for each specific version. The filename should contain the version prefix. For example, if the version is called **x**, the filename should be **x_ orm.xml**.

Note: The **orm.xml** file in the META-INF folder is loaded for any remote product version, regardless of whether you create a specific **orm.xml** file for a remote product version. It can have entities that are mapped in the same manner for all versions.

The reconciliation_types.txt File

As of UCMDB 10.00, the **reconciliation_types.txt** file is no longer relevant. Any CIT can be used for reconciliation. The federation engine automatically executes the mapping.

The reconciliation_rules.txt File (for backwards compatibility)

This file is used to configure the reconciliation rules if you want to perform reconciliation when the DBMappingEngine is configured in the adapter. If you do not use the DBMappingEngine, the generic UCMDB reconciliation mechanism is used and there is no need to configure this file.

Each row in the file represents a rule. For example:

```
multinode[node] expression[^node.name OR ip_address.name] end1_type[node]
end2_type[ip_address] link_type[containment]
```

The multinode is filled with the multinode name (the CMDB CIT that is connected to the federated database CIT in the TQL query).

This expression includes the logic that decides whether two multinodes are equal (one multinode in the CMDB and the other in the database source).

The expression is composed of `ORs` or `ANDs`.

The convention regarding attribute names in the expression part is `[className].[attributeName]`. For example, `attributeName` in the `ip_address` class is written `ip_address.name`.

For an ordered match (if the first `OR` sub-expression returns an answer that the multinodes are not equal, the second OR sub-expression is not compared), use `ordered expression` instead of `expression`.

To ignore case during a comparison, use the control (**^**) sign.

The parameters `end1_type`, `end2_type` and `link_type` are used only if the reconciliation TQL query contains two nodes and not just a multinode. In this case, the reconciliation TQL query is `end1_type > (link_type) > end2_type`.

There is no need to add the relevant layout as it is taken from the expression.

Types of Reconciliation Rules

Reconciliation rules take the form of OR and AND conditions. You can define these rules on several different nodes (for example, node is identified by `name from node`AND/OR`name from ip_address`).

The following options find a match:

- **Ordered match.** The reconciliation expression is read from left to right. Two `OR` sub-expressions are considered equal if they have values and they are equal. Two `OR` sub-expressions are considered not equal if both have values and they are not equal. For any other case there is no decision, and the

next OR sub-expression is tested for equality.

name from node OR from ip_address. If both the CMDB and the data source include name and they are equal, the nodes are considered as equal. If both have name but they are not equal, the nodes are considered not equal without testing the name of ip_address. If either the CMDB or the data source is missing name of node, the name of ip_address is checked.

- **Regular match.** If there is equality in one of the OR sub-expressions, the CMDB and the data source are considered equal.

  name from node OR from ip_address. If there is no match on name of node, name of ip_address is checked for equality.

For complex reconciliations, where the reconciliation entity is modeled in the class model as several CITs with relationships (such as node), the mapping of a superset node includes all relevant attributes from all modeled CITs.

Note: As a result, there is a limitation that all reconciliation attributes in the data source should reside in tables that share the same primary key.

Another limitation states that the reconciliation TQL query should have no more than two nodes. For example, the node > ticket TQL query has a node in the CMDB and a ticket in the data source.

To reconcile the results, name must be retrieved from the node and/or ip_address.

If the name in the CMDB is in the format of *.m.com, a converter can be used from CMDB to the federated database, and vice versa, to convert these values.

The node_id column in the database ticket table is used to connect between the entities (the defined association can also be made in a node table):

DB Node	
PK	node_id
	name

DB IP_Address	
PK	ip_id
	name

DB Ticket	
PK	ticket_id
	node_id

Note: The three tables must be part of the federated RDBMS source and not the CMDB database.

The transformations.txt File

This file contains all the converter definitions.

The format is that each line contains a new definition.

The transformations.txt File Template

```
entity[[CMDB_class_name]] attribute[[CMDB_attribute_name]] to_DB_class
[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.
transform.impl.GenericEnumTransformer(generic-enum-transformer-example.xml)]
from_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)]
```

entity. The entity name as it appears in the `orm.xml` file.

attribute. The attribute name as it appears in the `orm.xml` file.

to_DB_class. The full, qualified name of a class that implements the interface **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.FcmdbDalTransformerToExternalDB**. The elements in the parenthesis are given to this class constructor. Use this converter to transform CMDB values to database values, for example, to append the suffix of **.com** to each node name.

from_DB_class. The full, qualified name of a class that implements the **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.**

FcmdbDalTransformerFromExternalDB interface. The elements in the parenthesis are given to this class constructor. Use this converter to transform database values to CMDB values, for example, to append the suffix of **.com** to each node name.

For details, see "Out-of-the-Box Converters" on page 198.

The discriminator.properties File

This file maps each supported CI type (that is also used as a discriminator value in orm.xml) to a comma-separated list of possible corresponding values of the discriminator column, or a condition to match possible values of the discriminator column.

If a condition is used, use the syntax: `like(condition)`, where `condition` is a string that can contain the following wildcards:

- **%** (percent sign) - allows you to match any string of any length (including a zero length string)
- **_** (underscore) - allows you to match a single character

For example, `like(%unix%)` will match unix, linux, unix-aix, and so on. Like conditions may only be applied to string columns.

You can also have a single discriminator value mapped to any value that does not belong to another discriminator by stating `'all-other'`.

If the adapter you are creating uses discriminator capabilities, you must define all the discriminator values in the **discriminator.properties** file.

Example of Discriminator Mapping:

For example, the adapter supports the CI types `node`, `nt`, and `unix`, and the database contains a single table named `t_nodes` that contains a column called **type**. If the type is 10001, the row represents a node; if the type is 10004, it represents a unix machine, and so on. The **discriminator.properties** file might look like this:

```
node=10001, 10005
nt=10002,10003
unix=2%
mainframe=all-other
```

The **orm.xml** file includes the following code:

```
    <entity class="generic_db_adapter.node" >
        <table name="t_nodes" />
        ...
```

```
        <inheritance strategy="SINGLE_TABLE" />
        <discriminator-value>node</discriminator-value>
        <discriminator-column name="type" />
        ...
    </entity>
    <entity class="generic_db_adapter.nt" name="nt">
        <discriminator-value>nt</discriminator-value>
        <attributes>
    </entity>
    <entity class="generic_db_adapter.unix" name="unix">
        <discriminator-value>unix</discriminator-value>
        <attributes>
    </entity>
```

The discriminator_column attribute is then calculated as follows:

- If **type** contains `10002` or `10003` for a certain entry, the entry is mapped to the **nt** CIT.
- If **type** contains `10001` or `10005` for a certain entry, the entry is mapped to the **node** CIT.
- If **type** starts with `2` for a certain entry, the entry is mapped to the **unix** CIT.
- Any other value in the **type** column is mapped to the **mainframe** CIT.

> **Note:** The **node** CIT is also the parent of **nt** and **unix**.

The replication_config.txt File

This file contains a comma-separated list of CI and relationship types whose property conditions are supported by the replication plug-in. For details, see "Plug-ins" on page 204.

The fixed_values.txt File

This file enables you to configure fixed values for specific attributes of certain CITs. In this way, each of these attributes can be assigned a fixed value that is not stored in the database.

The file should contain zero or more entries of the following format:

```
entity[<entityName>] attribute[<attributeName>] value[<value>]
```

For example:

```
entity[ip_address] attribute[ip_domain] value[DefaultDomain]
```

The file also supports a list of constants. To define a constants list, use the following syntax:

```
entity[<entityName>] attribute[<attributeName>] value[{<Val1>, <Val2>, <Val3>, ...
}]
```

The Persistence.xml File

This file is used to override the default Hibernate settings and to add support for database types that are not out of the box (OOB database types are Oracle Server, Microsoft SQL Server, and MySQL).

If you need to support a new database type, make sure that you supply a connection pool provider (the default is `c3p0`) and a JDBC driver for your database (put the `*.jar` files in the adapter folder).

To see all available Hibernate values that can be changed, check the **org.hibernate.cfg.Environment** class (for details, refer to http://www.hibernate.org.)

Example of the persistence.xml File:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
    <!-- Don't change this value -->
    <persistence-unit name="GenericDBAdapter">
        <properties>
            <!--  Don't change this value  -->
            <property name="hibernate.archive.autodetection" value="class,
               hbm" />
            <!--The driver class name"/-->
            <property name="hibernate.connection.driver_class" value="com.
              mercury.jdbc.MercOracleDriver" />
            <!--The connection url"/-->
            <property name="hibernate.connection.url" value="jdbc:mercury:
              oracle://artist:1521;sid=cmdb2" />
            <!--DB login credentials"/-->
            <property name="hibernate.connection.username" value="CMDB" />
            <property name="hibernate.connection.password" value="CMDB" />
            <!--connection pool properties"/-->
            <property name="hibernate.c3p0.min_size" value="5" />
            <property name="hibernate.c3p0.max_size" value="20" />
            <property name="hibernate.c3p0.timeout" value="300" />
            <property name="hibernate.c3p0.max_statements" value="50" />
            <property name="hibernate.c3p0.idle_test_period" value="3000" />
            <!--The dialect to use-->
            <property name="hibernate.dialect" value="org.hibernate.dialect.
               OracleDialect" />
        </properties>
    </persistence-unit>
</persistence>
```

Connect to Database Using NT Authentication

It is possible to connect to an MS SQL Server that requires NT authentication. To do so a driver that can parse domain is needed (that is, jTDS JDBC Driver).

The authentication is done according to the given parameters (domain, username, password), and not with the current running process NT credentials.

1. In the **persistence.xml** edit the following properties as follows:

```
<!--The driver class name"/-->
<property name="hibernate.connection.driver_class"
value="net.sourceforge.jtds.jdbc.Driver"/>
<property name="hibernate.connection.url" value="jdbc:jtds:sqlserver://[host
name]:[port];DatabaseName=[database name];domain=[the domain]"/>
<!--DB login credentials"/-->
<property name="hibernate.connection.username" value="[username]"/>
<property name="hibernate.connection.password" value="[password]"/>
```

2. Place the JDBC driver file under: **<probe installation folder>\lib**.

3. Restart the probe.

Configure the Persistence.xml File for the SCCM Integration to Use NTLM Authentication

Note: This section applies to the SCCM integration only.

In order for the SCCM integration to use NTLM authentication, configure the **persistence.xml** file as follows:

1. Place the JDBC driver file under: **<probe installation folder>\lib**.

   For example, you can put the **jtds-1.3.1.jar** file from http://sourceforge.net/projects/jtds/files/ in the **DataFlowProbe\lib** folder.

2. Start the server and the probe.

3. In UCMDB, go to **Data Flow Management > Adapter Management > SCCMAdapter >** .

4. In the Resources pane, select the SCCM adapter configuration file in the **Packages > SCCMAdapter > Configuration Files** folder.

5. In the **adapter.conf** file, set **dal.use.persistence.xml=true**.

6. In the **persistence.xml** file, add the following:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
    <persistence-unit name="GenericDBAdapter">
        <properties>
            <!-- added to fix: org.hibernate.HibernateException:
'hibernate.dialect' must be set when no Connection avalable -->
            <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>

            <!--The driver class name"/-->
            <property name="hibernate.connection.driver_class"
value="net.sourceforge.jtds.jdbc.Driver"/>
            <property name="hibernate.connection.url"
value="jdbc:jtds:sqlserver://<DB_host>:<port>;DatabaseName=<DB_
name>;domain=<domain_name> "/>
        </properties>
    </persistence-unit>
</persistence>
```

Note: Replace the highlighted part with your connection URL.

7. No user or password is required in the **persistence.xml** file.

8. Go to **Data Flow Management > Integration Studio**, and click the **New Integration Point**  button.

9. Provide values for required fields.

   When you are required to enter credential ID, do the following:

   a. In the Choose Credential dialog box, select **Generid DB Protocol (SQL)** from the left Protocol pane.

b. In the right Credentials pane, click the **Create new connection details for selected protocol type**  button.

c. In the new dialog box, select **MicrosoftSQLServerNTLM**as database type.

d. Enter a port number.

e. Provide a username in the following format: **domain\username**.

f. Provide a password.

Out-of-the-Box Converters

You can use the following converters (transformers) to convert federated queries and replication jobs to and from database data.

This section includes the following topics:

The enum-transformer Converter

This converter uses an XML file that is given as an input parameter.

The XML file maps between hard-coded CMDB values and database values (enums). If one of the values does not exist, you can choose to return the same value, return null, or throw an exception.

The transformer performs a comparison between two strings using a case sensitive, or a case insensitive method. The default behavior is case sensitive. To define it as case insensitive use: `case-sensitive="false"` in the enum-transformer element.

Use one XML mapping file for each entity attribute.

Note: This converter can be used for both the `to_DB_class` and `from_DB_class` fields in the **transformations.txt** file.

Input File XSD:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="enum-transformer">
    <xs:complexType>
        <xs:sequence>
           <xs:element ref="value" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="db-type" use="required">
           <xs:simpleType>
               <xs:restriction base="xs:string">
                   <xs:enumeration value="integer"/>
                   <xs:enumeration value="long"/>
                   <xs:enumeration value="float"/>
                   <xs:enumeration value="double"/>
                   <xs:enumeration value="boolean"/>
                   <xs:enumeration value="string"/>
                   <xs:enumeration value="date"/>
                   <xs:enumeration value="xml"/>
                   <xs:enumeration value="bytes"/>
              </xs:restriction>
           </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="cmdb-type" use="required">
           <xs:simpleType>
              <xs:restriction base="xs:string">
                  <xs:enumeration value="integer"/>
                  <xs:enumeration value="long"/>
```

```
                <xs:enumeration value="float"/>
                <xs:enumeration value="double"/>
                <xs:enumeration value="boolean"/>
                <xs:enumeration value="string"/>
                <xs:enumeration value="date"/>
                <xs:enumeration value="xml"/>
                <xs:enumeration value="bytes"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="non-existing-value-action" use="required">
           <xs:simpleType>
               <xs:restriction base="xs:string">
                  <xs:enumeration value="return-null"/>
                  <xs:enumeration value="return-original"/>
                  <xs:enumeration value="throw-exception"/>
               </xs:restriction>
           </xs:simpleType>
       </xs:attribute>
       <xs:attribute name="case-sensitive" use="optional">
           <xs:simpleType>
               <xs:restriction base="xs:boolean">
               </xs:restriction>
           </xs:simpleType>
       </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="value">
  <xs:complexType>
      <xs:attribute name="cmdb-value" type="xs:string" use="required"/>
```

```
        <xs:attribute name="external-db-value" type="xs:string" use="required"/>
        <xs:attribute name="is-cmdb-value-null" type="xs:boolean"
use="optional"/>
        <xs:attribute name="is-db-value-null" type="xs:boolean" use="optional"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Example of Converting 'sys' Value to 'System' Value:

In this example, `sys` value in the CMDB is transformed into `System` value in the federated database, and `System` value in the federated database is transformed into `sys` value in the CMDB.

If the value does not exist in the XML file (for example, the string `demo`), the converter returns the same input value it receives.

```
<enum-transformer CMDB-type="string" DB-type="string" non-existing-value-
action="return-original" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/generic-enum-transformer.xsd">
    <value CMDB-value="sys" external-DB-value="System" />
</enum-transformer>
```

Example of Converting an External or CMDB Value to a Null Value:

In this example, a value of **NNN** in the remote database is transformed into a null value in the CMDBdatabase.

```
<value cmdb-value="null" is-cmdb-value-null="true" external-db-value="NNN"/>
```

In this example, the value **OOO** in the CMDB is transformed into a null value in the remote database.

```
<value cmdb-value="OOO" external-db-value="null" is-db-value-null="true"/>
```

The SuffixTransformer Converter

This converter is used to add or remove suffixes from the CMDB or federated database source value.

There are two implementations:

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdb AddSuffixTransformer**. Adds the suffix (given as input) when converting from federated database value to CMDB value and removes the suffix when converting from CMDB value to federated database value.

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdb RemoveSuffixTransformer**. Removes the suffix (given as input) when converting from federated database value to CMDB value and adds the suffix when converting from CMDB value to federated database value.

The PrefixTransformer Converter

This converter is used to add or remove a prefix from the CMDB or federated database value.

There are two implementations:

- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdb AddPrefixTransformer**. Adds the prefix (given as input) when converting from federated database value to CMDB value and removes the prefix when converting from CMDB value to federated database value.
- **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.AdapterToCmdb RemovePrefixTransformer**. Removes the prefix (given as input) when converting from federated database value to CMDB value and adds the prefix when converting from CMDB value to federated database value.

The BytesToStringTransformer Converter

This converter is used to convert byte arrays in the CMDB to their string representation in the federated database source.

The converter is:
com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.CmdbToAdapterBytes ToStringTransformer.

The StringDelimitedListTransformer Converter

This converter is used to transform a single string list to an integer/string list in the CMDB.

The converter is: **com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl. StringDelimitedListTransformer.**

The Custom Converter

It is possible to write your own custom converter (transformer) from scratch. This enables you to create any converter required for your needs.

There are two ways to write a custom converter:

1. Write a compiled Java converter

   a. Create a Java Project in a Java IDE (such as Eclipse, Intellij, or Netbeans).

   b. Add the federation-api.jar and db-interfaces.jar to your class path.

   c. Create a Java class that implements the following interfaces (from **db-interfaces.jar**):

      - FcmdbDalTransformerFromExternalDB
      - FcmdbDalTransformerValuesToExternalDB
      - FcmdbDalTransformerInit

   d. Compile the project and create a jar file.

   e. Place the jar file in the adapter's package (under adapterCode\<Adapter ID>)

   f. Deploy the package.

   g. Add the new converter class name to the **transformations.txt** file.

2. Write a Groovy (script based) converter

   An example is found in the original GDBA package, **GroovyExampleTransformer.groovy**.

   a. Create a Groovy file in the adapter's package (under adapterCode\<Adapter ID>). You can do this directly using the Adapter Management menu.

   b. Create a Groovy class that implements the following interfaces (from **db-interfaces.jar**):

      - FcmdbDalTransformerFromExternalDB
      - FcmdbDalTransformerValuesToExternalDB
      - FcmdbDalTransformerInit

   c. Add the new converter Groovy class name to the **transformations.txt** file accordingly.

   Note: Groovy is a scripting language that extends Java. Regular Java code is valid Groovy code as well.

Plug-ins

The generic database adapter supports the following plugins:

- An optional plug-in for full topology synchronization.
- An optional plug-in for synchronizing changes in topology. If no plug-in for synchronizing changes is implemented, it is possible to perform a differential synchronization, but that synchronization will actually be a full one.
- An optional plug-in for synchronizing layout.
- An optional plug-in to retrieve supported queries for synchronization. If this plugin is not defined, all TQL names are returned.
- An internal, optional plug-in to change the TQL definition and TQL result.
- An internal, optional plug-in to change a layout request and CIs result.
- An internal, optional plug-in to change a layout request and relationships result.
- An internal, optional plug-in to change the action of push Back IDs.

For details about implementing and deploying plug-ins, see "Implement a Plug-in" on page 156.

Configuration Examples

This section gives examples of configurations.

This section includes the following topics:

Use Case

A TQL query is:

node > (composition) > card

where:

- **node** is the CMDB entity
- **card** is the federated database source entity
- **composition** is the relationship between them

The example is run against the `ED` database. `ED nodes` are stored in the `Device` table and `card` is stored in the `hwCards` table. In the following examples, `card` is always mapped in the same manner.

Single Node Reconciliation

In this example the reconciliation is run against the `name` property.

Simplified Definition

The reconciliation is done by `node` and it is emphasized by the special tag **CMDB-class**.

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID" />
        <reconciliation-by-single-node>
            <or>
                <attribute CMDB-attribute-name="name" column-name="Device_Name"
/>
            </or>
        </reconciliation-by-single-node>
    </CMDB-class>
    <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-
class-name="node" link-class-name="composition">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-
column="Device_ID
        <primary-key column-name="hwCards_Seq" />
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"
/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
```

```
    </class>
</generic-DB-adapter-config>
```

Advanced Definition

The orm.xml File

Pay attention to the addition of the relationship mapping. For details, see the definition section in "The orm.xml File" on page 175.

Example of the orm.xml File:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/
persistence/orm http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
version="1.0">
       <description>Generic DB adapter orm</description>
       <package>generic_db_adapter</package>
       <entity class="generic_db_adapter.node" >
               <table name="Device"/>
               <attributes>
                       <id name="id1">
                               <column name="Device_ID"
                                  insertable="false"
                                  updatable="false"/>
                               <generated-value strategy="TABLE"/>
                       </id>
                       <basic name="name">
                               <column name="Device_Name"/>
                       </basic>
               </attributes>
       </entity>
       <entity class="generic_db_adapter.card" >
               <table name="hwCards"/>
               <attributes>
                       <id name="id1">
                               <column name="hwCards_Seq" insertable="false"
                                   updatable="false"/>
                               <generated-value strategy="TABLE"/>
                       </id>
                       <basic name="card_class">
                               <column name="hwCardClass" insertable="false"
                                   updatable="false"/>
                       </basic>
```

```
                        <basic name="card_vendor">
                                <column name="hwCardVendor" insertable="false"
                                   updatable="false"/>
                        </basic>
                        <basic name="card_name">
                                <column name="hwCardName" insertable="false"
                                   updatable="false"/>
                        </basic>
                </attributes>
        </entity>
        <entity class="generic_db_adapter.node_composition_card" >
                <table name="hwCards"/>
                <attributes>
                        <id name="id1">
                                <column name="hwCards_Seq" insertable="false"
                                   updatable="false"/>
                                <generated-value strategy="TABLE"/>
                        </id>
                        <many-to-one name="end1" target-entity="node">
                                <join-column name="Device_ID" insertable="false"
                                   updatable="false"/>
                        </many-to-one>
                        <one-to-one name="end2" target-entity="card"
>                               <join-column name="hwCards_Seq"
                                referenced-column-name="hwCards_Seq" insertable=
                                "false" updatable="false"/>
                        </one-to-one>
                </attributes>
        </entity>
</entity-mappings>
```

The reconciliation_rules.txt File

For details, see "The reconciliation_rules.txt File (for backwards compatibility)" on page 190.

```
multinode[node] expression[node.name]
```

The transformation.txt File

This file remains empty as no values need to be converted in this example.

Two Node Reconciliation

In this example, reconciliation is calculated according to the `name` property of `node` and of `ip_address` with different variations.

The reconciliation TQL query is **node > (containment) > ip_address**.

Simplified Definition

The reconciliation is by `name` of `node` OR of `ip_address`:

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID" />
        <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment">
            <or>
                <attribute CMDB-attribute-name="name" column-name="Device_Name"
/>
                <connected-node-attribute CMDB-attribute-name="name" column-
name="Device_PreferredIPAddress" />
            </or>
        </reconciliation-by-two-nodes>
    </CMDB-class>
    <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-
class-name="node" link-class-name="containment">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-
column="Device_ID" />
        <primary-key column-name="hwCards_Seq" />
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"
/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
    </class>
</generic-DB-adapter-config>
```

The reconciliation is `name` of `node` AND of `ip_address`:

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID" />
        <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment">
            <and>
                <attribute CMDB-attribute-name="name" column-name="Device_Name"
/>
                <connected-node-attribute CMDB-attribute-name="name" column-
name="Device_PreferredIPAddress" />
            </and>
        </reconciliation-by-two-nodes>
    </CMDB-class>
```

```
    <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-
class-name="node" link-class-name="containment">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-
column="Device_ID" />
        <primary-key column-name="hwCards_Seq" />
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"
/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
    </class>
</generic-DB-adapter-config>
```

The reconciliation is by `name` of `ip_address`:

```
<?xml version="1.0" encoding="UTF-8"?>
<generic-DB-adapter-config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/simplifiedConfiguration.xsd">
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID" />
        <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
CMDB-link-type="containment">
            <or>
                <connected-node-attribute CMDB-attribute-name="name" column-
name="Device_PreferredIPAddress" />
            </or>
        </reconciliation-by-two-nodes>
    </CMDB-class>
    <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-
class-name="node" link-class-name="containment">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-
column="Device_ID" />
        <primary-key column-name="hwCards_Seq" />
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"
/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
    </class>
</generic-DB-adapter-config>
```

Advanced Definition

The orm.xml File

Since the reconciliation expression is not defined in this file, the same version should be used for any reconciliation expression.

The reconciliation_rules.txt File

For details, see "The reconciliation_rules.txt File (for backwards compatibility)" on page 190.

```
multinode[node] expression[ip_address.name OR node.name] end1_type[node] end2_
type[ip_address] link_type[containment]

multinode[node] expression[ip_address.name AND node.name] end1_type[node] end2_
type[ip_address] link_type[containment]

multinode[node] expression[ip_address.name] end1_type[node] end2_type[ip_
address] link_type[containment]
```

The transformation.txt File

This file remains empty as no values need to be converted in this example.

Using a Primary Key that Contains More Than One Column

If the primary key is composed of more than one column, the following code is added to the XML definitions:

Simplified Definition

There is more than one primary key tag and for each column there is a tag.

```
    <class CMDB-class-name="card" default-table-name="hwCards" connected-CMDB-
class-name="node" link-class-name="containment">
        <foreign-primary-key column-name="Device_ID" CMDB-class-primary-key-
column="Device_ID" />
        <primary-key column-name="Device_ID" />
        <primary-key column-name="hwBusesSupported_Seq" />
        <primary-key column-name="hwCards_Seq" />
        <attribute CMDB-attribute-name="card_class" column-name="hwCardClass" />
        <attribute CMDB-attribute-name="card_vendor" column-name="hwCardVendor"
/>
        <attribute CMDB-attribute-name="card_name" column-name="hwCardName" />
    </class>
```

Advanced Definition

The orm.xml File

A new `id` entity is added that maps to the primary key columns. Entities that use this `id` entity must add a special tag.

If you use a foreign key (join-column tag) for such a primary key, you must map between each column in the foreign key to a column in the primary key.

For details, see "The orm.xml File" on page 175.

Example of the orm.xml File:

```
<entity class="generic_db_adapter.card" >
        <table name="hwCards" />
        <attributes>
            <id name="id1">
                <column name="Device_ID" insertable="false" updatable="false" />
                <generated-value strategy="TABLE" />
            </id>
            <id name="id2">
                <column name="hwBusesSupported_Seq" insertable="false"
updatable="false" />
                <generated-value strategy="TABLE" />
            </id>
            <id name="id3">
                <column name="hwCards_Seq" insertable="false" updatable="false"
/>
                <generated-value strategy="TABLE" />
            </id>

<entity class="generic_db_adapter.node_containment_card" >
        <table name="hwCards" />
        <attributes>
            <id name="id1">
                <column name="Device_ID" insertable="false" updatable="false" />
                <generated-value strategy="TABLE" />
            </id>
            <id name="id2">
                <column name="hwBusesSupported_Seq" insertable="false"
updatable="false" />
                <generated-value strategy="TABLE" />
            </id>
            <id name="id3">
                <column name="hwCards_Seq" insertable="false" updatable="false"
/>
                <generated-value strategy="TABLE" />
            </id>
            <many-to-one name="end1" target-entity="node">
                <join-column name="Device_ID" insertable="false"
updatable="false" />
            </many-to-one>
            <one-to-one name="end2" target-entity="card">
                <join-column name="Device_ID" referenced-column-name="Device_ID"
insertable="false" updatable="false" />
                <join-column name="hwBusesSupported_Seq" referenced-column-
name="hwBusesSupported_Seq" insertable="false" updatable="false" />
                <join-column name="hwCards_Seq" referenced-column-name="hwCards_
Seq" insertable="false" updatable="false" />
            </one-to-one>
        </attributes>
```

```
    </entity>
</entity-mappings>
```

Using Transformations

In the following example, the generic **enum** transformer is converted from values 1, 2, 3 to values a, b, c respectively in the name column.

The mapping file is generic-enum-transformer-example.xml.

```
<enum-transformer CMDB-type="string" DB-type="string" non-existing-value-
action="return-original" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../META-CONF/generic-enum-transformer.xsd">
    <value CMDB-value="1" external-DB-value="a" />
    <value CMDB-value="2" external-DB-value="b" />
    <value CMDB-value="3" external-DB-value="c" />
</enum-transformer>
```

Simplified Definition

```
    <CMDB-class CMDB-class-name="node" default-table-name="Device">
        <primary-key column-name="Device_ID" />
        <reconciliation-by-two-nodes connected-node-CMDB-class-name="ip_address"
          CMDB-link-type="containment">
            <or>
                <attribute CMDB-attribute-name="name" column-name="Device_Name"
                 from-CMDB-
converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.
                 transform.impl.GenericEnumTransformer(generic-enum-transformer-
example.
                 xml)" to-CMDB-
converter="com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.
                 transform.impl.GenericEnumTransformer(generic-enum-transformer-
example.
                 xml)" />
                <connected-node-attribute CMDB-attribute-name="name"
                 column-name="Device_PreferredIPAddress" />
            </or>
        </reconciliation-by-two-nodes>
    </CMDB-class>
```

Advanced Definition

There is a change only to the **transformation.txt** file.

The transformation.txt File

Make sure that the attribute names and entity names are the same as in the orm.xml file.

```
entity[node] attribute[name]
to_DB_class[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)] from_DB_class
[com.mercury.topaz.fcmdb.adapters.dbAdapter.dal.transform.impl.
GenericEnumTransformer(generic-enum-transformer-example.xml)]
```

Adapter Log Files

To understand the calculation flows and adapter lifecycle, and to view debug information, you can consult the following log files.

This section includes the following topics:

- "Log Levels" below
- "Log Locations" below

Log Levels

You can configure the log level for each of the logs.

In a text editor, open the **C:\hp\UCMDB\UCMDBServer\conf\log\fcmdb.gdba.properties** file.

The default log level is **ERROR**:

```
#loglevel can be any of DEBUG INFO WARN ERROR FATAL
loglevel=ERROR
```

- To increase the log level for all log files, change **loglevel=ERROR** to **loglevel=DEBUG** or **loglevel=INFO**.
- To change the log level for a specific file, change the specific **log4j** category line accordingly. For example, to change the log level of fcmdb.gdba.dal.sql.log to **INFO**, change:

  ```
  log4j.category.fcmdb.gdba.dal.SQL=${loglevel},fcmdb.gdba.dal.SQL.appender
  ```

  to:

  ```
  log4j.category.fcmdb.gdba.dal.SQL=INFO,fcmdb.gdba.dal.SQL.appender
  ```

Log Locations

The log files are located in the **C:\hp\UCMDB\UCMDBServer\runtime\log** directory.

- **Fcmdb.gdba.log**

  The adapter lifecycle log. Gives details about when the adapter started or stopped, and which CITs are supported by this adapter.

  Consult for initiation errors (adapter load/unload).

- **fcmdb.log**

  Consult for exceptions.

- **cmdb.log**

  Consult for exceptions.

- **Fcmdb.gdba.mapping.engine.log**

  The mapping engine log. Gives details about the reconciliation TQL query that the mapping engine uses, and the reconciliation topologies that are compared during the connect phase.

  Consult this log when a TQL query gives no results even though you know there are relevant CIs in the database, or the results are unexpected (check the reconciliation).

- **Fcmdb.gdba.TQL.log**

  The TQL log. Gives details about the TQL queries and their results.

  Consult this log when a TQL query does not return results and the mapping engine log shows that there are no results in the federated data source.

- **Fcmdb.gdba.dal.log**

  The DAL lifecycle log. Gives details about CIT generation and database connection details.

  Consult this log when you cannot connect to the database or when there are CITs or attributes that are not supported by the query.

- **Fcmdb.gdba.dal.command.log**

  The DAL operations log. Gives details about internal DAL operations that are called. (This log is similar to `cmdb.dal.command.log`).

- **Fcmdb.gdba.dal.SQL.log**

The DAL SQL queries log. Gives details about called JPAQLs (object oriented SQL queries) and their results.

Consult this log when you cannot connect to the database or when there are CITs or attributes that are not supported by the query.

- **Fcmdb.gdba.hibrnate.log**

  The Hibernate log. Gives details about the SQL queries that are run, the parsing of each JPAQL to SQL, the results of the queries, data regarding Hibernate caching, and so on. For details on Hibernate, see "Hibernate as JPA Provider" on page 136.

External References

For details on the JavaBeans 3.0 specification, see http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

Troubleshooting and Limitations – Developing Generic Database Adapters

This section describes troubleshooting and limitations for the generic database adapter.

General Limitations

When you update an adapter package, use Notepad++, UltraEdit, or some other third-party text editor rather than Notepad (any version) from Microsoft Corporation to edit the template files. This prevents the use of special symbols, which cause the deployment of the prepared package to fail.

JPA Limitations

- All tables must have a primary key column.
- CMDB class attribute names must follow the JavaBeans naming convention (for example, names must start with lower case letters).
- Two CIs that are connected with one relationship in the class model must have direct association in the database (for example, if `node` is connected to `ticket` there must be a foreign key or linkage

table that connects them).

- Several tables that are mapped to the same CIT must share the same primary key table.

Functional Limitations

- You cannot create a manual relationship between the CMDB and federated CITs. To be able to define virtual relationships, a special relationship logic must be defined (it can be based on properties of the federated class).
- Federated CITs cannot be trigger CITs in an impact rule, but they can be included in an impact analysis TQL query.
- A federated CIT can be part of an enrichment TQL, but cannot be used as the node on which enrichment is performed (you cannot add, update, or delete the federated CIT).
- Using a class qualifier in a condition is not supported.
- Subgraphs are not supported.
- Compound relationships are not supported.
- The external CI CMDB`id` is composed from its primary key and not its key attributes.
- A column of type `bytes` cannot be used as a primary key column in Microsoft SQL Server.
- TQL query calculation fails if attribute conditions that are defined on a federated node have not had their names mapped in the **orm.xml** file.

Chapter 6: Developing Java Adapters

This chapter includes:

Federation Framework Overview

Note:

- The term **relationship** is equivalent to the term **link**.
- The term **CI** is equivalent to the term **object**.
- A graph is a collection of nodes and links.

The Federation Framework functionality uses an API to retrieve information from federated sources. The Federation Framework provides three main capabilities:

- **Federation** on the fly. All queries are run over original data repositories and results are built on the fly in the CMDB.
- **Population.** Populates data (topological data and CI properties) to the CMDB from an external data

source.

- **Data Push.** Pushes data (topological data and CI properties) from the local CMDB to a remote data source.

All action types require an adapter for each data repository, which can provide the specific capabilities of the data repository and retrieve and/or update the required data. Every request to the data repository is made through its adapter.

This section also includes the following topics:

- "Federation on the Fly" below
- "Data Push" on page 220
- "Population" on page 220

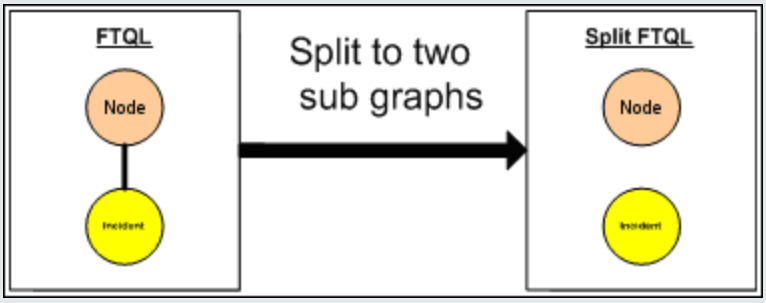
Federation on the Fly

Federated TQL queries enables data retrieval from any external data repository without replicating its data.

A federated TQL query uses adapters that represent external data repositories, to create appropriate external relationships between CIs from different external data repositories and the UCMDB CIs.

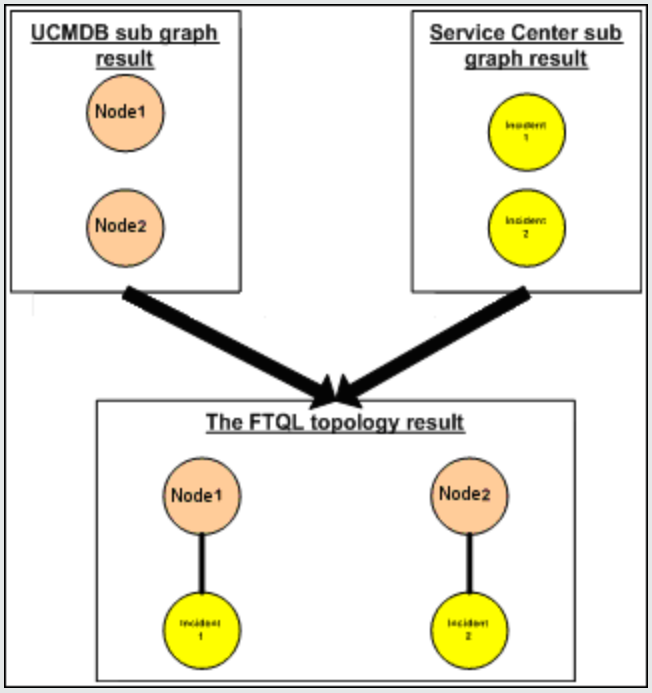
Example of Federation-on-the-Fly Flow:

1. The Federation Framework splits a federated TQL query into several subgraphs, where all nodes in a subgraph refer to the same data repository. Each subgraph is connected to the other subgraphs by a virtual relationship (but itself contains no virtual relationships).

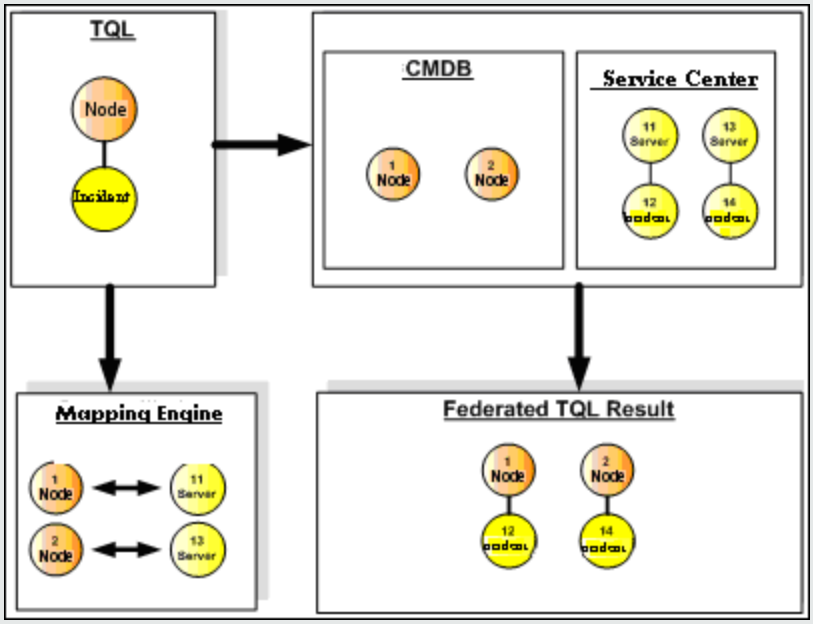


2. After the federated TQL query is split into subgraphs, the Federation Framework calculates each subgraph's topology and connects two appropriate subgraphs by creating virtual

relationships between the appropriate nodes.



3. After the federated TQL topology is calculated, the Federation Framework retrieves a layout for the topology result.



Data Push

You use the data push flow to synchronize data from your current local CMDB to a remote service or target data repository.

In data push, data repositories are divided into two categories: source (local CMDB) and target. Data is retrieved from the source data repository and updated to the target data repository. The data push process is based on query names, meaning that data is synchronized between the source (local CMDB) and target data repositories, and is retrieved by a TQL query name from the local CMDB.

The data push process flow includes the following steps:

1. Retrieving the topology result with signatures from the source data repository.
2. Comparing the new results with the previous results.
3. Retrieving a full layout (that is, all CI properties) of CIs and relationships, for changed results only.
4. Updating the target data repository with the received full layout of CIs and relationships. If any CIs or relationships are deleted in the source data repository and the query is exclusive, the replication process removes the CIs or relationships in the target data repository as well.

The CMDB has 2 hidden data sources (**hiddenRMIDataSource** and **hiddenChangesDataSource**), which are always the `source' data source in data push flows. To implement a new adapter for data push flows, you only have to implement the `target' adapter.

Population

You use the population flow to populate the CMDB with data from external sources.

The flow always uses one 'source' data source to retrieve the data, and pushes the retrieved data to the Probe in a similar process to the flow of a discovery job.

To implement a new adapter for population flows, you only have to implement the source adapter, since the Data Flow Probe acts as the target.

The adapter in the population flow is executed on the Probe. Debugging and logging should be done on the Probe and not on the CMDB.

The population flow is based on query names, that is, data is synchronized between the source data repository and the Data Flow Probe, and is retrieved by a query name in the source data repository. For example, in UCMDB, the query name is the name of the TQL query. However, in another data repository the query name can be a code name that returns data. The adapter is designed to correctly handle the query name.

Each job can be defined as an exclusive job. This means that the CIs and relationships in the job results are unique in the local CMDB, and no other query can bring them to the target. The adapter of the source data repository supports specific queries, and can retrieve the data from this data repository. The adapter of the target data repository enables the update of retrieved data on this data repository.

SourceDataAdapter Flow

- Retrieves the topology result with signatures from the source data repository.
- Compares the new results with the previous results.
- Retrieves a full layout (that is, all CI properties) of CIs and relationships, for changed results only.
- Updates the target data repository with the received full layout of CIs and relationships. If any CIs or relationships are deleted in the source data repository and the query is exclusive, the replication process removes the CIs or relationships in the target data repository as well.

SourceChangesDataAdapter Flow

- Retrieves the topology result that occurred since the last date given.
- Retrieves a full layout (that is, all CI properties) of CIs and relationships, for changed results only.
- Updates the target data repository with the received full layout of CIs and relationships. If any CIs or relationships are deleted in the source data repository and the query is exclusive, the replication process removes the CIs or relationships in the target data repository as well.

PopulateDataAdapter Flow

- Retrieves the full topology with requested layout result.
- Uses the topology chunk mechanism to retrieve the data in chunks.
- The probe filters out any data that was already brought in earlier runs
- Updates the target data repository with the received layout of CIs and relationships. If any CIs or relationships are deleted in the source data repository and the query is exclusive, the replication process removes the CIs or relationships in the target data repository as well.

PopulateChangesDataAdapter Flow

- Retrieves the topology with requested layout result that has changes since the last run.
- Uses the topology chunk mechanism to retrieve the data in chunks.

- The probe filters out any data that was already brought in earlier runs (including this flow).
- Updates the target data repository with the received layout of CIs and relationships. If any CIs or relationships are deleted in the source data repository and the query is exclusive, the replication process removes the CIs or relationships in the target data repository as well.

Instance-Based Population Flow

If the adapter is defined to support an instance-based flow (by means of the **<instance-based-data>** tag, as described in "XML Configuration Tags and Properties" on page 249), the population engine automatically finds removed CIs inside the instance and removes them from the UCMDB (assuming deletion is allowed for the specific population job). Each instance must have a Root CI, marked in the TQL definition with the name **Root**. Each time a root CI is passed, its entire instance (all the CIs connected to it) are compared to the last time it was sent to UCMDB, and any CIs that were connected to the root but are now not connected to it are deleted from UCMDB. For the adapter to correctly support instance-based flow, any change to any CI or attribute in the entire instance must trigger a resend of the entire instance to UCMDB.

Adapter and Mapping Interaction with the Federation Framework

An adapter is an entity in UCMDB that represents external data (data that is not saved in UCMDB). In federated flows, all interactions with external data sources are performed through adapters. The Federation Framework interaction flow and adapter interfaces are different for replication and for federated TQL queries.

This section also includes the following topics:

- "Adapter Lifecycle" below
- "Adapter assist Methods" on the next page

Adapter Lifecycle

An adapter instance is created for each external data repository. The adapter begins its lifecycle with the first action applied to it (such as, `calculate TQL` or `retrieve/update data`). When the **start** method is called, the adapter receives environmental information, such as the data repository configuration, logger, and so on. The adapter lifecycle ends when the data repository is removed from the configuration, and the **shutdown** method is called. This means that the adapter is stateful and can

contain the connection to the external data repository if it is required.

Adapter assist Methods

The adapter has several `assist` methods that can add external data repository configurations. These methods are not part of the adapter lifecycle and create a new adapter each time they are called.

- The first method tests the connection to the external data repository for a given configuration. testConnection can be executed either on the UCMDB server or the Data Flow Probe, depending on the type of adapter.
- The second method is relevant only for the source adapter and returns the supported queries for replication. (This method is executed on the Probe only.)
- The third method is relevant only for federation and population flows, and returns supported external classes by the external data repository. (This method is executed on the UCMDB server.)

All these methods are used when you create or view integration configurations.

Federation Framework for Federated TQL Queries

This section includes the following topics:

- "Definitions and Terms" below
- "Mapping Engine" on the next page
- "Federated Adapter" on the next page

See "Interactions between the Federation Framework, Server, Adapter, and Mapping Engine" on page 225 for diagrams illustrating the interactions between the Federation Framework, UCMDB, adapter, and Mapping Engine.

Definitions and Terms

Reconciliation data. The rule for matching CIs of the specified type that are received from the CMDB and the external data repository. The reconciliation rule can be of three types:

- **ID reconciliation.** This can be used only if the external data repository contains the CMDB ID of reconciliation objects.

- **Property reconciliation.** This is used when the matching can be done by properties of the reconciliation CI type only.
- **Topology reconciliation.** This is used when you need the properties of additional CITs (not only of the reconciliation CIT) to perform a match on reconciliation CIs. For example, you can perform reconciliation of the node type by the `name` property that belongs to the `ip_address` CIT.

Reconciliation object. The object is created by the adapter according to received reconciliation data. This object should refer to an external CI and is used by the Mapping Engine to connect between the external CIs and the CMDB CIs.

Reconciliation CI type. The type of CIs that represent reconciliation objects. These CIs must be stored in both the CMDB and in the external data repositories.

Mapping engine. A component that identifies relations between CIs from different data repositories that have a virtual relationship between them. The identification is performed by reconciling CMDB reconciliation objects and external CI reconciliation objects.

Mapping Engine

Federation Framework uses the Mapping Engine to calculate the federated TQL query. The Mapping Engine connects between CIs that are received from different data repositories and are connected by virtual relationships. The Mapping Engine also provides reconciliation data for the virtual relationship. One end of the virtual relationship must refer to the CMDB. This end is a `reconciliation` type. For the calculation of the two subgraphs, a virtual relationship can start from any end node.

Federated Adapter

The Federated adapter brings two kinds of data from external data repositories: external CI data and reconciliation objects that belong to external CIs.

- **External CI data.** The external data that does not exist in the CMDB. It is the target data of the external data repository.
- **Reconciliation object data.** The auxiliary data that is used by the federation framework to connect between CMDB CIs and external data. Each reconciliation object should refer to an External CI. The type of reconciliation object is the type (or subtype) of one of the virtual relationship ends from which data is retrieved. Reconciliation objects should fit the adapter received to reconciliation data. The reconciliation object can be one of three types: `IdReconciliationObject`, `PropertyReconciliationObject`, or `TopologyReconciliationObject`.

In the DataAdapter-based interfaces (DataAdapter, PopulateDataAdapter, and PopulateChangesDataAdapter), the reconciliation is requested as part of the query definition.

Interactions between the Federation Framework, Server, Adapter, and Mapping Engine

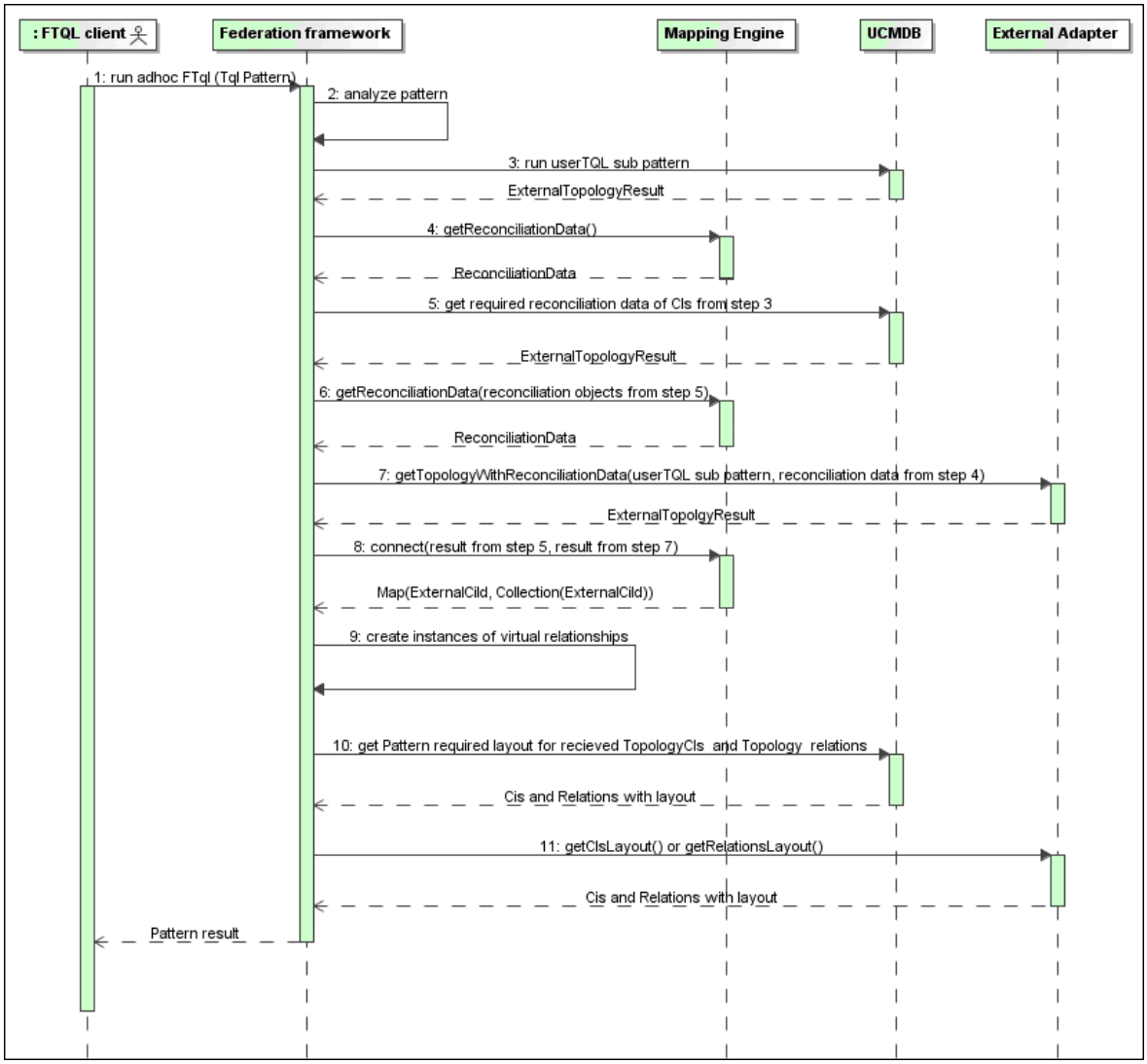
The following diagrams illustrate the interactions between the Federation Framework, UCMDB Server, the adapter, and the Mapping Engine. The federated TQL query in the example diagrams has only one virtual relationship, so that only the UCMDB and one external data repository are involved in the federated TQL query.

This section includes the following topics:

In the first diagram the calculation begins in the UCMDB and in the second diagram in the external adapter. Each step in the diagram includes references the appropriate method call of the adapter or mapping engine interface.

Calculation Starts at the Server End

The following sequence diagram illustrates the interaction between the Federation Framework, UCMDB, the adapter, and the Mapping Engine. The federated TQL query in the example diagram has only one virtual relationship, so that only UCMDB and one external data repository are involved in the federated TQL query.

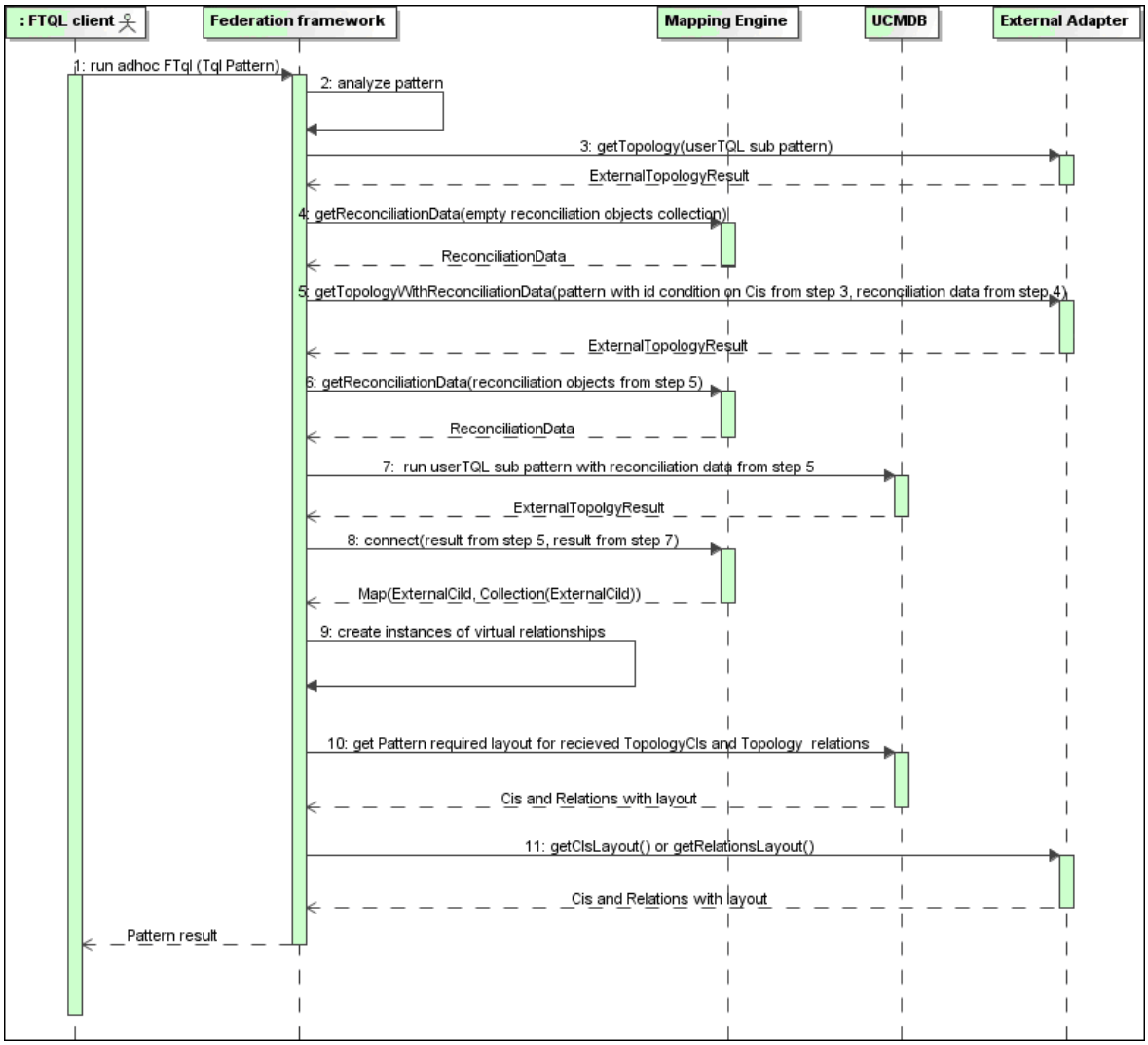


The numbers in this image are explained below:

Number	Explanation
1	The Federation Framework receives a call for a federated TQL calculation.
2	The Federation Framework analyzes the adapter, finds the virtual relationship, and divides the original TQL into two sub-adapters–one for UCMDB and one for the external data repository.
3	The Federation Framework requests the topology of the sub-TQL from UCMDB.

Number	Explanation
4	After receiving the topology results, the Federation Framework calls the appropriate Mapping Engine for the current virtual relationship and requests reconciliation data. The `reconciliationObject` parameter is empty at this stage, that is, no condition is added to reconciliation data in this call. The returned reconciliation data defines which data is needed to match the reconciliation CIs in UCMDB to the external data repository. The reconciliation data can be one of the following types: • **IdReconciliationData.** CIs are reconciled according to their ID. • **PropertyReconciliationData.** CIs are reconciled according to the properties of one of the CIs. • **TopologyReconciliationData.** CIs are reconciled according to the topology (for example, to reconcile node CIs, the IP address of **IP** is required too).
5	The Federation Framework requests reconciliation data for the CIs of the virtual relationship ends that were received in step "3" on the previous page from UCMDB.
6	The Federation Framework calls the Mapping Engine to retrieve the reconciliation data. In this state (by contrast with step "3" on the previous page), the Mapping Engine receives the reconciliation objects from step "5" above as parameters. The Mapping Engine translates the received reconciliation object to the condition on the reconciliation data.
7	The Federation Framework requests the topology of the sub-TQL from the external data repository. The external adapter receives the reconciliation data from step "6" above as a parameter.
8	The Federation Framework calls the Mapping Engine to connect between the received results. The `firstResult` parameter is the external topology result received from UCMDB in step "5" above and the `secondResult` parameter is the external topology result received from the External Adapter in step "7" above. The Mapping Engine returns a map where External CI ID from the first data repository (UCMDB in this case) is mapped to the External CI IDs from the second (external) data repository.
9	For each mapping, the Federation Framework creates a virtual relationship.
10	After the calculation of the federated TQL query results (only at the topology stage), the Federation Framework retrieves the original TQL layout for the resulting CIs and relationships from the appropriate data repositories.

Calculation Starts at the External Adapter End



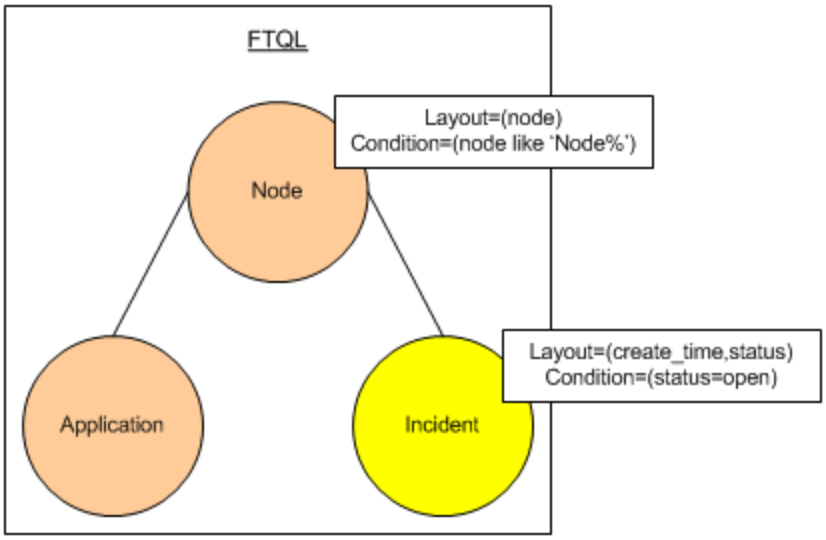
The numbers in this image are explained below:

Number	Explanation
1	The Federation Framework receives a call for an federated TQL calculation.
2	The Federation Framework analyzes the adapter, finds the virtual relationship, and divides the original TQL into two sub-adapters – one for UCMDB and one for the external data repository.

Number	Explanation
3	The Federation Framework requests the topology of the sub-TQL from the External Adapter. The returned `ExternalTopologyResult` is not supposed to contain any reconciliation object, since the reconciliation data is not part of the request.
4	After receiving the topology results, the Federation Framework calls the appropriate Mapping Engine with the current virtual relationship and requests reconciliation data. The `reconciliationObjects` parameter is empty at this state, that is, no condition is added to the reconciliation data in this call. The returned reconciliation data defines what data is needed to match the reconciliation CIs in UCMDB to the external data repository. The reconciliation data can be one of three following types: • **IdReconciliationData.** CIs are reconciled according to their ID. • **PropertyReconciliationData.** CIs are reconciled according to the properties of one of the CIs. • **TopologyReconciliationData.** CIs are reconciled according to the topology (for example, to reconcile node CIs, the IP address of **IP** is required too).
5	The Federation Framework requests reconciliation objects for the CIs that were received in step 3 from the external data repository. The Federation Framework calls the **getTopologyWithReconciliationData()** method in the External Adapter, where the requested topology is a one-node topology with CIs received in step 3 as the ID condition and reconciliation data from step 4.
6	The Federation Framework calls the Mapping Engine to retrieve the reconciliation data. In this state (by contrast with step 3), the Mapping Engine receives the reconciliation objects from step 5 as parameters. The Mapping Engine translates the received reconciliation object to the condition on the reconciliation data.
7	The Federation Framework requests the topology of the sub-TQL with reconciliation data from step 6 from UCMDB.
8	The Federation Framework calls the Mapping Engine to connect between the received results. The `firstResult` parameter is the external topology result received from the External Adapter at step 5 and the `secondResult` parameter is the external topology result received from UCMDB at step 7. The Mapping Engine returns a map where the External CI ID from the first data repository (the external data repository in this case) is mapped to the External CI IDs from the second data repository (UCMDB).
9	For each mapping, the Federation Framework creates a virtual relationship.
10	After the calculation of the federated TQL query results (only at the topology stage), the Federation Framework retrieves the original TQL layout for the resulting CIs and relationships from the appropriate data repositories.

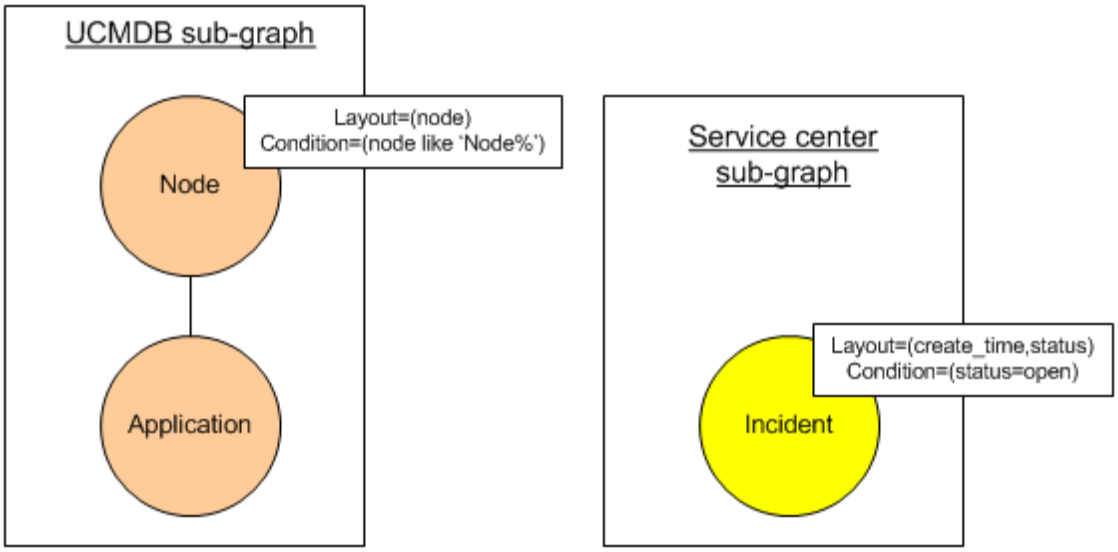
Example of Federation Framework Flow for Federated TQL Queries

This example explains how to view all open incidents on specific nodes. The ServiceCenter data repository is the external data repository. The node instances are stored in UCMDB, and the incident instances are stored in ServiceCenter. It is assumed that to connect the incident instances to the appropriate node, the `node` and `ip_address` properties of the host and IP are needed. These are reconciliation properties that identify the nodes from ServiceCenter in UCMDB.

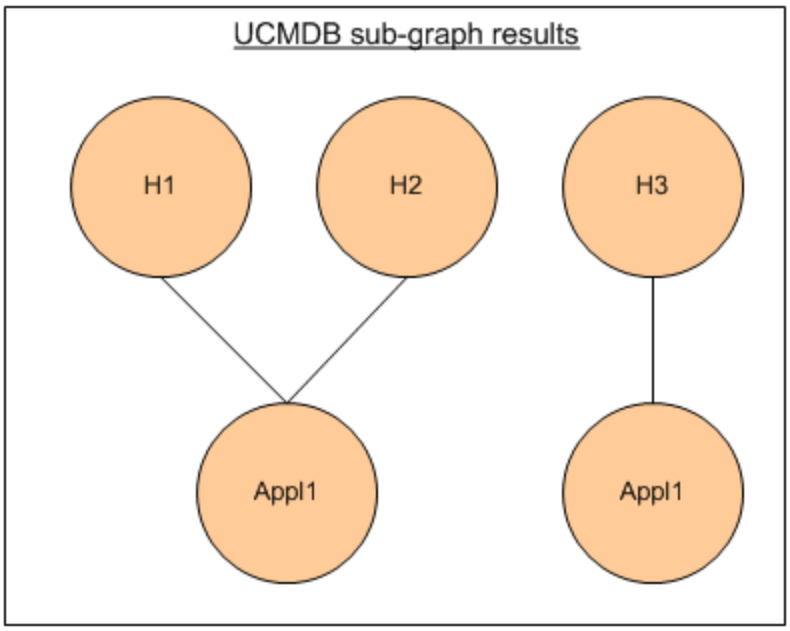


Note: For attribute federation, the adapter's **getTopology** method is called. The reconciliation data is adapted in the user TQL (in this case, the CI element).

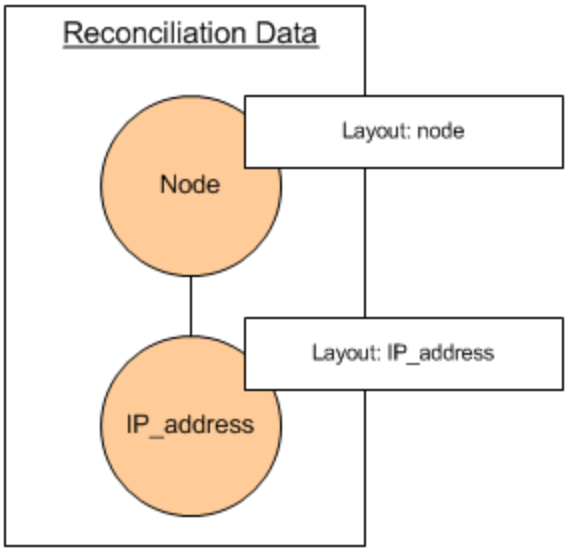
1. After analyzing the adapter, the Federation Framework recognizes the virtual relationship between `Node` and `Incident` and splits the federated TQL query into two subgraphs:



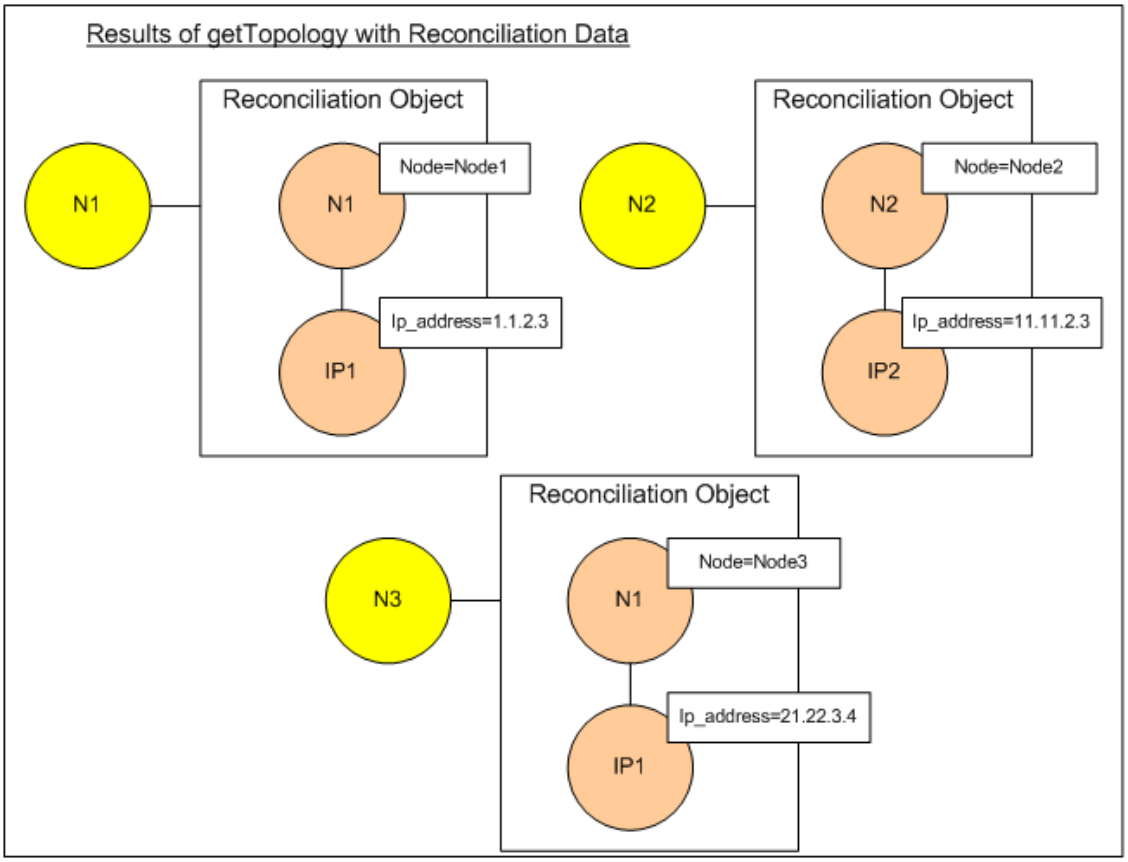
2. The Federation Framework runs the UCMDB subgraph to request the topology, and receives the following results:



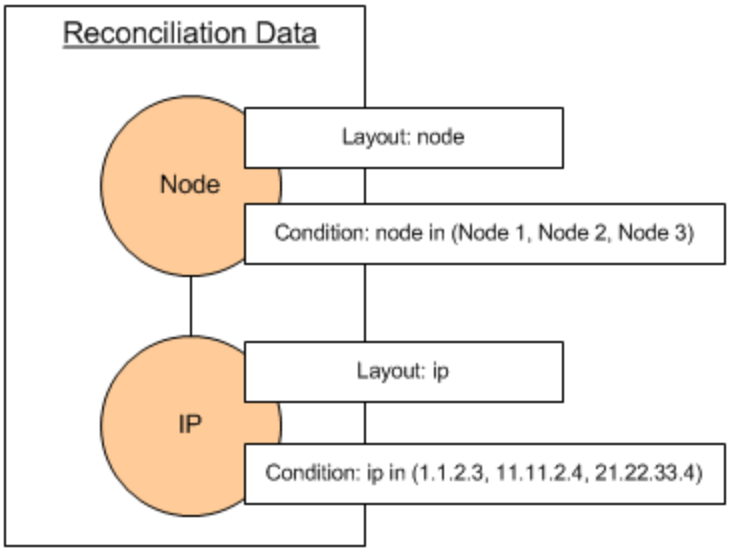
3. The Federation Framework requests, from the appropriate Mapping Engine, the reconciliation data for the first data repository (UCMDB) that contains the information to connect between received data from two data repositories. The reconciliation data in this case is:



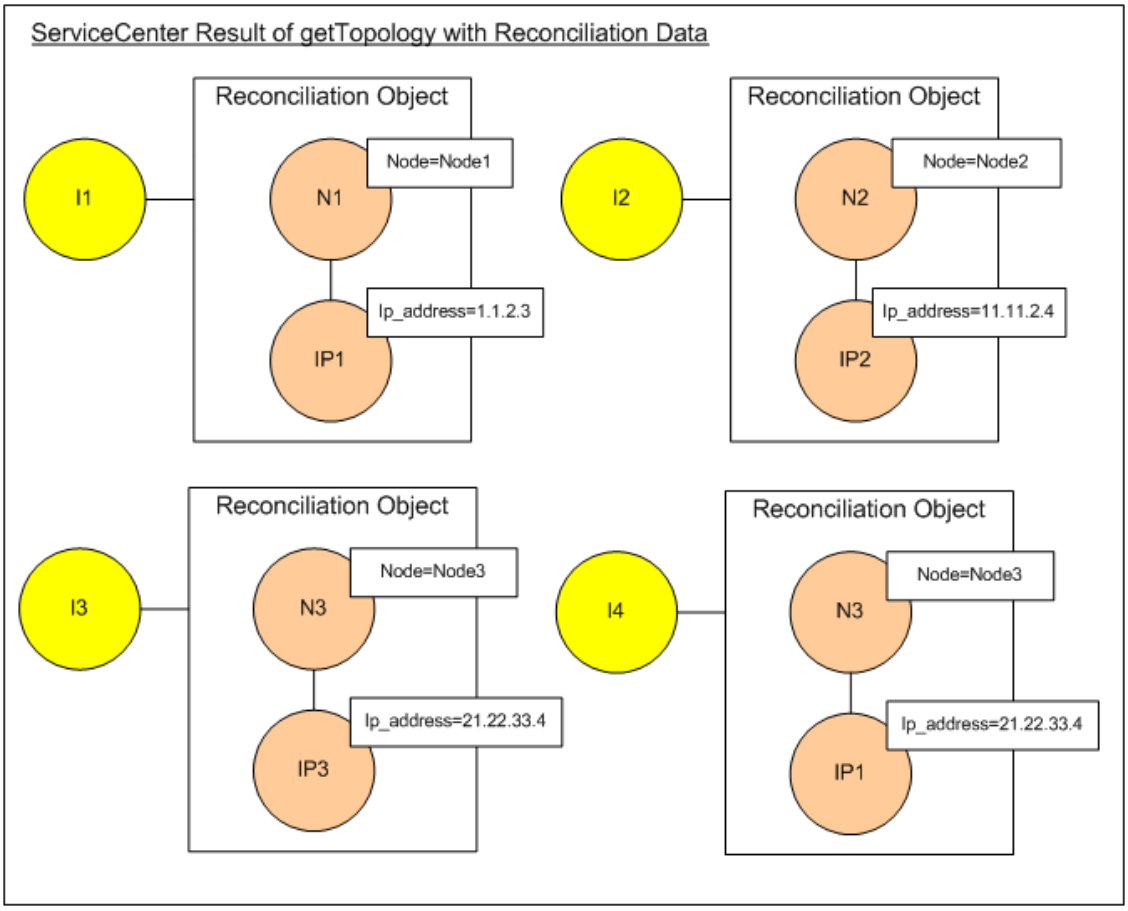
4. The Federation Framework creates a one-node topology query with the Node and ID conditions on it from the previous result (node in H1, H2, H3), and runs this query with the required reconciliation data on UCMDB. The result includes Node CIs that are relevant to the ID condition and the appropriate reconciliation object for each CI:



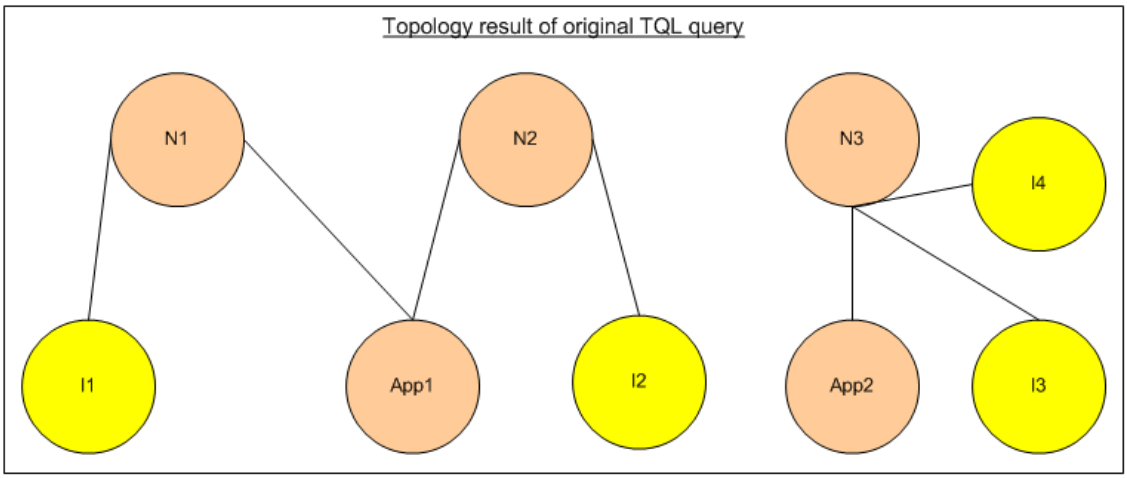
5. The reconciliation data for ServiceCenter should contain a condition for `node` and `ip` that is derived from the reconciliation objects received from UCMDB:



6. The Federation Framework runs the ServiceCenter subgraph with the reconciliation data to request the topology and appropriate reconciliation objects, and receives the following results:



7. The result after connection in Mapping Engine and creating virtual relationships is:



8. The Federation Framework requests the original TQL layout for received instances from UCMDB and ServiceCenter.

Federation Framework Flow for Population

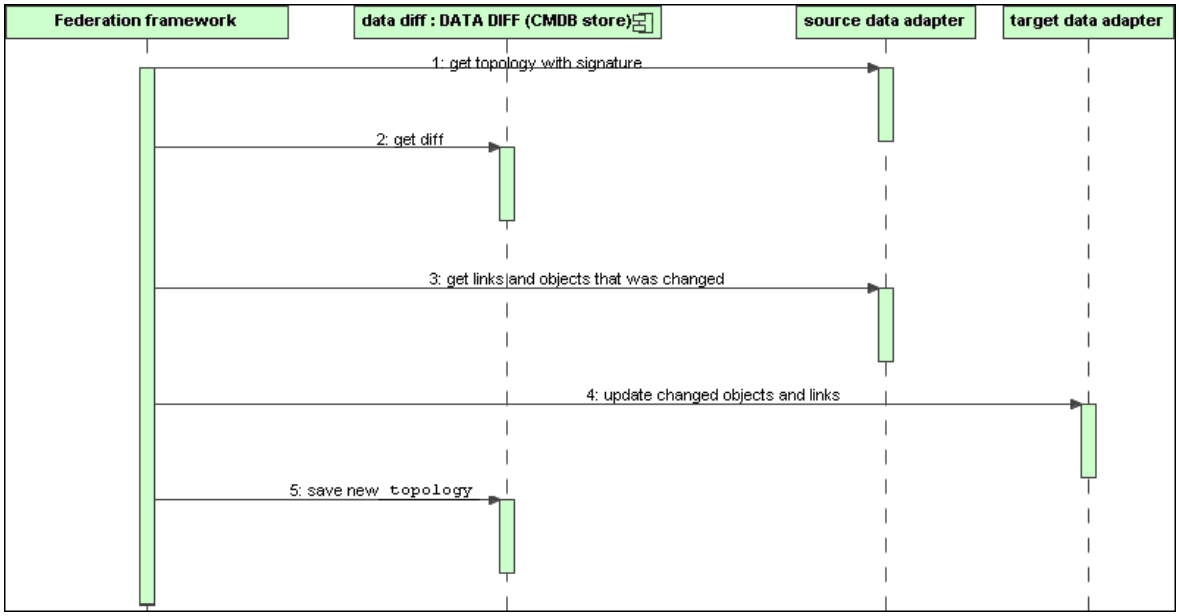
This section includes the following topics:

Definitions and Terms

Signature. Denotes the state of properties in the CI. If changes are made to property values in a CI, the CI signature must also be changed. The CI signature helps to detect whether a CI has changed without retrieving and comparing all CI properties. Both the CI and the CI signature are provided by the appropriate adapter. The adapter is responsible for changing the CI signature when the CI properties are altered.

Flow Diagram

The following sequence diagram illustrates the interaction between the Federation Framework and the source and target adapters in a population flow:



1. The Federation Framework receives the topology for the query result from the source adapter. The adapter recognizes the query by its name and runs it on the external data repository. The topology result contains the ID and signature for each CI and relationship in the result. The ID is the logical ID that defines the CI as unique in the external data repository. The signature should be modified if the CI or relationship is modified.

2. The Federation Framework uses signatures to compare the newly received topology query results with the saved ones, and to determine which CIs have changed.

3. After the Federation Framework finds the CIs and relationships that have changed, it calls the source adapter with the IDs of the changed CIs and relationships as a parameter to retrieve their full layout.

4. The Federation Framework sends the update to the target adapter. The target adapter updates the external data source with the received data.

5. After the update, the Federation Framework saves the last query result.

Adapter Interfaces

This section includes the following topics:

Definitions and Terms

External Relation. The relation between two external CI types that are supported by the same adapter.

Adapter Interfaces for Federated TQL Queries

Use the appropriate adapter interface for each adapter, as follows.

- A **Single Node topology interface** is used when the adapter does not support any external relations; that is, the adapter is never meant to receive a request with more than one external CI. The reconciliation data needed to complete the operation can be described as complex query (see SingleNodeFederationTopologyReconciliationAdapter below).

All SingleNode interfaces are created to simplify the workflow; for those cases where you need to use a more extensive query, use the **FederationTopologyAdapter** interface.

- A **FederationTopologyAdapter interface** is used to define adapters that support complex federated queries. The reconciliation request in these adapters is part of the **QueryDefinition** parameter.

  The Federation engine uses reconciliation data in order to connect the federated data to the proper local CIs. Reconciliation data may be fetched in more than one request (calculated recursively according to results). In this case, the adapter receives a request with only reconciliation data.

SingleNode Interfaces

The following interfaces have different types of reconciliation data:

- **SingleNodeFederationIdReconciliationAdapter.** Use if the adapter supports a **single-node TQL** and the reconciliation between data repositories is calculated by the ID.
- **SingleNodeFederationPropertyReconciliationAdapter.** Use if the adapter supports a **single-node TQL** and the reconciliation between data repositories is done by the properties of one CI.
- **SingleNodeFederationTopologyReconciliationAdapter.** Use if the adapter supports a **single-node TQL** and the reconciliation between data repositories is done by topology. The adapter should support the case where the query element is empty and only reconciliation topology is requested.

Data Adapter Interfaces

- **FederationTopologyAdapter.** Use this adapter to support complex federated TQL queries. Allows the most diversity. The adapter should support the case where the query definition is describing only reconciliation data.
- **PopulateDataAdapter.** Use this adapter to support complex federated TQL queries and population flows. In a population flow, this adapter retrieves the entire data set, and lets the probe filter the difference since the last execution of the job.
- **PopulateChangesDataAdapter.** Use this adapter to support complex federated TQL queries and population flows. In a population flow, this adapter supports the retrieval of only the changes that occurred since the last execution of the job.

Note: When developing an adapter that may return large data sets of data, its important to allow chunking by implementing the ChunkGetter Interface. See the Java document of the specific adapter for more information.

Resource Reporting Interfaces

The following interfaces enable the adapter to report the resources that can be configured to customize the adapter's behavior. This enables you to edit these resources directly from the Integration Studio. These interfaces should be used in addition to the regular adapter interfaces above.

- **PopulationQueriesResourcesLocator.** Defines which resources may be edited for each specific Population query.
- **PushQueriesResourceLocator.** Defines which resources may be edited for each Data Push query.
- **GeneralResourcesLocator.** Defines which general resources may be edited in this adapter.

Additional Interfaces

- **SortResultDataAdapter.** Use if you can sort the resulting CIs in the external data repository.
- **FunctionalLayoutDataAdapter.** Use if you can calculate the functional layout in the external data repository.

Adapter Interfaces for Synchronization

- **SourceDataAdapter.** Use for source adapters in population flows.
- **TargetDataAdapter.** Use for target adapters in data push flows.

Debug Adapter Resources

This task describes how to use the JMX console to create, view, and delete adapter state resources (any resources created using the resource manipulation methods in the DataAdapterEnvironment interface, which are saved in the UCMDB database or the Probe database) for debugging and development purposes.

1. Launch the Web browser and enter the server address, as follows:
   - For the UCMDB server: `http://localhost:8080/jmx-console`
   - For the Probe: `http://localhost:1977`

   You may have to log in with a user name and password.
2. To open the JMX MBEAN View page, do one of the following:
   - On the UCMDB server: click **UCMDB:service=FCMDB Adapter State Resource Services**

- On the Probe: click **type=AdapterStateResources**

3. Enter values in the operations that you want to use, and click **Invoke**.

Add an Adapter for a New External Data Source

This task explains how to define an adapter to support a new external data source.

This task includes the following steps:

1. **Prerequisites**

   Model-supported adapter classes for CIs and relationships in the UCMDB Data Model. As an adapter developer, you should:

   - have knowledge of the hierarchy of the UCMDB CI types to understand how external CITs are related to the UCMDB CITs
   - model the external CITs in the UCMDB class model
   - add the definitions for new CI types and their relationships
   - define valid relationships in the UCMDB class model for the valid relationships between adapter inner classes. (The CITs can be placed at any level of the UCMDB class model tree.)

Modeling should be the same regardless of federation type (on the fly or replication). For details on adding new CIT definitions to the UCMDB class model, see Working with the CI Selector in the *HP Universal CMDB Modeling Guide*.

For the adapter to support federated attributes on CITs, add this CIT to the supported classes with supported attributes and the reconciliation rule for this CIT.

2. **Define Valid Relationships for Virtual Relationships**

> **Note:** This section is only relevant for federation.

To retrieve federated CITs that are connected to local CMDB CITs, a valid link definition must exist between the two CITs in the CMDB.

a. Create a valid links XML file that contains these links (if they do not already exist).

b. Add the links XML file to the adapter package in the **\validlinks** folder. For details, see in the *HP Universal CMDB Administration Guide*.

> **Example of Valid Relationship Definition:**
>
> In the following example, the relation of type `containment` between instances of type `node` to instances of type `myclass1` is a valid relationship definition.

```
<Valid-Links>
    <Valid-Link>
        <Class-Ref class-name="containment">
        <End1 class-name="node">
        <End2 class-name="myclass1">
        <Valid-Link-Qualifiers>
    </Valid-Link>
</Valid-Links>
```

3. **Define an Adapter Configuration**

a. Navigate to **Adapter Management**.

b. Click the **Create new resource**  button and select **New Adapter**.

c. In the New adapter dialog box, select **Integration** and **Java Adapter**.

d. Right-click the adapter that you created and select **Edit Adapter Source** from the shortcut

menu.

e. Edit the following XML tags:

```
<pattern xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
id="newAdapterIdName"
xsi:noNamespaceSchemaLocation="../../Patterns.xsd"
description="Adapter Description" schemaVersion="9.0"
displayName="New Adapter Display Name">

<deletable>true</deletable>

<discoveredClasses>

<discoveredClass>link</discoveredClass>

<discoveredClass>object</discoveredClass>

</discoveredClasses>

<taskInfo
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.
 AdapterService">

<params
className="com.hp.ucmdb.discovery.probe.services.dynamic.core.
 AdapterServiceParams" enableAging="true"
enableDebugging="false" enableRecording=
"false" autoDeleteOnErrors="success" recordResult="false"
maxThreads="1" patternType="java_adapter"
maxThreadRuntime="25200000">

<className>com.yourCompany.adapter.MyAdapter.MyAdapterClass
</className>

</params>

<destinationInfo
className="com.hp.ucmdb.discovery.probe.tasks.BaseDestinationDa
ta">
```

```
<!-- check -->

<destinationData name="adapterId"
description="">${ADAPTER.adapter_id}</destinationData>

<destinationData name="attributeValues"
description="">${SOURCE.attribute_values}</destinationData>

<destinationData name="credentialsId"
description="">${SOURCE.credentials_id}</destinationData>

<destinationData name="destinationId"
description="">${SOURCE.destination_id}</destinationData>

</destinationInfo>

<resultMechanism isEnabled="true">

<autoDeleteCITs isEnabled="true">

<CIT>link</CIT>

<CIT>object</CIT>

</autoDeleteCITs>

</resultMechanism>

</taskInfo>

<adapterInfo>

<adapter-capabilities>

<support-federated-query>

<!--<supported-classes/> <!—see the section about supported
classes-->

<topology>

<pattern-topology /> <!—or <one-node-topology> -->
```

```
</topology>

</support-federated-query>

<!--<support-replicatioin-data>

<source>

<changes-source/>

</source>

<target/>

</adapter-capabilities>

<default-mapping-engine />

<queries />

<removedAttributes />

<full-population-days-interval>-1</full-population-days-
interval>

</adapterInfo>

<inputClass>destination_config</inputClass>

<protocols />

<parameters>

<!--The description attribute may be written in simple text or
HTML.-->

<!--The host attribute is treated as a special case by UCMDB-->

<!--and will automatically select the probe name (if possible)-
->

<!--according to this attribute's value.-->
```

```
<parameter name="credentialsId" description="Special type of
property, handled by UCMDB for credentials menu" type="integer"
display-name="Credentials ID" mandatory="true" order-index="12"
/>

<parameter name="host" description="The host name or IP address
of the remote machine" type="string" display-name="Hostname/IP"
mandatory="false" order-index="10" />

<parameter name="port" description="The remote machine's
connection port" type="integer" display-name="Port"
mandatory="false" order-index="11" />

</parameters>

<parameter name="myatt" description="is my att true?"
type="string" display-name="My Att" mandatory="false" order-
index="15" valid-values="True;False"/>True</parameters>

<collectDiscoveredByInfo>true</collectDiscoveredByInfo>

<integration isEnabled="true">

<category >My Category</category>

</integration>

<overrideDomain>${SOURCE.probe_name}</overrideDomain>

<inputTQL>

<resource:XmlResourceWrapper
xmlns:resource="http://www.hp.com/ucmdb/1-0-
0/ResourceDefinition" xmlns:ns4="http://www.hp.com/ucmdb/1-0-
0/ViewDefinition" xmlns:tql="http://www.hp.com/ucmdb/1-0-
0/TopologyQueryLanguage">

<resource xsi:type="tql:Query" group-id="2" priority="low" is-
live="true" owner="Input TQL" name="Input TQL">

<tql:node class="adapter_config" id="-11" name="ADAPTER" />
```

```
<tql:node class="destination_config" id="-10" name="SOURCE" />

<tql:link to="ADAPTER" from="SOURCE" class="fcmdb_conf_
aggregation" id="-12" name="fcmdb_conf_aggregation" />

</resource>

</resource:XmlResourceWrapper>

</inputTQL>

<permissions />

</pattern>
```

For details about the XML tags, see "XML Configuration Tags and Properties" on page 249.

4. **Define Supported Classes**

Define supported classes either in the adapter code by implementing the *getSupportedClasses()* method, or by using the pattern XML file.

```
<supported-classes>
    <supported-class name="HistoryChange" is-derived="false" is-
reconciliation-supported="false" federation-not-supported="false" is-id-
reconciliation-supported="false">
    <supported-conditions>
        <attribute-operators attribute-name="change_create_time">
            <operator>GREATER</operator>
            <operator>LESS</operator>
            <operator>GREATER_OR_EQUAL</operator>
            <operator>LESS_OR_EQUAL</operator>
            <operator>CHANGED_DURING</operator>
        </attribute-operators>
    </supported-conditions>
 </supported-class>
```

name	The name of the CI type
is-derived	Specifies whether this definition includes all inheriting children

is-reconciliation-supported	Specifies whether this class is used for reconciliation
is-id-reconciliation-supported	Specifies whether this class is used for id-reconciliation
federation-not-supported	Specifies whether this CIT should not be allowed for federation (blocking certain CITs, for example, a CIT defined solely for federation)
<supported-conditions>	Specifies the supported conditions on each attribute

5. **Implement the Adapter**

Select the correct adapter implementation class according to its defined capabilities. The adapter implementation class implements the appropriate interfaces according to defined capabilities.

If the adapter implements **getTopologyWithReconciliationData** and adapter capabilities include the ability to be used as a starting point, the adapter should also support requesting topology with reconciliation data without any conditions (only type). In this case the adapter should return full reconciliation data of the found results.

Adapter reconciliation support can be defined according to **global_id**, in which case **global_id** must be defined as part of the reconciliation attributes in the adapter supported classes. If adapter reconciliation support is defined according to **global_id**, then **getTopologyWithReconciliationData ()** should return the **global_id** as part of the reconciliation object properties. The UCMDB uses **global_id** for reconciliation of federation results for a CIT instead of the identification rule.

Part of the federation API is the DataAdapterEnvironment interface. This interface represents the environment of the data adapter. It contains the environment API needed for the adapter to work. For more information on the DataAdapterEnvironment interface, see "The DataAdapterEnvironment Interface" on page 251.

6. **Define Reconciliation Rules or Implement the Mapping Engine**

If your adapter supports federated TQL queries, you have two options for defining your Mapping Engine:

- Use the default mapping engine, which uses the CMDB's internal reconciliation rules for mapping. To use it, leave the **<default-mapping-engine>** XML tag empty.

- Write your own mapping engine by implementing the mapping engine interface and placing the JAR with the rest of the adapter code. To do this, use the following XML tag: **<default-mapping-engine>com.yourcompany.map.MyMappingEngine</default-mapping-engine>**

7. **Add Jars Required for Implementation to the Class Path**

   To implement your classes, add the **federation_api.jar** file to your code editor class path.

8. **Deploy the Adapter**

   Deploy the adapter package. For general details on deploying a package, see Package Manager in the *HP Universal CMDB Administration Guide*.

   The package should contain the following entities:

   - New CIT definition (optional):
   - Used only if the adapter supports new CI types that do not yet exist in UCMDB.
   - The new CIT definitions are located in the `class` folder in the package.
   - New data type definition (optional):
   - Used only if the new CITs require new data types.
   - The new data type definitions are located in the `typedef` folder in the package.
   - New valid relationships definition (optional):
   - Used only if the adapter supports the federated TQL.
   - The new valid relationships definitions are located in the `validlinks` folder in the package.
   - The pattern configuration XML file should be located in the `discoveryPatterns` folder in the package.
   - **Descriptor.** Defines the package definitions.
   - Place your compiled classes (normally a jar file) in the package under the **adapterCode\<adapter id>** folder.

     Note: The `adapter id` folder name has the same value as in the adapter configuration.

- If you create your own configuration file, you should place the file in the package under the **adapterCode\<adapter id>** folder.

9. **Update the Adapter**

   Changes to any of the adapter's non-binary files may be made in the Adapter Management module. Making changes to configuration files in the Adapter management module causes the adapter to reload with the new configurations.

   Updates may also be made by editing the files in the package (both binary and non-binary files), and then redeploying the package by using the Package Manager. For details, see "How to Deploy a Package" in the *HP Universal CMDB Administration Guide*.

Create a Sample Adapter

This example illustrates how to create a sample adapter. This task includes the following steps:

- "Select Adapter Logic" below
- "Load the Project" below

1. **Select Adapter Logic**

   When you implement an adapter, you must choose how to handle the condition logic in the implementation (property conditions, ID conditions, reconciliation conditions, and link conditions).

   a. Retrieve the entire data into the adapter memory and let it select or filter the needed CI Instances.

   b. Convert all the conditions into the data source language and let it filter and select the data. For example:

      - Convert the condition into a SQL query.
      - Convert the condition into a Java API filter object.

   c. Filter some of the data on the remote service, and have the adapter select and filter the remainder.

   In the MyAdapter example, the logic in option *a* is used.

2. **Load the Project**

Copy the files from the **C:\hp\UCMDB\UCMDBServer\tools\adapter-dev-kit\SampleAdapters** folder and follow the instructions in the readme files.

Note: If you use an adapter with large data sets, you may need to use caching and indexing to improve performance for Federation.

Online javadocs documentation is available at:

C:\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DBAdapterFramework_JavaAPI\index.html

XML Configuration Tags and Properties

id="newAdapterIdName"		Defines the adapter's real name. Used for logs and folder lookups.
displayName="New Adapter Display Name"		Defines the adapter's display name, as it appears in the UI.
<className>...</className>		Defines the adapter's interface implementing the Java class.
<category >My Category</category>		Defines the adapter's category.
<parameters>		Defines the properties for the configuration that are available in the UI when setting up a new integration point.
	name	The name of the property (used mostly by code).
	description	The display hint of the property.
	type	String or integer (use valid values with string for Boolean).
	display-name	The name of the property in the UI.
	mandatory	Specifies whether this configuration property is mandatory for the user.
	order-index	The placing order of the property (small = up).

	valid-values	A list of possible valid values separated by `;' characters (for example, valid-values="Oracle;SQLServer;MySQL" or valid-values="True;False").
<adapterInfo>		Contains the definition of the adapter's static settings and capabilities.
	<support-federated-query>	Defines this adapter as capable of federation.
	<start-point-adapter>	Specifies that this adapter is the start point for TQL query calculation.
	<one-node-topology>	The ability to federated queries with one federated query node.
	<pattern-topology>	The ability to federate complex queries.
	<support-replicatioin-data>	Defines the capability to run data push and population flows.
	<source>	This adapter may be used for population flows.
	<push-back-ids>	Push back the global ID of the CI to the global_id column of the table (must be defined in the orm.xml). The behavior can be overridden by implementing the **FcmdbPluginPushBackIds** plug-in.
	<changes-source>	This adapter may be used for population changes flows.
	<instance-based-data>	This tag defines that the adapter supports an instance based population flow.
	<target>	This adapter may be used for data push flows.
	<default-mapping-engine>	Allows defining a mapping engine for the adapter (by default, the adapter uses the default mapping engine). For any other mapping engine, enter the implementing class name of the mapping engine
	<removedAttributes>	Forces the removal of specific attributes from the result.

	<full-population-days-interval>	Specifies when to execute a full population job instead of a differential job (every `x' days). Uses the aging mechanism together with the changes flow.
	<adapter-settings>	The list of settings of the adapter.
	<list.attributes.for.set>	Determines which attributes override the previous value (if one exists).

The DataAdapterEnvironment Interface

OutputStream openResourceForWriting(String resourceName) throws FileNotFoundException;

This method opens a resource with a given name for writing. It is used for saving persistent data for the integration. This method should be used instead of trying to load files using java methods. The user should ensure that the stream is closed when finished writing to the stream. close()/flush() will save the resource. This method creates a runtime resource (it may not overwrite files that came in the adapter package).

Parameter

- **resourceName:** The name of the resource to retrieve. This name should be unique across all integrations of the same adapter.

Return Value

Returns a stream to which to write.

Exceptions

- This method throws the *FileNotFoundException* if the resource type is file and the file does not exist, if the resource is a directory rather than a regular file, or for some other reason the resource cannot be opened for reading.
- This method throws the *SecurityException* if a security manager exists and its *checkRead* method denies access to the file.

InputStream openResourceForReading(String resourceName) throws FileNotFoundException;

This method opens a resource with a given name for reading. It is used for reading persistent data for the integration. This method should be used instead of trying to load a file using java methods.The user should ensure that the stream is closed when finished reading it. It first attempts to load files that came in the adapter package. If not found, it attempts to load a runtime created resource from *DataAdapterEnvironment.openResourceForWriting(String)*. The runtime resources can be viewed using JMX (of the probe and server accordingly).

Parameter

- **resourceName:** The name of the resource to retrieve. This name should be unique across all integrations of the same adapter.

Return Value

Returns a stream to read.

Exceptions

- This method throws the *FileNotFoundException* if the resource type is **file** and the file does not exist, if the resource is a directory rather than a regular file, or for some other reason the resource cannot be opened for reading.
- This method throws the *SecurityException* if a security manager exists and its *checkRead* method denies read access to the file.

Properties openResourceAsProperties(String propertiesFile) throws IOException;

This method opens a resource with a given name and loads it as a *Properties* structure. It is used for reading persistent data for the integration. This method should be used instead of attempting to load the **.properties** files using java methods. It first attempts to load files that came in the adapter package. If not found, it attempts to load a runtime created resource from *DataAdapterEnvironment.openResourceForWriting(String)*. The runtime resources can be viewed using JMX (of the probe and server accordingly).

Parameter

- **propertiesFile:** The name of the resource to retrieve. This name should be unique across all integrations of the same adapter.

Return Value

Returns the file content represented in Properties.

Exceptions

- This method throws the *FileNotFoundException* if the resource type is **file** and the file does not exist, if the resource is a directory rather than a regular file, or for some other reason the resource cannot be opened for reading.
- This method throws the *SecurityException* if a security manager exists and its *checkRead* method denies read access to the file.
- This method throws the *IOException* if the properties file failed to convert to the *Properties* Object.

String openResourceAsString(String resourceName) throws IOException;

This method opens a resource with a given name and loads it as a string. It is used for reading persistent data for the integration. This method should be used instead of trying to load files using java methods.

It first attempts to load files that came in the adapter package. If not found, it attempts to load a runtime created resource from *DataAdapterEnvironment.openResourceForWriting(String)*. The runtime resources can be viewed using JMX (of the probe and server accordingly).

Parameter

- **resourceName:** The name of resource to retrieve. This name should be unique across all integrations of the same adapter.

Return Value

Returns the file content represented in String format.

Exceptions

- This method throws the *FileNotFoundException* if the resource type is **file** and the file does not exist, if the resource is a directory rather than a regular file, or for some other reason the resource cannot be opened for reading.

- This method throws the *SecurityException* if a security manager exists and its *checkRead* method denies read access to the file.
- This method throws the *IOException* if an I/O error occurs.

public void saveResourceFromString(String relativeFileName, String value) throws IOException;

This method receives a String and saves it as a resource. It is used for saving persistent data for the integration. This method should be used instead of trying to save files using java methods. This method converts the String into a stream and saves it to the resource. It creates a runtime resource, but cannot overwrite files that came in the adapter package). The runtime resources can be viewed using JMX (of the probe and server accordingly).

Parameter

- **relativeFileName:** The name of resource to retrieve. This name should be unique across all integrations of the same adapter.
- **value:** The String to save as a resource

Exceptions

This method throws the IOException if an I/O error occurs.

boolean resourceExists(String resourceName);

This method checks if the given resource name exists. It looks for files that came in the adapter package and for runtime created resources from *DataAdapterEnvironment.openResourceForWriting (String)*.

Parameter

- **resourceName:** The name of the resource to retrieve. This name should be unique across all integrations of the same adapter.

Return Value

Returns **True** if *resourceName* exists.

boolean deleteResource(String resourceName);

This method deletes the given resource from persistent data. It deletes a runtime resource, and may not delete files that came in the adapter package. The runtime resources can be viewed using JMX (for the probe and server accordingly).

Parameter

- **resourceName:** The name of the resource to delete. This name should be unique across all integrations of the same adapter.

Return Value

Returns **True** if the resource is successfully deleted.

Collection<String> listResourcesInPath(String path);

This method retrieves a list of resources in the given resource path. It looks for files that came in the adapter package and for a runtime created resources from *DataAdapterEnvironment.openResourceForWriting(String)*. The runtime resources can be viewed using JMX (for the probe and server accordingly).

Parameter

- **path:** The resource path. For example, "META-INF/myfiles/"

Return Value

Return a list of resources in the path.

DataAdapterLogger getLogger();

Retrieves the logger to be used by the adapter.This logger is used for logging events in your adapter.

Return Value

Returns the logger that is used by the DataAdapter.

DestinationConfig getDestinationConfig();

This method retrieves the destination configuration of the integration. This configuration holds all connection and running settings for the integration.

Return Value

Returns the DestinationConfig of the Adapter.

int getChunkSize();

This method retrieves the population chunk size requested for this integration.

Return Value

Returns the population chunk size.

int getPushChunkSize();

This method retrieves the push chunk size requested for this integration.

Return Value

Returns the push chunk size.

ClassModel getLocalClassModel();

This method retrieves a class model for querying information about the local UCMDB's class model. This method brings an updated ClassModel. Once the ClassModel object is returned, it is not updated for any class model changes. In order to retrieve an updated class model, use this method again to retrieve it.

Return Value

Returns the UCMDB's class model.

CustomerInformation getLocalCustomerInformation();

This method retrieves customer information for the customer that is executing the adapter.

Return Value

Returns customer information for the customer that is executing the adapter.

Object getSettingValue(String name);

This method retrieves a specific adapter setting.

Parameter

name: The name of setting.

Return Value

Returns the Object setting value.

Map<String, Object> getAllSettings();

This method retrieves all adapter settings.

Return Value

Returns the adapter settings.

boolean isMTEnabled();

This method checks if the server environment supports Multiple Tenancy (MT).

Return Value

Returns **true** if the server environment supports MT, otherwise returns **false**.

String getUcmdbServerHostName();

This method returns the local UCMDB server host name.

Return Value

Returns the local UCMDB server host name.

Chapter 7: Developing Push Adapters

This chapter includes:

Developing and Deploying Push Adapters

Generic Push Adapters provide a common platform that enables rapid development of integrations that push UCMDB data to external data repositories (databases and third-party applications). Generic Push Adapters are categorized according to the protocol used to push the data. For details on pushing via XML, using the Generic XML Push Adapter, see "Generic XML Push Adapter SQL Queries" on page 274. For details on pushing via Web Service, using the Generic Web Service Push Adapter, see "Generic Web Service Push Adapter" on page 274.

Developing a custom integration based on a Generic Push Adapter requires:

- Building a new adapter package from the appropriate Generic Push Adapter template files. For details, see "Build an Adapter Package" on the next page.
- Mappings between the UCMDB CI link types and the external data items. The mappings are stored as XML and are customized to each external data repository. For details, see "Create Mappings" on page 263.

- A Jython script to push the data items into the external data repository. For details, see "Write Jython Scripts" on page 267.
- Additional adapter-specific steps. For example, choosing the path of the file to be written for the XML push adapter, or creating a data receiver for the Web Service push adapter.

Build an Adapter Package

To create a new, MDR-specific push adapter, you should make a copy of the generic adapter and then edit it to customize it as an adapter for a specific push target.

Generic adapters packages can be found in one of the following two locations:

- Generic XML push adapter: **hp\UCMDB\UCMDBServer\content\adapters\push-adapter.zip**
- Generic web service adapter: **hp\UCMDB\UCMDBServer\content\adapters\web-service-push-adapter.zip**

To create a new push adapter from the generic push adapter:

1. Extract the content of the selected package zip file to a work folder.
2. Review the following directories in preparation for the rename and replace phase:
   - **adapterCode:** Contains the directory that is deployed to the **C:\hp\UCMDB\UCMDBServer\runtime\fcmdb\CodeBase directory**. Jars deployed here do not automatically restart the probe and do not appear automatically in the probe's CLASSPATH.
   - **discoveryConfigFiles:** Contains the adapter's mappings definitions and points to the correct Jython script (**push.properties**)
   - **discoveryPatterns:** Contains the adapter's XML definition that is deployed on the UCMDB server
   - **discoveryScripts:** Contains the adapter's Jython scripts via which the connection to the third party data store is made and data is pushed
   - **discoveryResources:** Contains the **UCMDBDataReceiver.jar** containing the Java integration classes for the web service.

   Note: When you deploy this package, the probe is restarted to include this **.jar** in the probe's

CLASSPATH. No action is required beyond deploying the package.

3. Make the following changes within the unzipped adapter directory structure:

   a. **discoveryConfigFiles\<Your_Push_Adapter_Name>:** rename the directory "PushAdapter" or "XMLtoWebService" to the name of the new push adapter (for example, “myPushAdapter”).

   b. **discoveryConfigFiles\<Your_Push_Adapter_Name>\push.properties:** In the **push.properties** file, do the following:

      - Update the name of **jythonScript.name** to the name of the Jython script to be used by the new push adapter (for example, **pushToMyService.py**).
      - Update the name of the mappings file to be used by the new push adapter (for example, myPushAdapter_mappings). Do no add the **.xml** extension, this is filled in automatically.

   c. **discoveryPatterns\<push adapter name>.xml:** Rename this file to the name of the new adapter's definition XML file (for example, **my_push_adapter.xml**).

   d. **discoveryPatterns\<your_push_adapter>.xml:** Update this file as follows:

      - For the XML element **<pattern>:** set the id and description attributes accordingly. For example:

        ```
        <pattern id="PushAdapter"
        xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery
        Pattern Description" schemaVersion="9.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        ```

        is changed to:

        ```
        <pattern id="MyPushAdapter" displayLabel="My Push Adapter"
        xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="Discovery
        Pattern Description" schemaVersion="9.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        ```

      - For the XML element **<parameters>:** update the children elements according to the needs of your adapter. By default, the following children elements are used to define a push adapter. These values are assigned when the integration point is defined in the Integration Studio after the adapter is configured. Update the parameter list so that the list of

parameters reflects the required connection attributes. Do not remove the **probeName** attribute.

- **host:** the server name hosting the web service
- **port:** port of the listening UCMDB Data Receiver service
- **Web Service Push Adapter: uri** - the remainder of the URL to form the service endpoint address of the data receiver.
- **probeName:** defines on which data flow probe the push job runs

◦ For the XML element **<integration>:** update the value of the child element **<category>** to something other than Generic. By default integration adapters belonging to the Generic category are not shown in the Integration Studio. If you are integrating with a third-party data store, set this value to "Third Party". If you are integrating with an HP BTO product, set this value to "HP BTO Products".

e. **adapterCode\PushAdapter:** rename this folder with the adapter ID used in the previous step (for example, **adapterCode\MyPushAdapter**).

f. **discoveryScripts\<your_Jython_push_script>.py:** create a file with the same name as the one defined in the **push.properties jythonScript.name** property. In the **discoveryScript** file, there is a script that inserts the CIs and links to an external Oracle database. Replace **discoveryScripts\pushScript.py** with the script you wrote (for details, see "Write Jython Scripts" on page 267). If you rename the script, the **jythonScript.name** property in **adapterCode\<adapter ID>\push.properties** should be updated accordingly.

◦ XML Push Adapter: **pushScript.py**

◦ Web Service Push Adapter: **XMLtoWebService.py**

g. **tql\<your_integration_TQLs>:** like a regular package, place the TQL XML definition of your integration TQLs in this directory. All TQLs in this folder are deployed when the adapter package is deployed.

h. **discoveryConfigFiles\<Your_Push_Adapter_Name>\mappings:** create an XML mapping file per TQL that you want to use in your integration. Note that the push adapter applies the transformations in the mapping file to the results of the integration TQLs and then sends that data in three parameters (**addResult**, **updateResult**, and **deleteResult**) of an ad hoc task to the data flow probe.

i. **adapterCode\<adapter ID>\mappings:** replace the **mappings.xml** file with the mapping files you prepared (for details, see "Create Mappings" on the next page).

XML Push Adapter: This mapping example corresponds to the example of the tables created in ORACLE in the sql_queries file.

To use a mapping file for each TQL method, assign the name of the corresponding TQL to each XML file, followed by **.xml**. In this case, the **mappings.xml** file is used as by default, if no specific mapping file is found for the current TQL name. The name of the default mapping file can be modified by changing the **mappingFile.default** property in **adapterCode\<adapter ID>\push.properties**.

4. After making all the above changes, create a **.zip** file by selecting the folders and files specified in step 3 above (for example, **my_Push_Adapter.zip**).

5. Deploy the newly created **.zip** file on the UCMDB server via the Package Manager (go to **Administration > Package Manager**).

6. Create an integration point in **Data Flow Management > Integration Studio** and define the integration TQLs that the integration point uses. Set a schedule for automatic data push.

Troubleshooting

The procedure for building a new push adapter requires complete and correct re-naming and replacing. Any error will likely affect the adapter. The package must be unzipped and re-zipped correctly to act as a UCMDB package. Refer to the out-of-the-box packages as examples. Common errors include:

- Including another directory on top of the package directories in the ZIP file.
  Solution: ZIP the package in the same directory as the package directories such as **discoveryResources**, **adapterCode**, etc. Do not include another directory level on top of this in the ZIP file.

- Omitting a critical re-name of a directory, a file, or a string in a file.
  Solution: Following the instructions in this section very carefully.

- Misspelling a critical re-name of a directory, a file, or string in a file.
  Solution: Do not change your naming convention in mid-stream once you begin the re-naming procedure. If you realize that you need to change the name, start over completely rather than trying to retroactively correcting the name, as there is a high risk of error. Also, use search and replace rather than manually replacing strings to reduce risk of errors.

- Deploying adapters with the same file names as other adapters, especially in the **discoveryResources** and **adapterCode** directories.
  Solution: You may be using a UCMDB version with a known issue that prevents mappings files from having the same name as any other adapter in the same UCMDB environment. If you attempt to deploy a package with duplicates names, the package deployment will fail. This problem may occur even if these files are in different directories. Further, this problem can occur regardless of whether the duplicates are within the package or with other previously deployed packages.

At this point you can create a new push adapter job in the Integration Studio using the new adapter you just deployed.

TQL Best Practices for Push Adapters

1. Create a folder structure in the TQL and View trees, and keep all new TQLs and views there. Use a naming convention.
2. Unless the TQL is small, first copy the most similar TQL.
3. Make one change at a time. Save, test, and preview after each change. Repeat until the results comply with your requirements.

Create Mappings

The raw TQL result data is in the form of the UCMDB class model schema. It is likely that the consumer uses a different data model. The push adapter provides a mapping mechanism to transform the data into a format more suitable for consumption. Mappings perform both direct and complex transformations, from direct, naming-type conversion, to parent/child aggregation and referencing functions.

The mapping specification can be found in the section "Mapping File Reference" on page 294. Use the reference to create a mapping file.

Note: The adapter properties file refers to the name of the mapping file. In adapter configuration files, the adapter implements a folder structure using the name of the adapter. Rename this folder when implementing an adapter to maintain uniqueness as required by the Package Manager.

Build a Mapping File

1. Start with a default mapping file.
2. Deploy the adapter and run it once.
3. Observe the results.
4. Identify and note what should be changed.
5. Make the changes identified in the previous step. The following list can help serve as a guide for the order of the changes.
    a. Start with the top, non-transformative section. Make sure the adapter runs after each change.
    b. Change the source CIs section to the UCMDB names in the TQL result.
    c. First map the keys.
    d. Then add all the direct mappings.
    e. Add the complex mappings.
    f. Add the link mappings.

Repeat steps 2-5 until the mapped data is suitable for consumption. Select the appropriate Generic adapter package from which to create the new push adapter.

The mapping files work the same way for all types of push adapters. The Generic XML push adapter writes the mapped results to a file. The Generic Web Service Push Adapter sends the XML results to a data receiver. For more details, see "Generic Web Service Push Adapter" on page 274.

Prepare the Mapping Files

Note: You can retrieve all of the CIs and relationships as they are in the CMDB without mapping, by not creating the **mappings.xml** file. This returns all of the CIs and relationships with all of their attributes.

There are two different ways to prepare mapping files:

- You can prepare a single, global mapping file.

  All mappings are placed in a single file named **mappings.xml**.

- You can prepare a separate file for each push query.

  Each mapping file is called **<query name>.xml**.

For details, see "Mapping File Schema" on page 297.

This task includes the following steps:

- "Create a mappings.xml File" below
- "Map CIs" below
- "Map Links" on the next page

1. **Create a mappings.xml File**

   The mapping file structure is created as follows (use an existing file as a template):

```
<?xml version="1.0" encoding="UTF-8"?>
<integration>
    <info>
        <source name="UCMDB"  versions="9.x" vendor="HP" >
        <!-- for example: -->
        <target name="Oracle" versions="11g" vendor="Oracle" >
    </info>
    <targetcis>
        <!--- CI Mappings --->
    </targetcis>
    <targetrelations>
        <!--- Link Mappings --->
    </ targetrelations>
</integration>
```

2. **Map CIs**

   There are two ways to map CMDB CI types:

   - Map a CI type so that CIs of that type and all inherited types are mapped in the same way:

```
<source_ci_type_tree name="node" mode="update_else_insert">
```

```
    <apioutputseq>1</apioutputseq>
    <target_ci_type  name="host">
        <targetprimarykey>
            <pkey>name</pkey>
        </targetprimarykey
        <target_attribute name=" name" datatype="STRING">
            <map type="direct" source_attribute="name" >
        </target_attribute>
        <!-- more target attributes --->
    </target_ci_type>
</source_ci_type_tree>
```

- Map a CI type so that only CIs of that type are processed. CIs of inherited types are not processed unless their type is also mapped (in one of the two ways):

```
<source_ci_type name="node" mode="update_else_insert">
    <apioutputseq>1</apioutputseq>
    <target_ci_type  name="host">
        <targetprimarykey>
            <pkey>name</pkey>
        </targetprimarykey
        <target_attribute name=" name" datatype="STRING">
            <map type="direct" source_attribute="name" >
        </target_attribute>
        <!-- more target attributes --->
    </target_ci_type>
</source_ci_type>
```

A CI type which is mapped indirectly (one of its ancestors is mapped using **source_ci_type_tree**), can also override its parent's map by having it appear in its own **source_ci_type_tree** or **source_ci_type**.

It is recommended to use **source_ci_type_tree** wherever possible. Otherwise, resulting CIs of a CI type that do not appear in the mapping files will not be transferred to the Jython script.

3. **Map Links**

There are two ways to map links:

- Map a link so that links of that type and all inherited links are mapped in the same way:

```
<source_link_type_tree name="dependency" target_link_type="dependency"
mode="update_else_insert" source_ci_type_end1="webservice" source_ci_
type_end2="sap_gateway">
```

```
    <target_ci_type_end1 name="webservice" >
    <target_ci_type_end2 name="sap_gateway" >
        <target_attribute name="name" datatype="STRING">
            <map type="direct" source_attribute="name" >
        </target_attribute>
</source_link_type_tree>
```

- Map a link so that only links of that type are processed. Links of inherited types are not processed unless their type is also mapped (in one of the two ways):

```
<link source_link_type="dependency" target_link_type="dependency"
mode="update_else_insert" source_ci_type_end1="webservice" source_ci_
type_end2="sap_gateway">
        <target_ci_type_end1 name="webservice" >
        <target_ci_type_end2 name="sap_gateway" >
        <target_attribute name="name" datatype="STRING">
            <map type="direct" source_attribute="name" >
    </target_attribute>
</link>
```

Write Jython Scripts

The mapping script is a regular Jython script, and should follow the rules for Jython scripts. For details, see "Developing Jython Adapters" on page 41.

The script should contain the **DiscoveryMain** function, which may return either an empty **OSHVResult** or a **DataPushResults** instance upon success.

To report any failure, the script should raise an exception, for example:

```
raise Exception('Failed to insert to remote UCMDB using TopologyUpdateService. See
log of the remote UCMDB')
```

In the DiscoveryMain function, the data items to be pushed to or deleted from the external application can be obtained as follows:

```
# get add/update/delete result objects (in XML format) from the Framework
addResult = Framework.getTriggerCIData('addResult')
updateResult = Framework.getTriggerCIData('updateResult')
deleteResult = Framework.getTriggerCIData('deleteResult')
```

The client object to the external application can be obtained as follows:

```
oracleClient = Framework.createClient()
```

This client object automatically uses the credentials ID, host name and port number passed by the adapter through the Framework.

If you need to use the connection parameters that you defined for the adapter (for details, see the step on editing the **discoveryPatterns\push_adapter.xml** file in "Build an Adapter Package" on page 259), use the following code:

```
propValue = str(Framework.getDestinationAttribute('<Connection Property Name'))
```

For example:

```
serverName = Framework.getDestinationAttribute('ip_address')
```

This section also includes:

- "Working with the Mapping's Results" below
- "Handling Test Connection in the Script" on page 271

Working with the Mapping's Results

Generic push adapters create XML strings that describe the data to be added, updated, or deleted from the target system. The Jython script needs to analyze this XML, and then performs the add, update, or delete operation on the target.

In the XML of the add operation that the Jython script receives, the `mamId` attribute for the objects and links is always the UCMDB identifier of the original object or link before its type, attribute, or other information was changed to the schema of the remote system.

In the XML of the update or remove operations, the `mamId` attribute of each object or link contains the string representation of the same `ExternalId` that was returned from the Jython script from the previous synchronization.

In the XML, the `id` attribute of a CI holds the `cmdbId` as an external id or the `ExternalId` of that CI if the CI got an `ExternalId` one when the CI was sent to the script. The `end1Id` and `end2Id` fields of the link hold for each of the link's ends the `cmdbId` as an external id or the `ExternalId` of that link's end if the CI at the link's end got an `ExternalId` when it was sent to the script.

When processing the CIs in the Jython script, the return value of the script is a mapping between the CI's CMDB id and the given id (the id given to each CI in the script). If a CI is pushed for the first time, the id that is in the XML of that CI is the CMDB id. If a CI is not pushed for the first time, the CI's id is the same id that was given to that CI in the script when it was first pushed.

The id is retrieved from the CI XML script as follows:

1. From the CI Element in the XML, retrieve the id from the id attribute. For example: `id = objectElement.getAttributeValue('id')`.

2. After retrieving the id from the XML, restore the id from the attribute (string). For example: `objectId = CmdbObjectID.Factory.restoreObjectID(id)`.

3. Check if the `objectId` received in the previous step is the CMDB id. You can do this by checking if the `objectId` has the new id that is given to it by the script. If it does, the returned id is not the CMDB id. For example:
   `newId = objectId.getPropertyValue(<the name of the id attribute which is given by the script>)`.

   If `newId` is null, then the id that was returned in the XML is a CMDB id.

4. If the id is a CMDB id (that is, `newId` is null), perform the following (if the id is not a CMDB id, go to step 5):

   a. Create a property for that CI that holds the new id. For example: `propArray = [TypesFactory.createProperty('<the name of the id attribute which is given by the script>', '<new id>')]`.

   b. Create an `externalId` to that CI. For example:
   ```
   cmdbId = extI.getPropertyValue('internal_id')
   className = extI.getType()
   externalId = ExternalIdFactory.createExternalCiId(className, propArray)
   ```

   c. Map the CMDB id to the newly created `externalId` (and in the next step return that mapping to the adapter). For example: `objectMappings.put(cmdbId, externalId)`

   d. When all of the CIs and links are mapped:
   ```
   updateResult = DataPushResultsFactory.createDataPushResults(objectMappings, linkMappings);
   return updateResult
   ```

5. If the id is the new id (that is, `newId` is not null), then the `externalId` is the `newId`.

It is also possible to report on the push status for each CI and link as follows:

1. `updateStatus = ReplicationActionDataFactory.createUpdateStatus();`

   where `updateStatus` is an instance of the `UpdateStatus` class that contains statuses of the CIs and links.

2. Add a status to `updateStatus` by calling the `reportCIStatus` or `reportRelationStatus` method.

   For example:

   ```
   status = ReplicationActionDataFactory.createStatus(Severity.FAILURE, 'Failed',
   ERROR_CODE_CI, errorParams,Action.ADD);

   updateStatus.reportCIStatus(externalId, status);
   ```

   Where `ERROR_CODE_CI` is the number of the error messages as they appear in adapter **properties.errors** file (for details on the **properties.errors** file, see "Error-Writing Conventions" on page 71), and `errorParams` contains the parameters to pass to the message. See **ReplicationActionDataFactory** javadoc for more details.

3. Create a push result with the statuses as follows:

   ```
   updateResult = DataPushResultsFactory.createDataPushResults(objectMappings,
   linkMappings, updateStatus);

   return updateResult
   ```

Example of the XML result

```
<root>

  <data>

    <objects>

      <Object mode="update_else_insert" name="UCMDB_UNIX" operation="add"
mamId="0c82f591bc3a584121b0b85efd90b174"
id="HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A">

        <field name="NAME" key="false" datatype="char" length="255">UNIX5</field>

        <field name="DATA_NOTE" key="false" datatype="char" length="255"></field>

      </Object>

    </objects>

    <links>

        <link targetRelationshipClass="TALK" targetParent="unix" targetChild="unix"
operation="add" mode="update_else_insert"
mamId="265e985c6ec51a8543f461b30fa58f81"
id="end1id%5BHiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D41372a1cbcaba27b214b84a2ec9eb535%0A%5D%0Aend2id%
5BHiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A%5D%0AHiddenRmi
```

```
DataSource%0Atalk%0A1%0Ainternal_
id%3DSTRING%3D265e985c6ec51a8543f461b30fa58f81%0A">

        <field name="DiscoveryID1">41372a1cbcaba27b214b84a2ec9eb535</field>

        <field name="DiscoveryID2">0c82f591bc3a584121b0b85efd90b174</field>

        <field name="end1Id">HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D41372a1cbcaba27b214b84a2ec9eb535%0A</field>

        <field name="end2Id">HiddenRmiDataSource%0Aunix%0A1%0Ainternal_
id%3DSTRING%3D0c82f591bc3a584121b0b85efd90b174%0A</field>

        <field name="NAME" key="false" datatype="char" length="255">TALK4</field>

        <field name="DATA_NOTE" key="false" datatype="char" length="255"></field>

        </link>

    </links>

  </data>

</root>
```

Note: If `datatype="BYTE"`, the returned result's value is a **String** that is generated as: `new String ([the byte array attribute])`. The `byte[] object` can be reconstructed by: `<the received String>.getBytes()`. If there are differences in the default locale between the server and the probe, the reconstruction is performed according to the server's default locale.

Handling Test Connection in the Script

A Jython script can be invoked to test the connection with an external application. In this case, the `testConnection` destination attribute will be `true`. This attribute can be obtained from the Framework as follows:

```
testConnection = Framework.getTriggerCIData('testConnection')
```

When run in test connection mode, a script should raise an exception if a connection to the external application cannot be established. Otherwise, if the connection is successful, the **DiscoveryMain** function should return an empty **OSHVResult**.

Support Differential Synchronization

For the Push adapter to support differential synchronization, the **DiscoveryMain** function must return an object implementing the **DataPushResults** interface, which contains the mappings between the IDs

that the Jython script receives from the XML and the IDs that the Jython script creates on the remote machine. The latter IDs are of the type **ExternalId**.

The **ExternalIdUtil.restoreExternal** command, which receives the ID of the CI in the CMDB as a parameter, restores the external ID from the ID of the CI in the CMDB. This command can be used, for example, while performing differential synchronization, and a link is received where one of its ends is not in the bulk (it was already synchronized).

If the **DiscoveryMain** method in the Jython script on which the Push adapter is based returns an empty **ObjectStateHolderVector** instance, the adapter will not support differential synchronization. This means that even when a differential synchronization job is run, in actuality, a full synchronization is being performed. Therefore, no data can be updated or removed on the remote system, since all data is added to the CMDB during each synchronization.

Important: If you are implementing differential synchronization on an existing adapter that was created in version 9.00 or 9.01, you must use the push-adapter.zip file from version 9.02 or later to recreate your adapter package. For details, see "Build an Adapter Package" on page 259.

This task enables the Push adapter to perform differential synchronization.

The Jython script returns the **DataPushResults** object which contains two Java maps - one for object ID mappings (keys and values are ExternalCiId type objects) and one for link IDs (keys and values are ExternalRelationId type objects).

- Add the following **from** statements to your Jython script:

```
from com.hp.ucmdb.federationspi.data.query.types import ExternalIdFactory

from com.hp.ucmdb.adapters.push import DataPushResults

from com.hp.ucmdb.adapters.push import DataPushResultsFactory

from com.mercury.topaz.cmdb.server.fcmdb.spi.data.query.types import
ExternalIdUtil
```

- Use the **DataPushResultsFactory** factory class to obtain the **DataPushResults** object from the **DiscoveryMain** function.

```
# Create the UpdateResult object

updateResult = DataPushResultsFactory.createDataPushResults(objectMappings,
linkMappings);
```

- Use the following commands to create Java maps for the **DataPushResults** object:

```
#Prepare the maps to store the mappings if IDs

objectMappings = HashMap()

linkMappings = HashMap()
```

- Use the **ExternalIdFactory** class to create the following ExternalId IDs:
  - ExternalId for objects or links originating in a CMDB (for example, all of the CIs in an add operation are from the CMDB):

    ```
    externaCIlId = ExternalIdFactory.createExternalCmdbCiId(ciType, ciIDAsString)

    externalRelationId = ExternalIdFactory.createExternalCmdbRelationId(linkType,
    end1ExternalCIId,
    end2ExternalCIId, linkIDAsString)
    ```

  - ExternalId for objects or links not originating in a CMDB (usually, every update and remove operation contains such objects):

    ```
    myIDField = TypesFactory.createProperty("systemID", "1")

    myExternalId = ExternalIdFactory.createExternalCiId(type, myIDField)
    ```

    Note: If the Jython script updated existing information and the ID of the object (or link) changes, you must return a mapping between the previous external ID and the new one.

- Use the **restoreCmdbCiIDString** or **restoreCmdbRelationIDString** methods from the **ExternalIdFactory** class to retrieve the UCMDB ID string from an External ID of an object or link that originated in UCMDB.
- Use the **restoreExternalCiId** and **restoreExternalRelationId** methods from the **ExternalIdUtil** class to restore the **ExternalId** object from the mamId attribute value of the XML of the update or remove operations.

  Note: ExternalId objects are actually an array of properties. This means that you can use an **ExternalId** object to store any information you may need that will identify the data on the remote system.

Generic XML Push Adapter SQL Queries

In the adapter package, the **sql_queries** file located in **adapterCode > PushAdapter > sqlTablesCreation**, contains the queries needed to create tables in a new schema in Oracle for testing the adapter. The tables correspond to the adapterCode\<adapter ID>\mappings\mappings.xml file.

Note: The **sql_queries** file is not needed for the adapter. It is only an example.

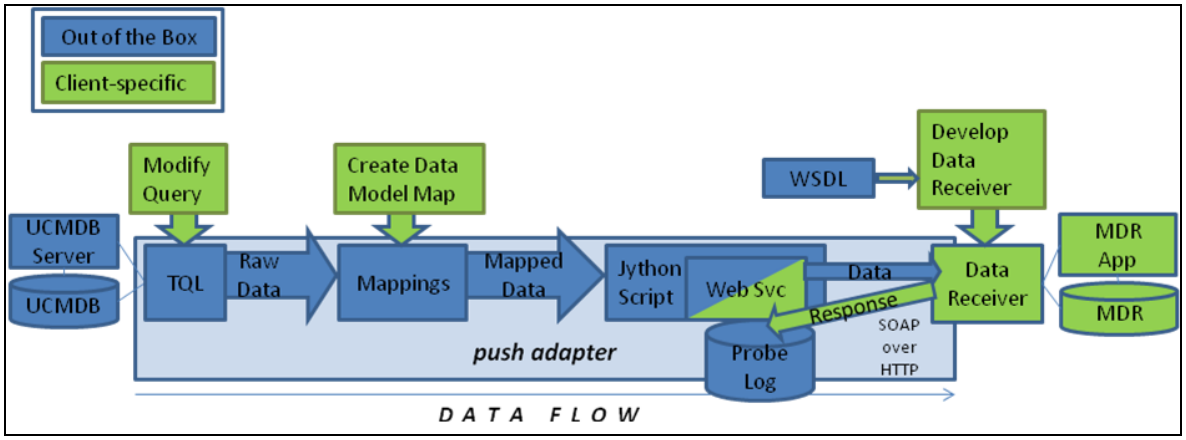
Generic Web Service Push Adapter

The generic web service push adapter provides a UCMDB-initiated push of SOAP messages containing query data to a web service data receiver. The mapped results are sent in standard SOAP messages via the HTTP POST protocol to the data receiver. The data receiver must understand the SOAP messages produced by the push adapter. To facilitate the development of the proper data receiver, a WSDL is provided with this push adapter.

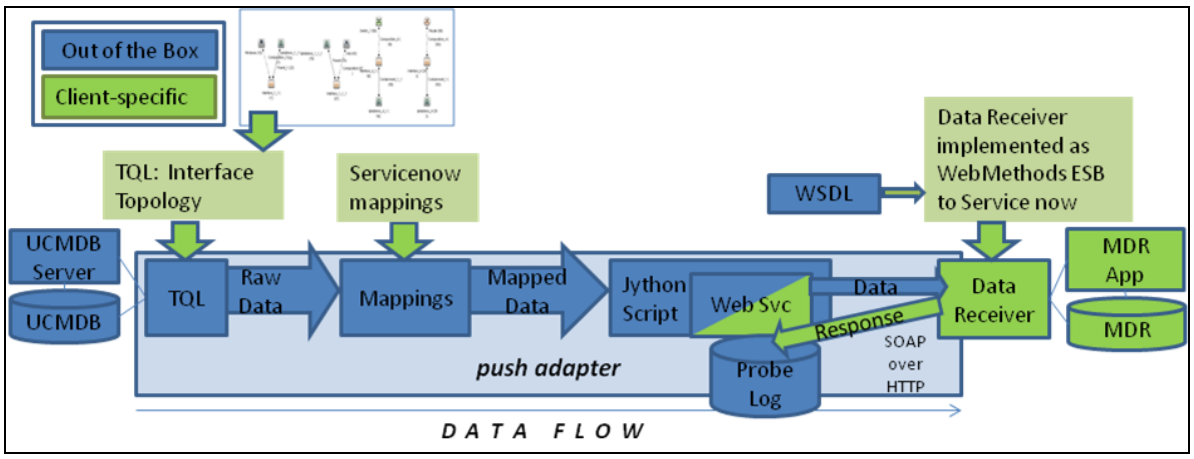
Custom processing of the SOAP message response XML is possible in the Jython script.

To understand the format of the incoming mapped data, the developer of the data receiver should communicate with the developer of the mappings file. An **.xsd** is currently not provided with this version of the web service push adapter, so data must be processed in a way reflective of the incoming data, which is a combination of the original TQL and the applied mappings.

The web service push adapter functions for pushing data to the client are shown below. The items in green are customized or supplied by the client to implement the adapter for a specific push target. The items in blue are out-of-the-box components.

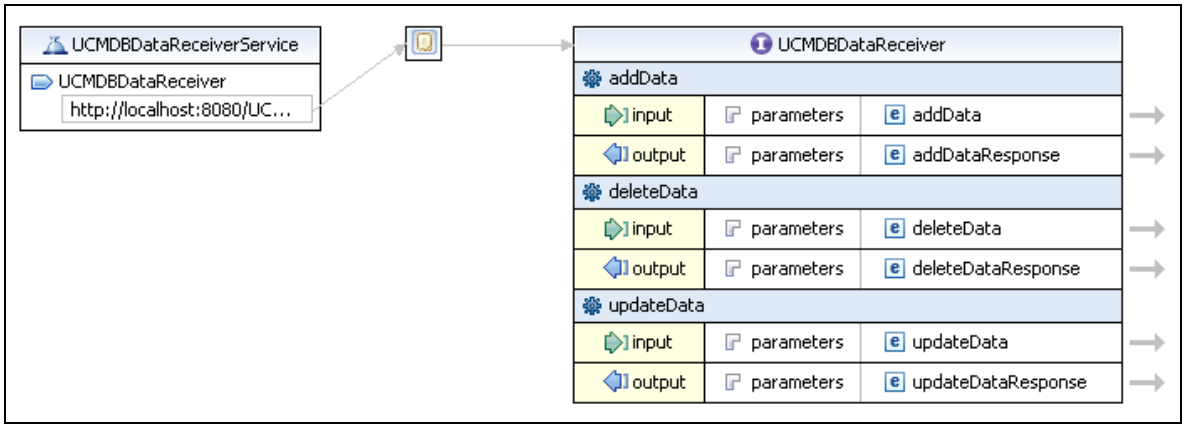


An example of an implementation of the generic web service push adapter to an MDR-specific push adapter using an Enterprise Service Bus (ESB) is shown here:



WSDL

A WSDL is supplied to the client developer to create a data receiver capable of communicating with the UCMDB push adapter via a web service. The **UCMDBDataReceiver.wsdl** describes the SOAP messages that are used to communicate data from UCMDB to the data receiver. The design diagram from the WSDL is shown here:



The data receiver (which is in practice a server or “service endpoint" in SOAP terminology) should implement three methods: **addData**, **deleteData**, and **updateData**, corresponding to the data sets that the UCMDB pushes. The HTTP headers contain the correct **SoapAction** keyword that indicates the type of data that is being sent. The data receiver is responsible for implementing the business logic and processing the data.

The default WSDL URL is:

- http://localhost:8080/UCMDBDataReceiver/services/UCMDBDataReceiver?wsdl

As implemented by the Data Receiver, the URL could look similar to the following:

- http://testWSPAserver:4444/MyCo.IT.SvcMgt.ws.us:provider/UCMDBDataReceiver?wsdl

The URL of the web service is the same as the WSDL URL without the "?wsdl" at the end.

The source for the WSDL is included below:

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://ucmdb.hp.com"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://ucmdb.hp.com" xmlns:intf="http://ucmdb.hp.com"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!--WSDL created by Apache Axis version: 1.4 Built on Apr 22, 2006 (06:55:48
PDT)-->

    <wsdl:types>

        <schema elementFormDefault="qualified"
        targetNamespace="http://ucmdb.hp.com"
        xmlns="http://www.w3.org/2001/XMLSchema">

            <element name="addData">

                <complexType>

                    <sequence>

                    <element name="xmlAdded" type="xsd:string"/>

                    </sequence>

                </complexType>

            </element>

            <element name="addDataResponse">

                <complexType/>

            </element>

            <element name="deleteData">

                <complexType>

                    <sequence>

                    <element name="xmlDeleted" type="xsd:string"/>

                    </sequence>
```

```
                </complexType>
            </element>
            <element name="deleteDataResponse">
                <complexType/>
            </element>
            <element name="updateData">
                <complexType>
                    <sequence>
                    <element name="xmlUpdate" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="updateDataResponse">
                <complexType/>
            </element>
        </schema>
    </wsdl:types>

    <wsdl:message name="addDataRequest">
        <wsdl:part element="impl:addData" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="deleteDataResponse">
        <wsdl:part element="impl:deleteDataResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="updateDataResponse">
        <wsdl:part element="impl:updateDataResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="deleteDataRequest">
```

```
        <wsdl:part element="impl:deleteData" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="addDataResponse">
        <wsdl:part element="impl:addDataResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="updateDataRequest">
        <wsdl:part element="impl:updateData" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="UCMDBDataReceiver">
        <wsdl:operation name="addData">
            <wsdlsoap:operation soapAction="addDataRequest"/>
            <wsdl:input message="impl:addDataRequest" name="addDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:addDataResponse" name="addDataResponse">
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="deleteData">
            <wsdlsoap:operation soapAction="deleteDataRequest"/>
            <wsdl:input message="impl:deleteDataRequest"
            name="deleteDataRequest">
            </wsdl:input>
            <wsdl:output message="impl:deleteDataResponse"
            name="deleteDataResponse">
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="updateData">
            <wsdlsoap:operation soapAction="updateDataRequest"/>
            <wsdl:input message="impl:updateDataRequest"
            name="updateDataRequest">
```

```
            </wsdl:input>

            <wsdl:output message="impl:updateDataResponse"
            name="updateDataResponse">

            </wsdl:output>

        </wsdl:operation>

    </wsdl:portType>

    <wsdl:binding name="UCMDBDataReceiverSoapBinding"
    type="impl:UCMDBDataReceiver">

        <wsdlsoap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />

        <wsdl:operation name="addData">

            <wsdl:input name="addDataRequest">

                <wsdlsoap:body use="literal" />

            </wsdl:input>

            <wsdl:output name="addDataResponse">

                <wsdlsoap:body use="literal" />

            </wsdl:output>

        </wsdl:operation>

        <wsdl:operation name="deleteData">

            <wsdl:input name="deleteDataRequest">

                <wsdlsoap:body use="literal" />

            </wsdl:input>

            <wsdl:output name="deleteDataResponse">

                <wsdlsoap:body use="literal" />

            </wsdl:output>

        </wsdl:operation>

        <wsdl:operation name="updateData">

            <wsdl:input name="updateDataRequest">

                <wsdlsoap:body use="literal" />

            </wsdl:input>

            <wsdl:output name="updateDataResponse">

                <wsdlsoap:body use="literal" />
```

```
            </wsdl:output>

        </wsdl:operation>

    </wsdl:binding>

    <wsdl:service name="UCMDBDataReceiverService">

        <wsdl:port binding="impl:UCMDBDataReceiverSoapBinding"
        name="UCMDBDataReceiver">

            <wsdlsoap:address
            location="http://localhost:8080/UCMDBDataReceiver/services/
            UCMDBDataReceiver"/>

        </wsdl:port>

    </wsdl:service>

</wsdl:definitions>
```

Response Handling

The data receiver should return a string in the **addDataResponse**, **deleteDataResponse**, or **updateDataResponse** structures. The adapter passes the response data unprocessed to the probe's **probeMgr-adaptersDebug.log**. The receiver can return any string data, and the responses are wrapped in SOAP-compliant XML. In the Jython script you can use the **SOAPMessage** and related Java classes to parse the response messages. The following is an example of a response message from the data receiver:

```
<2012-03-16 15:47:38,080> [INFO ] [Thread-110] - XMLtoWebService.py:addData
received response:

<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

<intf:addDataResponse xmlns:intf="http://ucmdb.hp.com">

    <xml>&lt;result&gt;&lt;status&gt;error&lt;/status&gt;
    &lt;message&gt;Error publishing config item changes&lt;/message&gt;
    &lt;/result&gt;</xml>

</intf:addDataResponse>

</soapenv:Body>
```

The message shown is an error message **<Error publishing config item changes>**, but the content can be anything that the data receiver is designed to respond with. The response is an error message simply because that is the intent, because the designer says it is an error message and the push adapter expects the response to be some indication of success or failure. The content can be reconciliation IDs of all the successfully added CIs, or error messages for specific CIs. Customization of the GWSPA could

include parsing the response message and taking actions such as resending certain CIs or performing other logging.

Testing the WSDL

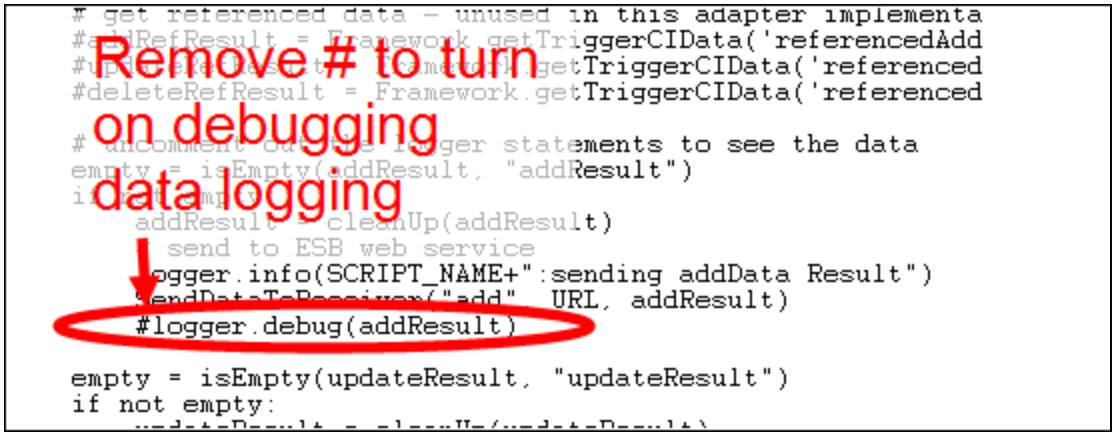
The SOAPUI Eclipse plug-in is used to test web service layers during development. You can use SOAPUI to assist with customization of a web service. SOAPUI offers an integrated development environment (IDE) to test building, sending, and receiving of SOAP messages. In the SOAPUI perspective, the WSDL on pages 276-280 generated the following sample message:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ucm="http://ucmdb.hp.com">
    <soapenv:Header/>
    <soapenv:Body>
        <ucm:addData>
            <ucm:xmlAdded>?</ucm:xmlAdded>
        </ucm:addData>
    </soapenv:Body>
</soapenv:Envelope>
```

The "?" in the **xmlAdded** element above is the location of the data, which is supplied by the web service push adapter integration.

Observing Results

When the push adapter is operating normally, in non-debug mode, the data is never written to a file until the final result is written (the intermediate TQL results and mapped data results are not normally visible in any log file). However, the results can be written to the probe's debug file by un-commenting the **logger.debug** statements (remove the "#" character) in the DiscoveryMain section as shown here:



Ensure the logger statement begins on the same column as the other preceding and following lines. Jython is indent-sensitive and the script will fail if the indention of all lines is not correct.

The debug log file **probeMgr-adaptersDebug.log** on the probe here shows the contents of the output:

```
<2011-12-07 14:02:23,019> [INFO ] [Thread-273] - XMLtoWebService.py started

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - ESB Push parameters:

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - Wshost=harpy.trtc.com

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] - WShostport=5555

<2011-12-07 14:02:23,019> [DEBUG] [Thread-273] -
WSuri=ws/DtITServiceManagement.esla.v1.ws.provider:UMDBDataReceiver

<2011-12-07 14:02:23,019> [INFO ] [Thread-273] - URL is
http://harpy.trtc.com:5555/ws/DtITServiceManagement.esla.v1.ws.
provider:UMDBDataReceiver

<2011-12-07 14:02:23,035> [DEBUG] [Thread-273] - Connected to
http://harpy.trtc.com:5555/ws/DtITServiceManagement.esla.v1.ws.
provider:UMDBDataReceiver

<2011-12-07 14:02:23,035> [ERROR] [Thread-273] - sending results

<2011-12-07 14:02:23,035> [DEBUG] [Thread-273] - <?xml version="1.0"
encoding="UTF-8"?>

<root>

    <data>

        <objects>

            <Object mode="" name="u_imp_ip_switch" operation="add"
            mamId="9e8c2f6bdfe4b7d0864c79e70833902c">
```

```
                <field name="Correlation ID" key="true" datatype="char"
                length="">9e8c2f6bdfe4b7d0864c79e70833902c</field>

                <field name="name" key="false" datatype="char" length="">nma_
                09sw</field>

                <field name="location" key="false" datatype="char" length="" />

                <field name="u_chassis_vendor_type" key="false" datatype="char"
                length="">ciscoCat2960-24TT</field>

                <field name="serial_number" key="false" datatype="char"
                length="" />

                <field name="ram" key="false" datatype="char" length="" />

                <field name="os_version" key="false" datatype="char" length=""
                />

            </Object>
```

Modifying the Jython Script

XMLtoWebService.py

The Jython script used by the Web Service push adapter is very similar to the XML push adapter. The script uses **UCMDBDataReceiver.jar**, included with the adapter. The script implements the **SendDataToReceiver()** method. **SendDataToReceiver()** uses three parameters:

1. Action (add, update, or delete)
2. The URL of the Data Receiver
3. The data

For example, the add block looks like: **SendDataToReceiver("add", URL, addResult)**

All web service and SOAP layers are wrapped. The URL is the service endpoint address of the UCMDB data receiver. This is the same URL used to obtain the wsdl via the "?wsdl" suffix.

The source of the Jython script is shown below. The web service integration wrapper lines are highlighted in green.

```
####################################

# script: XMLtoWebService.py

####################################
```

```
# This jython script accepts TQL data results (adds, updates, and deletes) from
the Integration adapter.

# and sends it to a web service. The web service is called UCMDBDataReceiver.

# A web service client of this name must be addressable at the URL provided by
the parameters.

# The SendDataToReceiver.jar exposes the SendDataToReceiver function, as well as
the service locator.

# examples of the service locator are in the testconnection section.

# regular expressions

import re

# logging

import logger

# web service interface

from com.hp.ucmdb import SendDataToReceiver

from com.hp.ucmdb.SendDataToReceiver import locateService

from com.hp.ucmdb.SendDataToReceiver import SendData

##############################################

########      VARIABLES             ##########

##############################################

SCRIPT_NAME = "XMLtoWebService.py"

logger.info(SCRIPT_NAME+" started")

def cleanUp(str):


    # replace mode=""

    str = re.sub("mode=\"\w+\"\s+", "", str)


    # replace mamId with id

    str = re.sub("\smamId=\"", " id=\"", str)


    # replace empty attributes

    str = re.sub("[\n|\s|\r]*<field name=\"\w+\" datatype=\"\w+\" />", "", str)
```

```
    # replace targetRelationshipClass with name
    str = re.sub("\stargetRelationshipClass=\"", " name=\"", str)

    # replace Object with object with name
    str = re.sub("<Object mode=\"", "<object mode=\"", str)
    str = re.sub("<Object operation=\"", "<object operation=\"", str)
    str = re.sub("<Object name=\"", "<object name=\"", str)
    str = re.sub("</Object>", "</object>", str)

    # replace field to attribute
    str = re.sub("<field name=\"", "<attribute name=\"", str)
    str = re.sub("</field>", "</attribute>", str)

    #logger.debug("String = %s" % str)
    #logger.debug("cleaned up")

    return str
def isEmpty(xml, type = ""):
    objectsEmpty = 0
    linksEmpty = 0

    m = re.findall("<objects />", xml)
    if m:
        #logger.warn("\t[%s] No objects found" % type)
        objectsEmpty = 1

    m = re.findall("<links />", xml)
    if m:
        #logger.warn("\t[%s] No links found" % type)
        linksEmpty = 1
```

```
    if objectsEmpty and linksEmpty:
        return 1
    return 0
##############################################
########      MAIN                  ##########
##############################################
def DiscoveryMain(Framework):
    #fix this for web service export
    errMsg = "UCMDBDataReceiver Service not found."
    testConnection = Framework.getTriggerCIData("testConnection")
    # Get Web Service Push variables
    WShostName = Framework.getTriggerCIData("Host Name")
    WShostport = Framework.getTriggerCIData("Protocol Port")
    WSuri = Framework.getTriggerCIData("URI")

    logger.info(SCRIPT_NAME+":ESB Push parameters:")
    logger.info("Host Name="+WShostName)
    logger.info("Protocol Port="+WShostport)
    logger.info("URI="+WSuri)
    URL = "http://"+WShostName+":"+WShostport+"/"+WSuri
    logger.info("URL="+URL)
    if testConnection == 'true':
        # locate the service
        test_receiver = SendDataToReceiver()
        locator = test_receiver.locateService(URL)
        #locator = locateService(URL)
        if(locator):
            logger.info(SCRIPT_NAME+":Test connection was successful")
            return
        else:
            raise Exception, errMsg
```

```
        return
# do same thing here if not just a test connection -
receiver = SendDataToReceiver()
locator = receiver.locateService(URL)
if(locator):
    logger.info(SCRIPT_NAME+":Connected to "+URL)
else:
    logger.error(SCRIPT_NAME+":no locator")
    raise Exception, errMsg
    return

# get add/update/delete result objects from the Framework
addResult = Framework.getTriggerCIData('addResult')
updateResult = Framework.getTriggerCIData('updateResult')
deleteResult = Framework.getTriggerCIData('deleteResult')
logger.debug(deleteResult)

# get referenced data - unused in this adapter implementation
#addRefResult = Framework.getTriggerCIData('referencedAddResult')
#updateRefResult = Framework.getTriggerCIData('referencedUpdateResult')
#deleteRefResult = Framework.getTriggerCIData('referencedDeleteResult')
# uncomment out the logger statements to see the data
empty = isEmpty(addResult, "addResult")
if not empty:
    addResult = cleanUp(addResult)
    # send to ESB web service
    logger.info(SCRIPT_NAME+":sending addData Result")
    rcvr = SendDataToReceiver()
    resp = rcvr.SendData("add", URL, addResult)
    logger.info(SCRIPT_NAME+":addData received response:"+resp)
    #logger.debug(addResult)
```

```
    empty = isEmpty(updateResult, "updateResult")
    if not empty:
        updateResult = cleanUp(updateResult)
        # send to ESB web service
        #logger.debug(updateResult)
        logger.info(SCRIPT_NAME+":sending updateData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("update", URL, updateResult)
        logger.info(SCRIPT_NAME+":received response:"+resp)

    empty = isEmpty(deleteResult, "deleteResult")
    if not empty:
        deleteResult = cleanUp(deleteResult)
        # send to ESB web service
        #logger.debug(deleteResult)
        logger.info(SCRIPT_NAME+":sending deleteData Result")
        rcvr = SendDataToReceiver()
        resp = rcvr.SendData("delete", URL, deleteResult)
        logger.info(SCRIPT_NAME+":received response:"+resp)
    logger.info(SCRIPT_NAME+" ended")
```

Customizing Response Message Processing

The data receiver should return a string containing any response or status desired. The web service push adapter passes by default the response to the probe's info-level log. The response message is SOAP-formatted XML containing the returned response string(s) inside. Any data can be returned by the receiver such as grouped or individual error or success messages. If additional processing is desired, the response can be processed by the adapter's Jython script. No Java programming is required.

An example of a return response message, sent using the following:

```
// stub example for building your own UCMDBDataData Receiver
public class UCMDBDataReceiver {
```

```
public String addData (String xmlAdd){
System.out.println(xmlAdd); // do something with the data
// send back a response message based on what you did
String tr = new String("a test response from addData!");
return tr;
}
```

is shown here:

```
<soapenv:Body xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <addDataResponse xmlns="http://ucmdb.hp.com">
        <addDataReturn>a test response from addData!</addDataReturn>
    </addDataResponse>
</soapenv:Body>
```

Modifying the Data Receiver

A Java client can implement the classes contained in **UCMDBDataReceiver.jar** and call the web service in the same manner as Jython. In addition, the unwrapped methods may also be called. A Javadoc exists for the **UCMDBDataReceiver.jar** classes. The source code below shows how to use these essential methods to wrap the data in a SOAP message and send it to the receiver over HTTP.

The process is to create a **UCMDBDataReceiverServiceLocator** object, then assign the **UCMDBDataReceiverEndPointAddress** to the URL of the data receiver.

To send data, the locator's **getUCMDBDataReceiver** method is called to create a **UCMDBDataReceiver** object. The **UCMDBDataReceiver** object implements the methods to actually send the add/change/delete data. There are three identical code blocks to process each type of request.

The source code for the **SendDataToReceiver** class is listed below. Highlighted objects and methods are the essential elements to use.

```
/**
* Test SendData for the UCMDB Data Receiver for the UCMDB Web Service Push
Adapter
*/
package com.hp.ucmdb;
import com.hp.ucmdb.SendDataToReceiver;
```

```
/**
* TestSendData can be used to verify the SOAP classes are working.
* TestSendData creates a SendDataToReceiver class and invokes its SendData
method.
* a response String is returned.
* The test URL is typically appended with "?wsdl" to get the WSDL of the
service.
*/
public class TestSendData {
    /**
    * @param args - test SOAP message.
    *  optional arguments [0] a test string [1] a service endpoint URL of a
    Data Receiver.
    * the default URL is sent the incoming argument as a test message.
    * the default URL is
    "http://localhost:8080/UCMDBDataReceiver/services/UCMDBDataReceiver".
    * If any errors are encountered, TestClient will attempt to throw
    exceptions.
    */
    public static void main(String[] args) {
        // use test message if supplied, otherwise supply a default test string
        String teststring = new String("Test SOAP message from
        UCMDBDataReceiver TestSendData.");
        if(args.length > 0) {
            teststring = args[0];
        }
        // use test URL if supplied, otherwise supply the default URL
        String URL = new String("");
        if(args.length > 1) {
            URL = args[1];
        }
        // return response
        String response = new String("");
```

```
        // perform the tests
        try {
            if(URL.equals("")) {
                UCMDBDataReceiverServiceLocator locator = new
                UCMDBDataReceiverServiceLocator();
                UCMDBDataReceiver receiver = locator.getUCMDBDataReceiver();
                URL = locator.getUCMDBDataReceiverAddress();
                System.out.println("TestClient: tested
                URL="+locator.getUCMDBDataReceiverAddress());
                System.out.println("TestClient: receiver="+receiver.toString
                ());
            }
            SendDataToReceiver sdtr = new SendDataToReceiver();
            // this sends a test push and gets a response message
            response = sdtr.SendData("add", URL, args[0]);
            System.out.println("Response received was:"+response);
        } catch (Exception e) {
            System.out.println("TestClient: Remote Error:");
            e.printStackTrace();
        }
    }
}
```

Source code is also included in the **UCMDBDataReceiver.jar** file for the other classes:

- TestClient.java
- UCMDBDataReceiver.java
- UCMDBDataReceiverProxy.java
- UCMDBDataReceiverService.java
- UCMDBDataReceiverServiceLocator.java
- UCMDBDataReceiverSoapBindingStub.java

The source was generated in the Eclipse IDE, then modified. Exercise caution when modifying the UCMDB code, as much of it is auto-generated to match the SOAP specification and the UCMDB data receiver.

Javadoc

A fully-commented **javadoc** is provided with the generic web service push adapter. The **javadoc** is included in the docs folder **javadoc**. Start with **index.html**. The overview page provides access to the documentation for all classes and methods in the SDK.

All Classes

- **SendDataToReceiver:** API for the web service wrapper
- **TestClient:** test client to verify connectivity to an service endpoint
- **UCMDBDataReceiver:** web service wrapper

The rest are automatically generated by the web service builder:

- UCMDBDataReceiverProxy
- UCMDBDataReceiverService
- UCMDBDataReceiverServiceLocator
- UCMDBDataReceiverSoapBindingStub

Overview

Basic usage of the SDK, including source code examples, is explained in the documentation in the package. This **javadoc** is for the UCMDB web service push adapter. The API may be called from Jython or Java.

The SDK provides two source samples, **TestClient** and **SendDataToReceiver**. **TestClient** provides a very limited test of the responding local client. **SendDataToReceiver** is the main class used to send data to a web service.

First, use this SDK (mainly the enclosed WSDL) to implement a UCMDB data receiver to communicate with this web service. Then use this SDK to create a push adapter in the UCMDB to push UCMDB TQL result data to the data receiver. Basic usage of this API is described below, with both Jython and Java implementations.

Implementing SendDataToReceiver()

SendDataToReceiver() wraps all functions with a single method:

- Jython: `SendDataToReceiver("add",yourURL,"Hello!")`
- Java: `SendDataToReceiver("add",yourURL,"Hello!");`

Or, create a **SendDataToReceiver** object (for example, to manipulate other settings) and then call the **SendData** method separately, as shown here:

- Jython:

```
rcvr = SendDataToReceiver()
responseMsg = rcvr.SendData("add", yourURL, "Hello!")
```

- Java:

```
SendDataToReceiver rcvr = new SendDataToReceiver();
String responseMsg = rcvr.SendData("add", yourURL, "Hello!");
```

Or, if you need to do it a step at a time, you can do the following:

1. Create a new **UCMDBDataReceiverServiceLocator()** object x, then set the object's endpoint address later, shown here:
   - **Jython:**

```
x = UCMDBDataReceiverServiceLocator()
x.setUCMDBDataReceiverEndPointAddress(URL)
```

   - **Java :**

```
UCMDBDataReceiverServiceLocator x = new UCMDBDataReceiverServiceLocator();
x. setUCMDBDataReceiverEndPointAddress(URL);
```

2. Then, create a UCMDBDataReceiver with
   - **Jython:** `y = x.getUCMDBDataReceiver()`
   - **Java:** `UCMDBDataReceiver y = x.getUCMDBDataReceiver();`
3. Then, send the data via the SOAP web service like this:

- **Jython:**
  - `y.addData(yourData)`
  - or `y.updateData(yourData)`
  - or `y.deleteData(yourData)`
- **Java:**
  - `y.addData(yourData);`
  - or `y.updateData(yourData);`
  - or `y.deleteData(yourData);`

4. It may be necessary to test connectivity, then if successful reuse the same locator object to return **UCMDBDataReceiver** to use for data transfer.

The classes contain no destructors and do not perform memory management.

Mapping File Reference

Using Mappings

A mapping must be created for each target attribute in the transformed XML output. The mappings specify where and how to obtain the data. If the data is in another corresponding attribute in UCMDB, then a direct mapping is used.

To pull data from multiple attributes, or attributes from the UCMDB CI's child or parent CI's attributes, other complex mappings may be necessary. The mapping schema below shows all possible mappings.

The mapping file is an XML file which defines which CI/Relationship types in UCMDB are mapped to which CI/Relationship types in the target data store. The format is explained in detail below. The mapping file controls which CI and relationship types are pushed, as well as controlling exactly which attributes are pushed.

A mapping entry exists for each attribute to be pushed to the target MDR. Each mapping entry may consist of one or more attributes in the raw UCMDB push data. Mapping entries allow completely granular control of the final structure and naming of the data to be pushed to the target MDR.

Direct Mappings

Mappings transform one data model to another (in this case, the UCMDB to the push target MDR). Transformations may be simple, in the case of a 1:1 relationship between the UCMDB attribute and the target, they differ only by name and perhaps type.

Most attribute mappings are direct. For example, the server name "ServerX", may be represented in UCMDB as a CI of type **unix** with an attribute name of **primary_server_name**, of type **string** with a length of 50. The target MDR's data model may specify the same logical entity with a CI type of **linux**, with an attribute name of **hostname** with a type of **char[]** with a maximum length of 250. Direct mappings can accomplish all these aforementioned types of translation tasks.

Here is an example of a direct mapping:

```
 <target_attribute name="dns_domain" datatype="char">
<map type="direct" source_attribute="domain_name" />
</target_attribute>
```

This direct mapping maps the UCMDB attribute **dns_domain** to the **domain_name** attribute in the target data model.

Use the **char** data type regardless of the actual data type, unless it is necessary to use the actual data type.

Complex Mappings

More complex mappings enable additional transformations:

- To map attribute values from multiple CIs to one target CI.
- To map attributes of children CIs (those having a **container_f** or contained relationship) to the parent CI in the target data store. For example, setting a value called **Number of CPUs** on a target Host CI. Another example could be setting the value **Total Memory** (by adding up the memory size values of all memory CIs of a host CI in UCMDB) on a target Host CI.
- To map attributes of parent CIs (those having a **container_f** or contained relationship) on the target data store's CI. For example, setting a value called **Container Server** on a target attribute called **Installed Software** CI, by getting the value from the containing host of the software CI in UCMDB.

Below is an example of a complex mapping, using two source attributes separated by a comma character, to create the target attribute **os**":

```
<target_attribute name="os" datatype="char">
```

```
    <map type="compoundstring">
        <source_attribute name="discovered_os_name" />
        <constant value="," />
        <source_attribute name="host_osinstalltype" />
    </map>
</target_attribute>
```

Reversing Link Directions

It is possible that the UCMDB contains data that differs in structure from source to source. For example, the relationship between an IpAddress CI and an Interface CI may be a **parent**, as may occur with the HP Network Node Manager integration. Or it may be a **containment** link as is commonly created by Universal Discovery. Furthermore, the direction of these links are opposite to each other.

It is currently not possible to reverse the direction of links in the mappings file. Reversal of the **_end1** and **_end2** variables either switches the order of the data in the transformed XML or the link is missing in the source data.

One possible solution to this problem is to define an Enrichment rule as follows:

1. The enrichment's TQL part is a subset of a TQL that is used by the push adapter. This TQL in particular selects all the links that are in the opposite direction of what is desired in the transformed xml.
2. The enrichment part defines a new link of the correct direction and desired type.
3. Enrichment is activated and then creates the correct links.
4. The integration job TQL now refers to the enriched link rather than the original link.
5. The <link> mappings in the push adapter then refer to the enriched link as well and produce a set of links consistent in type and direction.

Mapping File Schema

Element Name and Path	Description	Attributes
integration	Defines the mapping contents of the file. Must be the outermost block in the file except for the beginning line and any comments.	
info (integration)	Defines information about the data repositories being integrated.	
source (integration > info)	Defines information about the source data repository.	1. **Name:** type **Description:** Name of the source data repository. **Is required?:** Required **Type:** String 2. **Name:** versions **Description:** Version(s) of the source data repositories. **Is required?:** Required **Type:** String 3. **Name:** vendor **Description:** Vendor of the source data repository. **Is required?:** Required **Type:** String

Element Name and Path	Description	Attributes
target (integration > info)	Defines information about the target data repository.	1. **Name:** type **Description:** Name of the source data repository. **Is required?:** Required **Type:** String 2. **Name:** versions **Description:** Version(s) of the source data repository. **Is required?:** Required **Type:** String 3. **Name:** vendor **Description:** Vendor of the source data repository. **Is required?:** Required **Type:** String
targetcis (integration)	Container element for all CIT mappings.	
source_ci_type_tree (integration > targetcis)	Defines a source CIT and all of the CI types which inherit from it.	1. **Name:** name **Description:** Name of the source CIT. **Is required?:** Required **Type:** String 2. **Name:** mode **Description:** The type of update required for the current CI type. **Is required?:** Required **Type:** One of the following strings: a. **insert:** Use this only if the CI does not already exist. b. **update:** Use this only if the CI is known to exist. c. **update_else_insert:** If the CI exists, update it; otherwise, create a new CI. d. **ignore:** Do nothing with this CI type.

Element Name and Path	Description	Attributes
source_ci_type (integration > targetcis)	Defines a source CIT without the CI types which inherit from it.	1. **Name:** name **Description:** Name of the source CIT. **Is required?:** Required **Type:** String 2. **Name:** mode **Description:** The type of update required for the current CI type. **Is required?:** Required **Type:** One of the following strings: a. **insert:** Use this only if the CI does not already exist. b. **update:** Use this only if the CI is known to exist. c. **update_else_insert:** If the CI exists, update it; otherwise, create a new CI. d. **ignore:** Do nothing with this CI type.
target_ci_type (integration > targetcis > source_ci_type -OR- integration > targetcis > source_ci_type_tree)	Defines a target CIT.	1. **Name:** name **Description:** Target CI type name. **Is required?:** Required **Type:** String 2. **Name:** schema **Description:** The name of the schema that will be used to store this CI type at the target. **Is required?:** Not Required **Type:** String 3. **Name:** namespace **Description:** Indicates the namespace of this CI type on the target. **Is required?:** Not Required **Type:** String

Element Name and Path	Description	Attributes
targetprimarykey (integration > targetcis > source_ci_type) -OR- (integration > targetcis > source_ci_type_tree -OR- (integration > targetrelations > link) -OR- (integration > targetrelations > source_link_type_tree)	Identifies target CIT primary key attributes.	
pkey (integration > targetcis > source_ci_type > targetprimarykey -OR- integration > targetcis > source_ci_type_tree > targetprimarykey -OR- (integration > targetrelations > link > targetprimarykey) -OR- integration > targetrelations > source_link_type_tree > targetprimarykey)	Identifies one primary key attribute. Required only if mode is **update** or **insert_else_update**.	

Element Name and Path	Description	Attributes
target_attribute (integration > targetcis > source_ci_type -OR- integration > targetcis > source_ci_type_tree -OR- integration > targetrelations > link -OR- integration > targetrelations > source_link_type_tree)	Defines the target CIT's attribute.	1. **Name:** name **Description:** Name of the target CIT's attribute. **Is required?:** Required **Type:** String 2. **Name:** datatype **Description:** Data type of the target CIT's attribute. **Is required?:** Required **Type:** String 3. **Name:** length **Description:** For string/char data types, integer size of target attribute. **Is required?:** Not Required **Type.** Integer 4. **Name.** option **Description.** The conversion function to be applied to the value. **Is required.** False **Type.** One of the following strings: a. **uppercase** – Convert to uppercase b. **lowercase** – Convert to lowercase If this attribute is empty, no conversion function will be applied.

Element Name and Path	Description	Attributes
map (integration > targetcis > source_ci_type > target_attribute -OR- integration > targetcis > source_ci_type_tree > target_attribute) -OR- (integration > targetrelations > link > target_attribute -OR- integration > targetrelations > source_link_type_tree > target_attribute)	Specifies how to obtain the source CIT's attribute value.	1. **Name.** type **Description.** The type of mapping between the source and target values. **Is required.** Required **Type.** One of the following strings: a. **direct** – Specifies a 1-to-1 mapping from source attribute's value to target attribute's value. b. **compoundstring –** Sub-elements are joined into a single string and the target attribute value is set. c. **childattr** – Sub-elements are one or more child CIT's attributes. Child CITs are defined as those with **composition** or **containment** relationship. d. **constant** – Static string 2. **Name.** value **Description.** Constant string for type=**constant** **Is required.** Only required when type=**constant** **Type.** String 3. **Name.** attr **Description.** Source attribute name for type=**direct** **Is required.** Only required when type=**direct** **Type.** String

Element Name and Path	Description	Attributes
aggregation (integration > targetcis > source_ci_type > target_attribute > map -OR- integration > targetcis > source_ci_type_tree > target_attribute > map -OR- (integration > targetrelations > link > target_attribute > map -OR- integration > targetrelations > source_link_type_tree > target_attribute > map) Only valid when the map's type is **childattr**	Specifies how the source CI's child CI attribute values are combined into a single value to map to the target CI attribute. Optional.	**Name:** type **Description.** The type of aggregation function **Is required?:** Required **Type.** One of the following strings: • **csv** – Concatenates all included values into a comma-separated list (numeric or string/character). • **count** – Returns a numeric count of all included values. • **sum –** Returns the sum of all numeric included values. • **average** – Returns a numeric average of all included values. • **min** – Returns the lowest numeric/character included value. • **max** – Returns the highest numeric/character included value.

Element Name and Path	Description	Attributes
source_child_ci_type (integration > targetcis > source_ci_type > target_attribute > map -OR- integration > targetcis > source_ci_type_tree > target_attribute > map -OR- (integration > targetrelations > link > target_attribute > map -OR- integration > targetrelations > source_link_type_tree > target_attribute > map) Only valid when the map's type is **childattr.**	Specifies from which connected CI the child attribute is taken.	1. **Name.** name **Description.** The type of the child CI **Is required.** Required **Type.** String 2. **Name.** source_attribute **Description.** The attribute of the child CI that is mapped. **Is required.** Required only if the **childAttr** aggregation type (which is on the same path) is not =count. **Type.** String

Element Name and Path	Description	Attributes
validation (integration > targetcis > source_ci_type > target_attribute > map -OR- integration > targetcis > source_ci_type_tree > target_attribute > map -OR- (integration > targetrelations > link > target_attribute > map -OR- integration > targetrelations > source_link_type_tree > target_attribute > map) Only valid when the map's type is **childatt**	Allows exclusion filtering of the source CI's child CIs based on attribute values. Used with the aggregation sub-element to achieve granularity of exactly which children attributes are mapped to the target CIT's attribute value. Optional.	1. **Name.** minlength **Description.** Excludes strings shorter than the given value. **Is Required?:** Not required **Type.** Integer 2. **Name.** maxlength **Description.** Excludes strings longer than the given value. **Is Required?:** Not required **Type.** Integer 3. **Name.** minvalue **Description.** Excludes numbers smaller than the specified value. **Is Required?:** Not required **Type.** Numeric 4. **Name.** maxvalue **Description.** Excludes numbers greater than the specified value. **Is Required?:** Not required **Type.** Numeric
targetrelations (integration)	Container element for all relationship mappings. Optional.	

Element Name and Path	Description	Attributes
source_link_type_tree (integration > targetrelations)	Maps a source Relationship type without the types which inherit from it to a target Relationship. Mandatory only if **targetrelation** is present.	1. **Name:** name **Description.** Source relationship name. **Is required?:**Required **Type.** String 2. **Name:** target_link_type **Description.** Target relationship name **Is required?:** Required **Type.** String 3. **Name:** nameSpace **Description:** The namespace for the link that will be created on the target. **Is required?:** Not required **Type:** String 4. **Name:** mode **Description:** The type of update required for the current link. **Is required?:** Required **Type:** One of the following strings: ■ **insert** – Use this only if the CI does not already exist. ■ **update** – Use this only if the CI is known to exist. ■ **update_else_insert** – If the CI exists, update it; otherwise, create a new CI. ■ **ignore** – Do nothing with this CI type. 5. **Name:** source_ci_type_end1 **Description:** Source relationship's End1 CI type. **Is required?:** Required **Type:** String 6. **Name:** source_ci_type_end2 **Description:** Source relationship's End2 CI type. **Is required?:** Required

Element Name and Path	Description	Attributes
		Type: String

Element Name and Path	Description	Attributes
link (integration > targetrelations)	Maps a source Relationship to a target Relationship. Mandatory only if **targetrelation** is present.	1. **Name:** source_link_type **Description:** Source relationship name. **Is Required?:** Required **Type:** String 2. **Name:** target_link_type **Description:** Target relationship name. **Is required?:** Required **Type:** String 3. **Name:** nameSpace **Description:** The namespace for the link that will be created on the target. **Is required?:** Not required **Type:** String 4. **Name:** mode **Description:** The type of update required for the current link. **Is required?:** Required **Type:** On the following strings: ■ **insert** – Use this only if the CI does not already exist. ■ **update** – Use this only if the CI is known to exist. ■ **update_else_insert** – If the CI exists, update it; otherwise, create a new CI. ■ **ignore** – Do nothing with this CI type. 5. **Name:** source_ci_type_end1 **Description:** Source relationship's End1 CI type **Is required?:** Required **Type:** String 6. **Name:** source_ci_type_end2 **Description:** Source relationship's End2 CI type **Is required?:** Required **Type:** String

Element Name and Path	Description	Attributes
target_ci_type_end1 (integration > targetrelations > link -OR- integration > targetrelations > source_link_type_tree)	Target relationship's End1 CI type.	1. **Name:** name **Description:** Name of the target relationship's End1 CI type. **Is required?:** Required **Type:** String 2. **Name:** superclass **Description:** Name of the End1 CI type's super-class. **Is required?:** Not required **Type:** String
target_ci_type_end2 (integration > targetrelations > link -OR- integration > targetrelations > source_link_type_tree)	Target relationship's End2 CI type.	1. **Name:** name **Description:** Name of the target relationship's End2 CI type. **Is required?:** Required **Type:** String 2. **Name:** superclass **Description:** Name of the End2 CI type's super-class. **Is required?:** Not required **Type:** String

Mapping Results Schema

Element Name and Path	Description	Attributes
root	The root of the result document.	
data (root)	The root of the data itself.	
objects (root > data)	The root element for the objects to update.	

Element Name and Path	Description	Attributes
Object (root > data > objects)	Describes the update operation for a single object and all of its attributes.	1. **Name:** name **Description:** Name of the CI type **Is required?:** Required **Type:** String 2. **Name:** mode **Description:** The type of update required for the current CI type. **Is required?:** Required **Type:** One of the following strings: a. **insert** – Use this only if the CI does not already exist. b. **update** – Use this only if the CI is known to exist. c. **update_else_insert** – If the CI exists, update it; otherwise, create a new CI. d. **ignore** – Do nothing with this CI type. 3. **Name:** operation **Description:** The operation to perform with this CI. **Is required:** Required **Type:** One of the following strings: a. **add** – The CI should be added b. **update** – The CI should be updated c. **delete** – The CI should be deleted If no value is set, then the default value of **add** is used. 4. **Name:** mamId **Description:** The ID of the object on the source CMDB. **Is required?:** Required **Type:** String

Element Name and Path	Description	Attributes
field (root > data > objects > Object -OR- root > data > links > link)	Describes the value of a single field for an object. The field's text is the new value in the field, and if the field contains a link, the value is the ID of one of the ends. Each end ID appears as an object (under <objects>).	1. **Name:** name **Description:** Name of the field. **Is required?:** Required **Type:** String 2. **Name:** key **Description:** Specifies whether this field is a key for the object. **Is required?:** Required **Type:** Boolean 3. **Name:** datatype **Description:** The type of the field. **Is required?:** Required **Type:** String 4. **Name:** length **Description:** For string/character data types, this is the integer size of the target attribute. **Is required?:** Not Required **Type:** Integer

Element Name and Path	Description	Attributes
links (root > data)	The root element for the links to update.	1. **Name:** targetRelationshipClass **Description:** The name of the relationship (link) in the target system. **Is required?:** Required **Type:** String 2. **Name:** targetParent **Description:** The type of first end of the link (parent). **Is required?:** Required **Type:** String 3. **Name:** targetChild **Description:** The type of the second end of the link (child). **Is required?:** Required **Type:** String 4. **Name:** mode **Description:** The type of update required for the current CI type. **Is required?:** Required **Type:** One of the following strings: a. **insert** – Use this only if the CI does not already exist. b. **update** – Use this only if the CI is known to exist. c. **update_else_insert** – If the CI exists, update it; otherwise, create a new CI. d. **ignore** – Do nothing with this CI type. 5. **Name:** operation **Description:** The operation to perform with this CI. **Is required?:** Required **Type:** One of the following strings: a. add – The CI should be added b. update – The CI should be updated c. delete – The CI should be deleted If no value is set, then the default value of **add** is used.

Element Name and Path	Description	Attributes
		6. **Name:** mamId **Description:** The ID of the object on the source CMDB. **Is required?:** Required **Type:** String

Customization

This section explains some of the basic procedures for common types of customization for push adapters.

Adding an Attribute

1. Ensure that the attribute is included in the TQL result.
2. Add the attribute mapping to the mappings file in the correct CI mapping section.
3. Ensure the data receiver is prepared to receive the additional attribute in the data.

Removing an Attribute

To remove an attribute, remove the attribute from the mapping file.You should also remove the attribute from the TQL if it is no longer used in the result or as a conditional node.

Adding a CI Type

1. Add the CI type to the TQL.
2. Ensure the CI type and it's attribute data appear in the TQL result (use calculate and preview).
3. Add the CI type's mapping in the mappings file. Copy another CI type's mappings to quickly create a new CI type.
4. Modify the copied XML's name and attribute mappings to correspond to the new CI type and its attributes. See "Mapping File Reference" on page 294 for the available types of mappings.

Removing a CI Type

1. Remove the CI Type from the TQL
2. Remove the mapping section for that CI type in the mappings file.

Adding Links

1. Ensure the two end CIs are present in the data.
2. Ensure the link you need to add is in fact a valid link (check in the CI type manager).
3. Add the link elements in the relationships section of the mappings xml.

Removing Links

1. Remove the link section of the link you want to remove in the mappings file.
2. If possible, remove the link from the TQL (unless it affects the efficiency or function of the TQL).

Chapter 8: Developing Generic Adapters

This chapter includes:

Instance Sync

The push and population Generic Adapter operations work with instance data. For more information about the concepts of *instance* and *root*, see "Instance-Based Population Flow" on page 222 and "Achieving Data Push using the Generic Adapter" below.

Achieving Data Push using the Generic Adapter

Data Push uses the existing Enhanced Generic Push Adapter framework with minor XML schema changes.

Note: The Generic Adapter works in instance mode (meaning it does not work with single CI types, but with collections of CIs grouped together by a main root CI). For more information, see "Instance-Based Population Flow" on page 222.

The XML schema changes needed to accommodate bidirectional mapping semantics are:

- the **<targetcis>** tag has been renamed to **<target_entities>**.
- the **<source_instance_type>** tag has been renamed to **<source_instance>**.
- the **<target_ci_type>** tag has been renamed to **<target_entity>**.
- the **<for-each-source-ci>** tag thas been renamed to **<for-each-source-entity>**.
- the header **versions** attribute has been renamed to **version** and no longer requires a decimal.

This section provides information about pushing data using the Generic Adapter Framework:

Push Overview

The Generic Adapter works on data structures that represent the TQL query result. Each adapter built over the Generic Adapter Framework will handle this data structure and push it to its required target.

The data structure is named **ResultTreeNode (RTN)**. The RTN is created according to the mapping file of the adapter and the results of the TQL query. The queries used for the Generic Adapter Framework must be root based, that is the query must contain one query node with element name **root**, or one or more relationship elements beginning with the prefix **root**. This CI or relationship serves as the root element of the query. For details, see Data Push in the *HP Universal CMDB Data Flow Management Guide*.

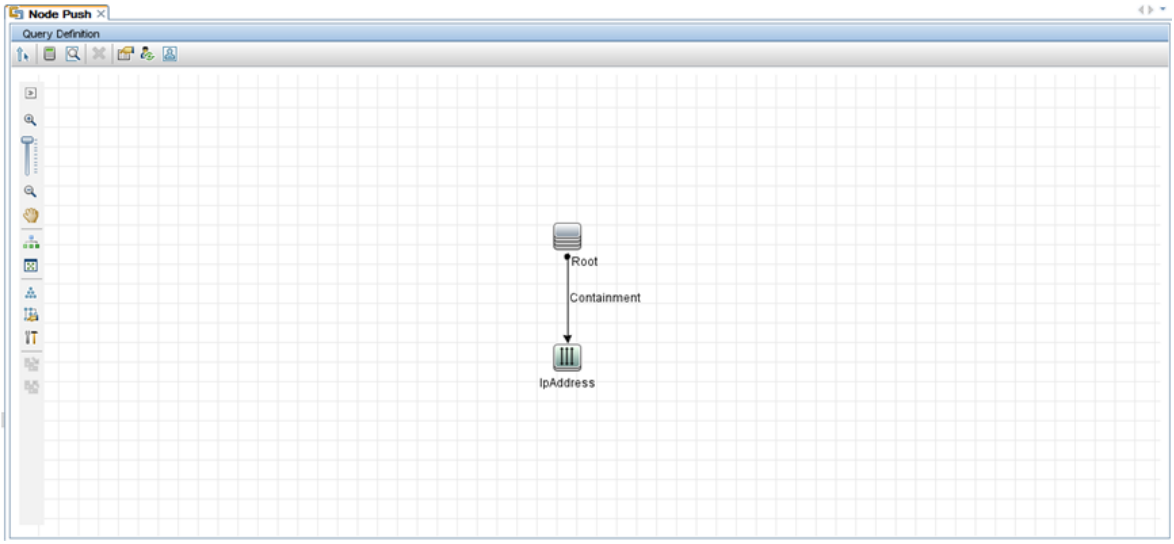
There are two basic steps involved in developing enhanced push adapters:

1. Implementing the PushAdapterConnector interface – this interface receives the data to add, update and delete as a list of RTNs, and perform the push into the target.
2. Creating the mapping file – the mapping file determines the creating of the RTN structure, by mapping CIs and attributes of the TQL result.

The Mapping File

The following example demonstrates how to create the mapping file.

In this example, we will simulate a push of a node and an IP address. We will create a TQL query called: **Node Push**, as follows:

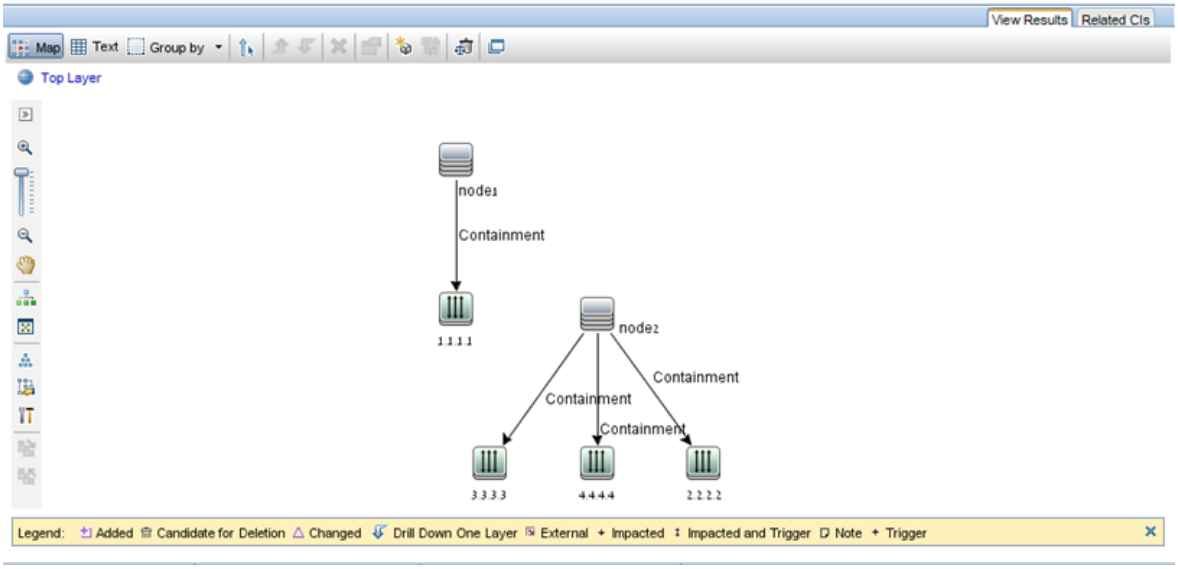


In the mapping file we create two target CI types: **Computer** and **IP**. Computer has one variable and two attributes. IP has one attribute.

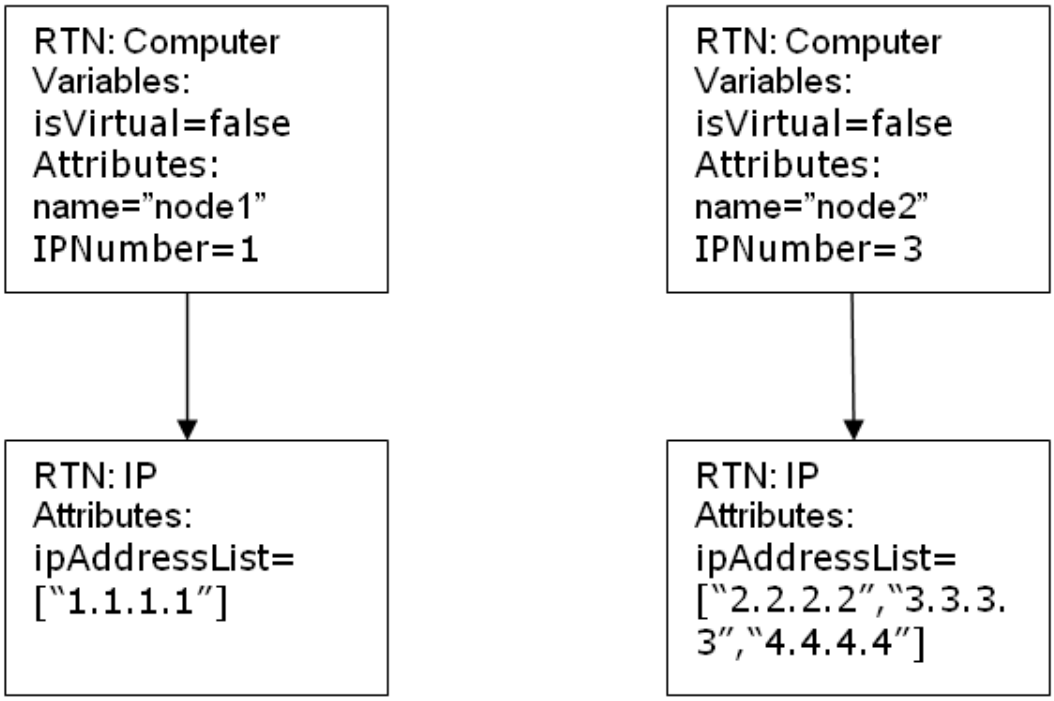
The following is the mapping XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <info>
        <source name="UCMDB" version="10.20" vendor="HP"/>
        <target name="PushProduct" version="9.3" vendor="HP"/>
    </info>

    <import>
        <scriptFile path="mappings.scripts.PushFunctions"/>
    </import>

    <target_entities>
        <source_instance query-name="Node Pusy" root-element-name="Root">
            <target_entity name="Computer" is-valid="(Root['root_iscandidatefordeletion'] == null) ? true : !Root['root_iscandidatefordeletion']">
                <variable name="isVirtual" datatype="BOOLEAN" value="PushFunctions.isVirtual(Root['root_class'])"/>
                <target_mapping name="name" datatype="STRING" value="Root['name']"/>
                <target_mapping name="ipNumber" datatype="INTEGER" value="Root.IpAddress.size()"/>
                <target_mapping name="description" datatype="STRING" value="PushFunctions.getDescription(isVirtual)"/>
                <target_entity name="IP">
                    <target_mapping name="IpAddressList" datatype="STRING_LIST" value="Root.IpAddress*.getAt('name')"/>
                </target_entity>
            </target_entity>
        </source_instance>
    </target_entities>

</integration>
```

The query results appear as follows:



Here is the RTN list built according to this mapping file:



Each root instance is mapped separately using the mapping file. Thus in this example, the PushAdapterConnector receives a list of two RTN roots.

Note: The previous push adapter had the ability to create a general mapping for a CI type. The new push adapter mapping is per TQL query. While running a push job that uses a query named **x**, the

adapter looks for the relevant mapping file (the one that has attribute: query-name=x).

You can calculate the values in the mapping file using groovy script language. For details, see "The Groovy Traveler" below.

The Groovy Traveler

Access the TQL query results in the following manner:

- **Root[attr]** returns the attribute **attr** of the Root element.
- **Root.Query_Element_Name** returns a list of the CI instances named in the TQL Query_Element_Name and that are linked to the current root CI.
- **Root.Query_Element_Name[2][attr]** returns the attribute **attr** of the third Query_Element_Name that is linked to the current root CI.
- **Root.Query_Element_Name*.getAt(attr)** returns a list of the attributes **attr** of the CI instances named Query_Element_Name in the TQL and that are linked to the current root CI.

There are additional attributes that can be accessed by the groovy traveler:

- **cmdb_id** – returns the UCMDB ID of the CI or relationship as a string.
- **external_cmdb_id** – returns the external ID of the CI or relationship as a string.
- **Element_type** – returns the element type of the CI or relationship as a string.

The import tag:

```
<import>
<scriptFile path="mappings.scripts.PushFunctions"/>
</import>
```

This means that we are declaring an import for all the groovy scripts in the mapping file. In this example, **PushFunctions** is a groovy script file that contains some static functions, and we can access them during the mapping (i.e. value=" PushFunctions.foo()")

source_instance_type

The mapping is done per TQL, the query-name value is the related TQL of the current mapping. The '*' means that this mapping file is associated with all TQL queries beginning with the prefix: **Node Push**.

```
<source_instance_type query-name="Node Push*" root-element-name="Root">
```

The source_instance_type tag designates the root element we are mapping.

root-element-name should be exactly the same as the name of the root in the TQL.

target_entity

This tag is used for the creation of the RTN.

The name attribute represents the target_entity name: name=Computer

The **is-valid** attribute is a Boolean value that is calculated during the mapping, and determines if the current target_ci is valid. Invalid target_entities are not added to the RTN. In this example, we do not want to create a target_entity instance for which the **root_iscandidatefordeletion** attribute in UCMDB is true.

The target_entity can have variables that are calculated during the mapping:

```
<variable name="vSerialNo" datatype="STRING" value="Root['serial_number']"/>
```

The variable **vSerialNo** gets the value of the **serial_number** of the current root.

The attribute of the RTN is created by the **target_mapping** tag. The result of the execution of the groovy script in the **value** field,is assigned to the RTN attribute.

```
<target_mapping name="SerialNo" datatype="STRING" value="vSerialNo"/>
```

SerialNo assigns the value of the variable **vSerialNo**.

It is possible to define a target_entity as child of another target_entity as follows:

```
<target_entity name="Portfolio">
<variable name="vSerialNo" datatype="STRING" value="Root['global_id']"/>
<target_mapping name="CMDBId" datatype="STRING" value="globalId"/>
<target_entity name=Asset">
<target_mapping name="SerialNo" datatype="STRING" value="vSerialNo"/>
</target_entity>
</target_entity>
```

The RTN **Portfolio** will have child RTN named **Asset**.

for-each-source-entity

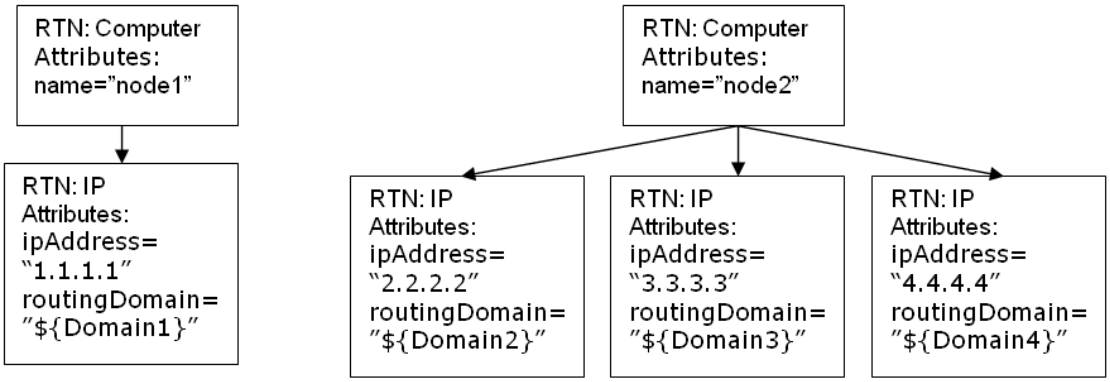
This tag lists the specific CIs of the root instance. It has the following fields:

- **source-entities=" "** – the list of CIs for which a target CI is created. This list is defined by the groovy traveler in the **Root.IpAddress** field.
- **count-index=" "** – a variable that holds the index of the CI in the current iteration of the loop.
- **var-name=" "** – the name of the CI in the current iteration of the loop.

Let's modify our example mapping file:

```
<target_entity name="Computer">
    <target_mapping name="name" datatype="STRING" value="Root['name']"/>
    <for-each-source-entity count-index="i" var-name="currIP" source-entities="Root.IpAddress">
        <target_entity name="IP">
            <target_mapping name="ipAddress" datatype="STRING" value="Root.IpAddress[i]['name']"/>
            <target_mapping name="routingDomain" datatype="STRING" value="currIP['routing_domain']"/>
        </target_entity>
</target_entity>
```

The RTN list that will build according to this mapping file will look like this:



dynamic_mapping

This tag adds the ability to create a mapping of data from the target data store during the creation of the RTN structure.

Example: Assume that the target is a database with a table named **Computer** that has an **id** column and a **name** column that is correlated to **Node.name** in UCMDB. Both columns are unique. The database also has a table named **IP** that has a referenced key to the **parentID** in the Computer table. The 'dynamic_ mapping' can create a map that stores the name and id as <name,id>. Based on this map, the Adapter can match ids with computers and can push the correct value to the **parentID** attribute in the IP table. You can use this map to assign a value to the parentID attribute while creating the RTN.

The mapping is determined by the **map_property**. The dynamic_mapping is executed once for each chunk.

```
<dynamic_mapping name="IdByName " keys-unique="true">
```

The **name** attribute represents the name of the map. The **keys-unique** attribute indicates if the keys are unique (each key is mapped to one value, or to a set of values).

The name of the map in this example is **IdByName** and it has unique keys. In order to access the map in the script, execute the following command:

```
DynamicMapHolder.getMap('IdByName')
```

It returns a reference to that map.

The **map_property** tag creates the property on which the mapping is based.

Example:

```
<map_property property-name="SQLQuery" datatype="STRING"
property-value="SELECT name, id FROM Computer"/>
```

In this example the name of the property is **SQLQuery** and its value is an SQL statement that creates the map. The implementation of the methods **retrieveUniqueMapping** and **retrieveNonUniqueMapping** for the PushConnector interface will determine the actual content of the returned map.

Global Variables

The following global variables are accessible to the groovy script in the mapping file:

- **Topology** – Type: Topology. An instance of the topology of the current chunk.
- **QueryDefinition** - Type: QueryDefinition. An instance of the query definition of the current TQL.
- **OutputCI** – Type: ResultTreeNode. The RTN of the root element in the current tree mapping.
- **ClassModel** – Type: ClassModel. An instance of the class model.
- **CustomerInformation** – Type: CustomerInformation. Information about the customer running the job.
- **Logger** – Type: DataAdapterLogger. This logger is available in the adapter for writing logs to the UCMDB logging framework.

Write Groovy Scripts

In this section we create the **PushFunctions.groovy** file. This file will contain static functions that are used during the mapping of the root instance.

```
package mappings.scripts

public class PushFunctions {

	public static boolean isVirtual(def nodeRole){
		return isListContainsOne(def list, "MY_VM", "MY_SIMULATOR");
	}



public static String getDescription(boolean isVirtual){
		if(isVirtual){
			return "This is a VM";
		}
		else{
			return "This is physical machine";
		}
	}



private static boolean isListContainsOne(def list, ...stringList){
		//returns true if the list contains one of the values.
	}
}
```

Implement PushAdapterConnector Interface

The implementation should support the following basic steps:

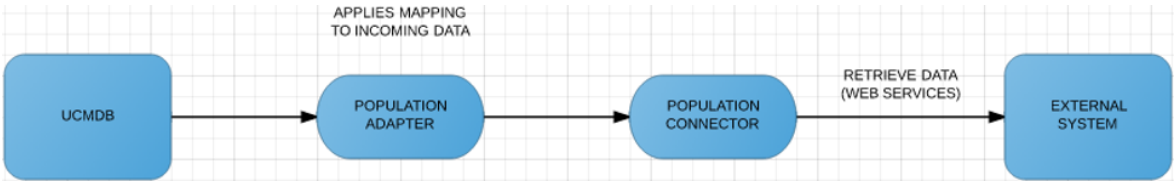
```
public class PushExampleAdapter implements PushAdapterConnector

{

public UpdateResult pushTreeNodes(PushConnectorInput input) throws
DataAccessException{

// 1. build an UpdateResult instance - the UpdateResult is used to return
mappings between the sent ids to the actual ids that entered the data store.
// Also has an update status which allows to pass the status of data that was
actually pushed, detailed status reports on failed IDs, and actions actually
performed on successful ids.
// 2. handle the data:
// a. handle data to add. Can be retrieved by:
input.getResultTreeNodes.getDataToAdd();
//  b. handle data to update.
//  c. handle data to delete.
```

```
// 3. Return the Update result.
       }

public void start(PushDataAdapterEnvironment env) throws DataAccessException{
       // this method is called when the integration point created,
or when the adapter is reloaded
       //(i.e after changing one of the mapping files
       // and pressing ‘save’).

}



public void testConnection(PushDataAdapterEnvironment env) throws
DataAccessException {
               // this method is called when pressing the 'test
connection' button in the
               //creation of the integration point.
               // For example if we push data to RDBMS this method
can create a connection
               //to the database and will run a dummy SQL statement.
               // If it fails it writes an error message to the log
and throws an exception.
       }



Map<Object, Object> retrieveUniqueMapping(MappingQuery mappingQuery){
//This method will create the map according to the given mappingQuery. It will
be called in the
// mapping stage of the adapter execution, before the ‘UpdateResult
pushTreeNodes’ method.
// This method is called when the ‘keys-unique’ attribute of the ‘dynamic_
mapping’ tag is true.
}

Map<Object, Set<Object>> retrieveNonUniqueMapping(MappingQuery mappingQuery){
 // This method is called when the ‘keys-unique’ attribute of the ‘dynamic_
mapping’ tag is false.
//  In this case a key can be mapped to several values.
}
}
```

Achieving Data Population using the Generic Adapter

This section contains the following topics:

The Population Framework Architecture



The mechanism is similar to that of the Push Adapter framework, meaning that a user of this Population Adapter framework must provide a mapping file and a connector implementation, and bundle them together in a UCMDB adapter package.

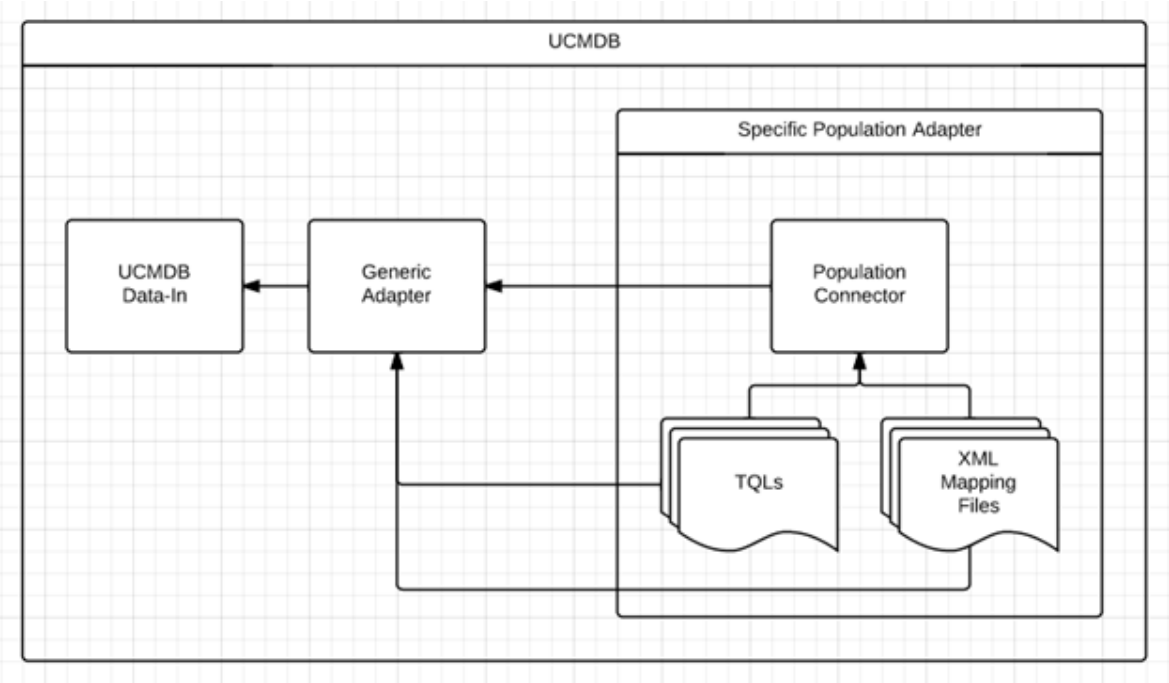
The operation flow contains the following steps

1. The UCMDB user triggers the population operation from the UI.
2. The command is sent to the population adapter.
3. The population adapter calls the population connector and retrieves data in chunks.
4. The population adapter applies the defined mapping on data from each chunk and sends it forward to the UCMDB Server.

Main Artifacts involved in Population

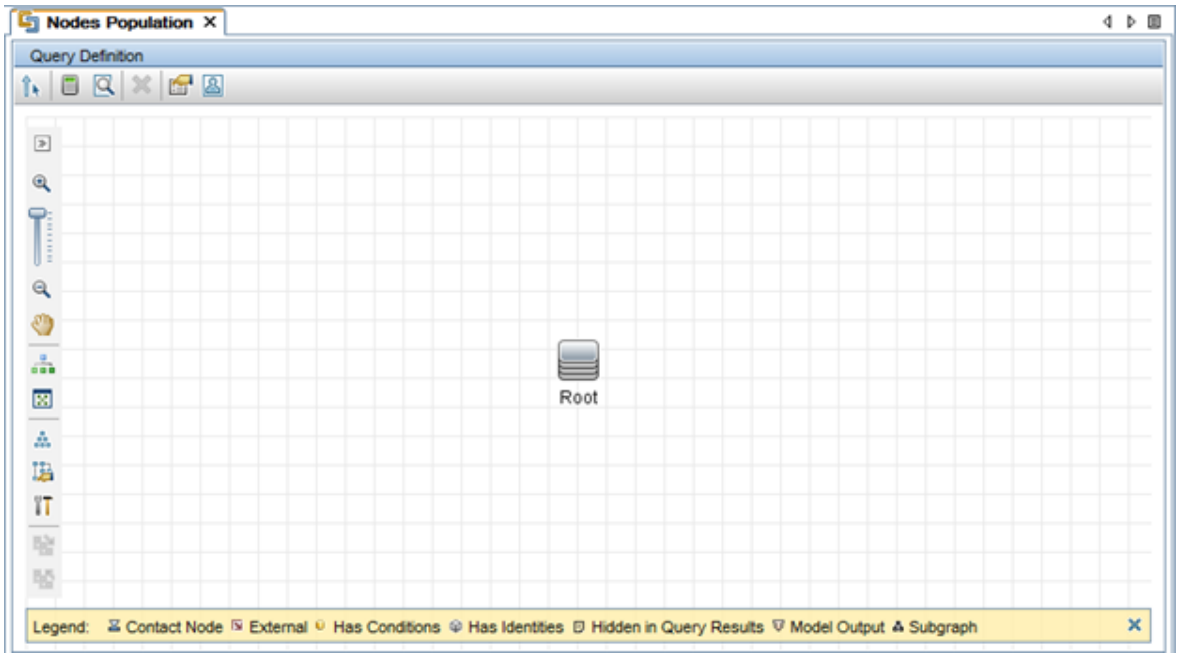
The main artifacts involved in population are:

- the TQL queries hat specify the data that will be populated in UCMDB
- the XML mapping files that specify how the connector-returned data will be mapped to UCMDB
- required data
- the population connector responsible for retrieving the external system data and returning it to the UCMDB Generic Adapter.



Population TQL Queries

The role of a population TQL query is to indicate the data that will be populated in UCMDB. For example, the TQL in the following figure is used to bring Node instances in UCMDB.



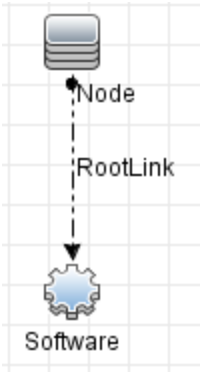
The population connector is responsible for understanding the population TQL queries and providing the required data from the external system.

Population Mapping Files

The XML mapping files have the same purpose as for push operations, except that the direction is reversed. These mapping files describe how the data returned by the connector will be mapped to the UCMDB data.

The information provided here is relevant for population mapping and is not already mentioned for the Enhanced Generic Push Adapter.

Following is an example of a mapping for UCMDB Nodes and Running Software. The first image shows the Nodes and Software Population TQL query. The second image displays the Nodes and Software Population Mapping.




```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Nodes And Software Population" root-element-name="PC">
        <!-- need to match case in UCMDB TQL -->
        <target_entity name="RootLink">
            <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
        </target_entity>
        <target_entity name="Node" type="Util.getNodeType(PC)">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC['description']"/>
        </target_entity>
        <target_entity name="Software">
            <target_mapping name="name" datatype="STRING" value="PC.Programs[0]['name']"/>
            <target_mapping name="root_container_name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="product_name" datatype="STRING" value="'vmware_hypervisor'"/>
        </target_entity>
    </source_instance>
</target_entities>
```

This population job fetches data from an external system in the form of the ResultTreeNode (RTN) PC. The ResultTreeNode API was introduced by the Enhanced Generic Push Adapter and can be found in the

push-interfaces.jar file located in the UCMDB Server **lib** folder. For more information, see "Achieving Data Push using the Generic Adapter" on page 315.

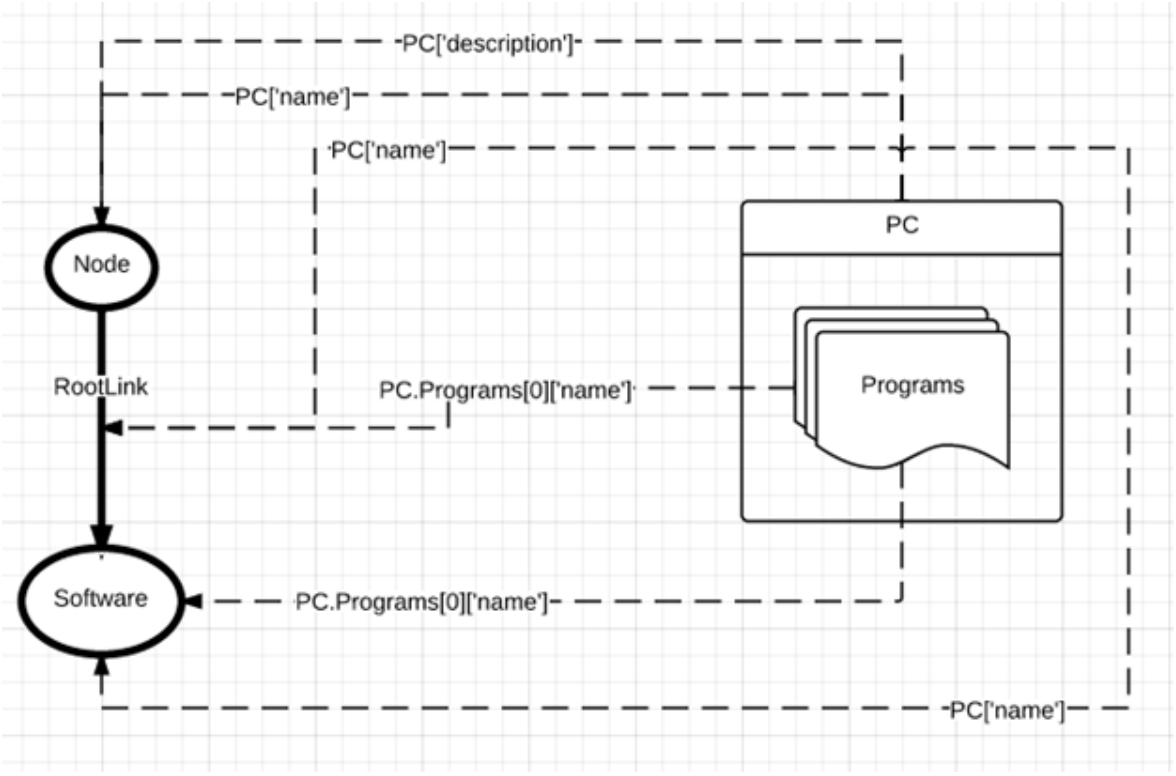
The PC RTN contains general node information in the form of attributes as well as an embedded Programs entity that contains software type entities with the relevant attributes.

One PC instance will be mapped to 3 entities in UCMDB:

- a Node CI
- a Running Software CI
- a Composition link CI

For more information about the format of the PC instance, see "Population Request Output" on page 336.

The way in which connector data is mapped to UCMDB data is shown in the following figure:



Let's analyze the key lines:

```
<!--The query name must match the one selected in the UI-->
<source_instance query-name="Nodes And Software Population" root-element-name="PC">
```

The **source_instance** definition states that we will bring entities into UCMDB and the UCMDB topology that groups these entities is defined by the Nodes and Software Population TQL query. In addition, the data structure returned by the connector that will be used to create the UCMDB data is a **ResultTreeNode** named **PC**.

```
<target_entity name="RootLink">
    <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
</target_entity>
```

The **target_entity** tag states that a new UCMDB entity starts here and this entity corresponds to the **RootLink** element inside the Nodes and Software Population TQL query. This indication also includes the UCMDB CI type of the new entity.

The **RootLink** entity that will be created will have one attribute, **name**, whose value will be something like **Computer_22 has MySQL Server**.

This sample mapping uses manual link population. We recommended using the automatic approach described in "Automatic Link Population" below.

The Population type Attribute

```
<target_entity name="Node" type="Util.getNodeType(PC)">
```

Note that the Node entity has a **type** attribute. This **type** indicates the exact CI type that this entity will have in UCMDB. The **type** attribute is not mandatory, because the entity's default creation type is obtained from the TQL element it refers to (in this case, Node). However, if we want to return multiple instances of the UCMDB Node CI type and some of the instances are Windows while others are Unix, we can use the **type** attribute to specify the exact UCMDB creation type. So in this case, we create a **getNodeType** function inside the **Util** function script file, which receives as input the PC tree and returns the valid UCMDB CI type identifier ("unix" for Unix and "nt" for Windows).

> **Note:** The **target_entity type** attribute is only available in the context of a Population flow, and its value must be a valid Groovy expression.

We can describe the creation of the Software entity in the same way.

Automatic Link Population

In the mapping sample in "Population Mapping Files", we saw what is needed to explicitly map a populated link. A mapped target entity must be present for each TQL link element, such as the one shown below:

```
<target_entity name="RootLink">
    <target_mapping name="name" datatype="STRING" value="PC['name'] + 'has' + PC.Programs[0]['name']"/>
</target_entity>
```

Using the Generic Adapter automatic link population mechanism, we no longer need to map the TQL link elements with mapping sections such as the one shown above. The framework will generate a link CI instance of the type specified in the TQL with empty properties. This operation will be performed for all the links in the population TQL query.

The sample mapping results in a link CI of type composition being created that has as link ends (end1 and end2) the Node and Running Software CI instances.

You should use Manual Link Population if the link you are populating requires:

- Dynamic link type (using the **type** attribute)
- Link properties

Manual Link Population

The Generic Adapter achieves the population of links by defining (mapping) the three entities required by a link:

- the Link entity
- the Link's End 1 entity
- the Link's End 2 entity

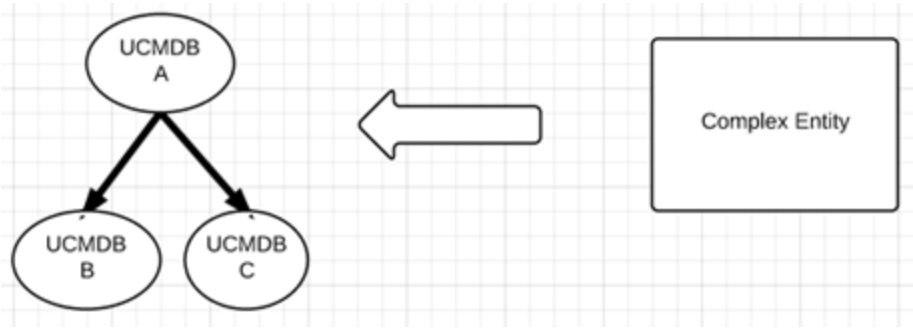
Let's analyze the mapping example shown in "Population Mapping Files" on page 327. In this case, we are populating three entity types in UCMDB: Nodes, Running Software, and the Composition link between them. Because we want to populate a link (the link named **RootLink** of type Composition), we also need to map the two link ends. Thus, from looking at the TQL query we see that the entities that need mapping are Node (end 1) and Software (end 2). The way the Generic Adapter framework understands the link structure is by looking at the created entity's element name and definition in the TQL query. Because the population job must also bring instances of Node and Running Software, the needed ends' mapping is already in place.

Types of Link Population

There are two types of link population situations:

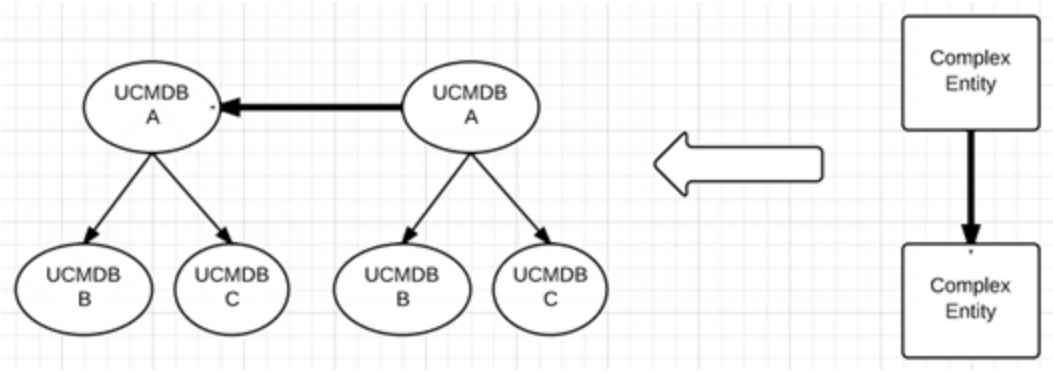
- Decomposing a complex external entity in multiple related UCMDB entities

In this case, a complex external entity such as PC is converted into the UCMDB **Node** and **Running Software** types, which need to be linked by a **Composition** link. This type of link only exists in the context of UCMDB.



- Links between complex external entities

  In this case, we need to model a link between two complex external entities such as PCs.



The Population Connector

The population connector is responsible for retrieving external system data. This data is passed on to the Generic Adapter in the established API format (**ResultTreeNode**), which is then mapped to the UCMDB data structures and inserted in UCMDB through the data-in process.

Similar to the push connector, the population connector can be implemented in both Java and Groovy and must implement the Population Connector Java Interface shown in the figure below.

To configure the population connector, add the following line in the adapter configuration XML file:

```
<adapter-settings>
    <adapter-setting name="PopulationConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPopulationConnector</adapter-setting>
```

```java
public interface PopulationAdapterConnector extends GenericConnector, DynamicMappingConnector {

    /**
     * Executes a population request and provides the entities that will be populated in the format of
     * {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode}
     * Used for both population and federation with the flow type flag which is present in the
     * {@link com.hp.ucmdb.adapters.population.connector.PopulationConnectorInput}
     *
     * @param input population request
     * @return a population response
     * @throws DataAccessException
     */
    PopulationConnectorOutput populate(PopulationConnectorInput input) throws DataAccessException;

    /**
     * Returns the collection of queries the current connector supports for population
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getPopulationQueries(DataAdapterEnvironment env);

    /**
     * Returns the collection of queries the current connector supports for federation
     * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
     * <p/>
     * The method also supports return of queries with their folders path: E.g.
     * "Folder/Secondary Folder/Query Name 1"
     * "Folder/Secondary Folder/Query Name 2"
     *
     * @param env the adapter environment
     * @return a collection of the supported queries
     */
    Collection<String> getFederationQueries(DataAdapterEnvironment env);

    /**
     * Update target id (global id) for each source object.
     *
     * @param idMapping mapping between source id (external id) and target id (string)
     */
    void updateGlobalIDsFromTarget(FCmdbExtenalToTargetIdMappingSet idMapping);

    /**
     * This methods reports population queries resources used by the adapter.<br>
     * This allows editing these resources directly from the Integration Studio.
     *
     * @param input  the requested information
     * @param output the returned resources
     * @see com.hp.ucmdb.federationspi.adapter.resource.PushQueriesResourceLocator
     */
    void locatePopulationQueriesResources(DataAdapterEnvironment env, LocatePopulationQueriesResourcesInput input,
                                          LocatePopulationQueriesResourcesOutput output);

    /**
     * Returns the collection of classes the current adapter supports for query.<br>
     * Notes:<br>
     * 1. This method is independent of the adapter life cycle (i.e. start/shutdown methods).<br>
     * 2. If adapter configuration (xml) defines supported classes, this method doesn't need to be implemented (return null).<br>
     *
     * @param env the Adapter env
     * @return collection of the supported classes
     * @throws DataAccessException in case of an error
     */
    Collection<SupportedClassConfig> getSupportedClasses(DataAdapterEnvironment env);
}
```

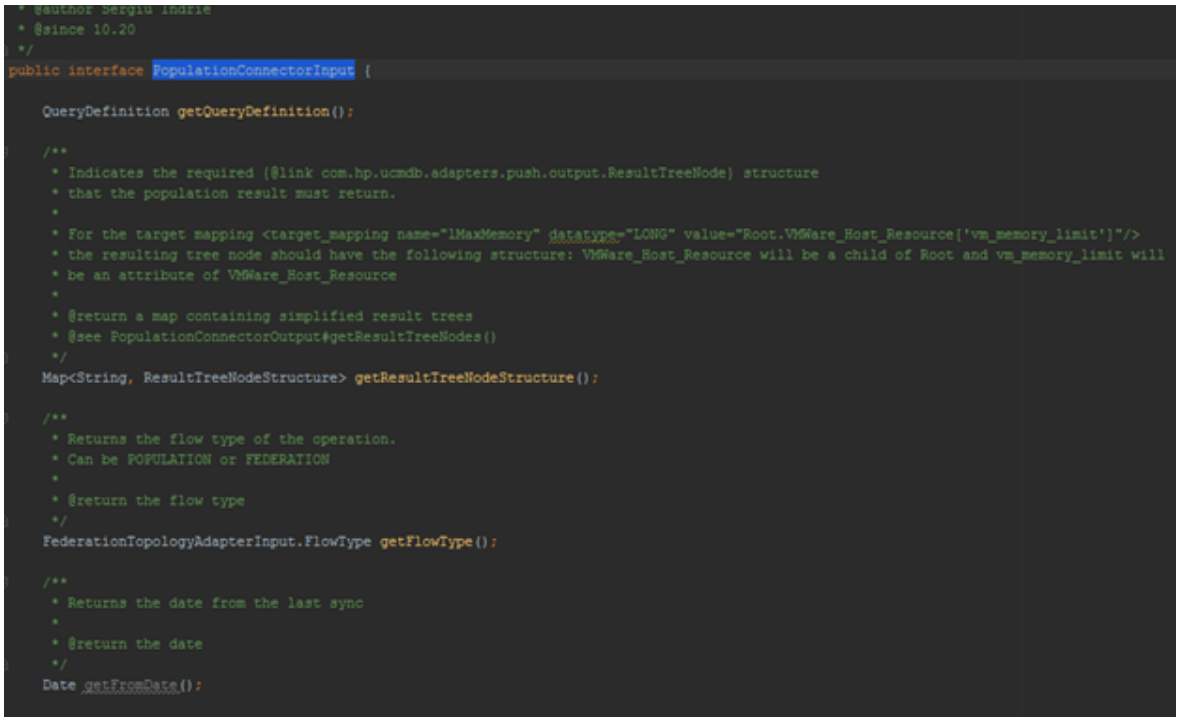
The first method, `populate`, is the main connector method responsible for retrieving the data from the external system. This method receives as input a population TQL query and returns the results in the generic *ResultTreeNode* format. For more information, see "Achieving Data Push using the Generic Adapter" on page 315. Along with the main business data, the connector also returns status and chunking information.

The second method, `getSupportedQueries`, indicates the TQLs that are supported by the population connector.

The third and fourth methods refer to more advanced use cases, pushing back the IDs of the populated data and locating the relevant population resources within the adapter for a specific query. For more information about these APIs, see the **push-interfaces.jar** file.

Population Request Input

A population request is defined by a **QueryDefinition** object that describes the UCMDB population query. The population connector is responsible for reading this query object and translating it into the external system's query language.



```
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationConnectorInput {

    QueryDefinition getQueryDefinition();

    /**
     * Indicates the required {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode} structure
     * that the population result must return.
     *
     * For the target mapping <target_mapping name="lMaxMemory" datatype="LONG" value="Root.VMWare_Host_Resource['vm_memory_limit']"/>
     * the resulting tree node should have the following structure: VMWare_Host_Resource will be a child of Root and vm_memory_limit will
     * be an attribute of VMWare_Host_Resource
     *
     * @return a map containing simplified result trees
     * @see PopulationConnectorOutput#getResultTreeNodes()
     */
    Map<String, ResultTreeNodeStructure> getResultTreeNodeStructure();

    /**
     * Returns the flow type of the operation.
     * Can be POPULATION or FEDERATION
     *
     * @return the flow type
     */
    FederationTopologyAdapterInput.FlowType getFlowType();

    /**
     * Returns the date from the last sync
     *
     * @return the date
     */
    Date getFromDate();
```

In addition to the QueryDefinition object, there are:

- **getResultTreeNodeStructure** – Indicates the required structure that the population result must return.
- **getFlowType** – Used to determine if the request to the connector is of type POPULATION or FEDERATION.

getFromDate – Indicates the date from the last synchronization. If the date is null, then the FULL POPULATION runs otherwise the Diff POPULATION runs (if the flow type is FEDERATION the getFromDate method will always return null).

A sample population request is shown in the following figure:

In this example, the request contains the **Nodes Population** query. We can see that the query contains only one TQL element of type **Node**.

ResultTreeNodeStructure

To implement your **PopulationAdapterConnector**, you must read the UCMDB Population TQL, understand what UCMDB is asking for, and provide the results using external system entities. For example, UCMDB may be asking for all Nodes related to Business Service instances in your external system, and it might be that the external system equivalent for a computer is **PC**, which is related to a **Service** entity. Thus, your population connector must return instances of **PC** connected to instances of **Service**. In this case, the mapping will look something like this:

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="PC">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService">
            <target_mapping name="name" datatype="STRING" value="PC.Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC.Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

In this case, to return the **PC** instances related to **Service** instances, we are returning a **PC** RTN that contains **Service** as a child node. However, we could have chosen to create the mapping in the format of a **Service** RTN with a **PC** child like this (rendering the mapping invalid):

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="Service">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="Service.PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService">
            <target_mapping name="name" datatype="STRING" value="Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

Thus, to aid the development of the population connector, the population request sent by the Generic Adapter also includes the RTN structure of the data used in the mapping file. This indicates to the implementing connector the needed format of the returned RTN.

In the first case the **ResultTreeNodeStructure** is:

PC
- name
  Service
  - name
  - description

And in the second case the **ResultTreeNodeStructure** is:

Service
- name
- description
  PC
  - name

Population Request Output

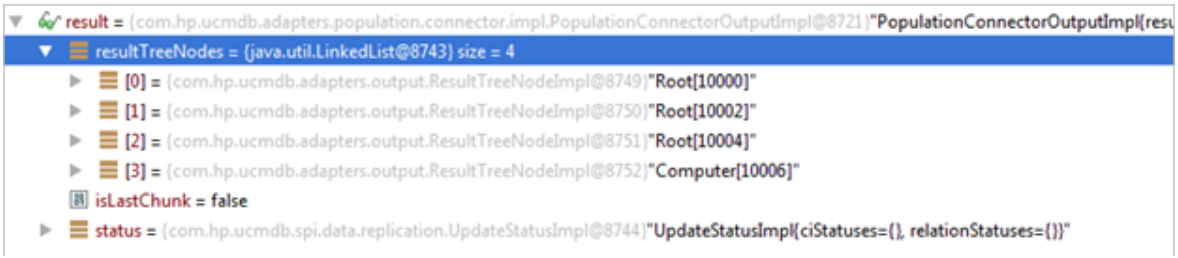
Upon processing a population request, the population connector must return a `PopulationConnectorOutput`.

```
public interface PopulationConnectorOutput {

    /**
     * The result trees representing the external entities in (@link com.hp.ucmdb.adapters.push.output.ResultTreeNode) format.
     *
     * @return a list of result trees or an empty list
     */
    List<ResultTreeNode> getResultTreeNodes();

    /**
     * Adds a result tree to the population output object.
     *
     * @param resultTreeNode the result tree to add
     */
    void addResultTreeNode(ResultTreeNode resultTreeNode);

    /**
     * This method indicates if the result received is the last chunk of data.
     *
     * @return true if this is the last chunk, false otherwise.
     */
    boolean isLastChunk();

    /**
     * Set whether this result object is the last chunk or not.
     */
    void setLastChunk(boolean isLastChunk);

    /**
     * Returns the status information about the population result.
     */
    UpdateStatus getStatus();

    /**
     * Set the population result status.
     */
    void setStatus(UpdateStatus status);

}
```

This output object contains:

- the queried data, in `ResultTreeNode` format
- status information (needed in case of failure)
- chunk information

A sample population response can be seen in the following figure:



In the response above, the connector returned four data instances (corresponding to the UCMDB Node), an empty status (signaling success) and a flag indicating that this is not the last chunk.

Population Adapter Modes

The UCMDB Adapter Framework allows two types of population adapters:

- Standard Population Adapter
  - Is characterized by the absence of the **<changes-source/>** tag in the adapter XML file
  - Will always bring the full query data from the external system. In this case, the UCMDB Probe Framework is responsible for determining the difference between two consecutive runs. The Probe Framework achieves this by comparing the previous result for the given query with the current result, and computing the differences. Full population is achieved by not comparing the current query result and treating it as the final result. This flow implies that the populated data is not filtered by a “from date”, because filtering by a date would render data comparison meaningless.
- changes-source Population Adapter
  - Is configured by the use of the **<changes-source/>** tag in the adapter XML file:

```
<adapterInfo>
    <adapter-capabilities>
        <support-federated-query/>
        <support-replicatioin-data>
            <source>
                <changes-source/>
                <push-back-ids/>
                <re-populate/>
            </source>
            <target/>
        </support-replicatioin-data>
```

- The changes-source adapter is responsible for computing the difference between two consecutive runs.

Deleting CIs when using a changes-source adapter

If you are using a changes-source population adapter, your adapter is responsible for explicitly deleting CIs. This is done by using the **is-deleted** mapping file XML attribute, which accepts a valid Groovy expression.

For example, in the mapping file shown below, the population connector returns **Service** instances. Although the instances are still valid, some CIs that were part of those **Service** instances were deleted. To signal the delete of those CIs, you need to use the **is-deleted** attribute on the **BusinessService** mapping.

```
<target_entities>
    <source_instance query-name="Nodes And BS Population" root-element-name="Service">
        <!-- Node -->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="Service.PC['name']"/>
        </target_entity>

        <!-- Business Service -->
        <target_entity name="BusinessService" is-deleted="Functions.isOlderThanThreeMonths()">
            <target_mapping name="name" datatype="STRING" value="Service['name']"/>
            <target_mapping name="description" datatype="STRING" value="Service['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

Explicit External ID Mapping

There may be situations where the populated data (CIs) will need to have a connector/adapter controlled **ExternalId**. Use the following mapping construct to do this:

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Node with ID" root-element-name="Computer">
        <!-- need to match case in UCMDB TQL -->
        <target_entity name="Root">
            <!--This is how the RTN External ID is set-->
            <variable name="external_id_obj" datatype="STRING" value="Computer['external_id_obj']"/>
            <!--RTN Attributes-->
            <target_mapping name="name" datatype="STRING" value="Computer.Asset[0]['name']"/>
            <target_mapping name="description" datatype="STRING" value="Computer['name']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

In this case, the Root CI is populated with an ExternalId that was created at the connector level and placed on the Computer['external_id_obj']. The creation of the ExternalId can also be done at the mapping level using a Groovy script.

Note: The mechanism of explicitly creating an external ID overrides the `target_entity` **type** attribute. Thus, when creating an external ID either with the mapping script file or inside the connector, the **type** attribute is ignored, and the final UCMDB type of the populated CI will be the UCMDB type set in the **ExternalId** object.

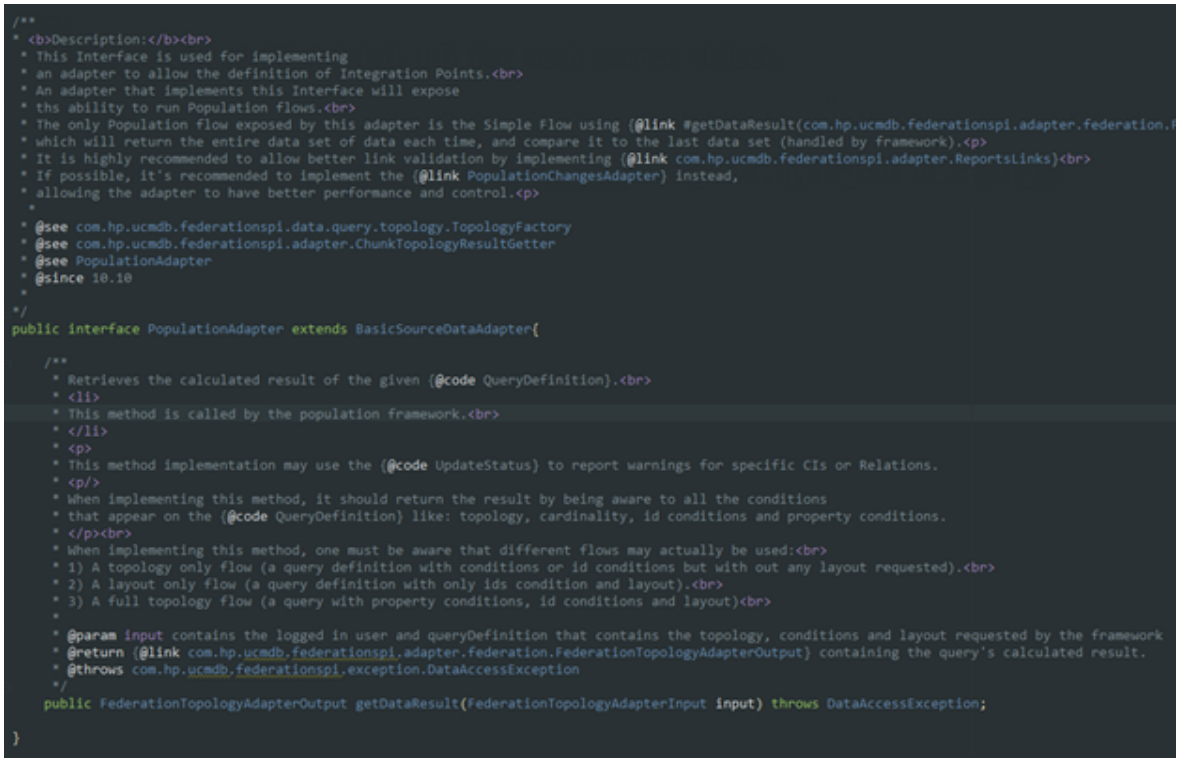
Global ID Pushback

There are situations where populated CIs in UCMDB need to be kept in sync in the external system as well. For this scenario, the Generic Adapter framework allows the enabling of pushback IDs. To use this feature, a callback is performed for all the CIs that were populated in UCMDB, informing the Population Adapter about the assigned global ID for each CI.

To enable this functionality, add the line that is marked in the example below to the adapter configuration XML file:

```
<adapterInfo>
    <adapter-capabilities>

        <support-federated-query>
        <supported-classes>
            <supported-class is-derived="true" all-attributes-supported="true" name="node" is-reconciliation-supported="true"/>
            <supported-class is-derived="true" all-attributes-supported="true" name="business_service" is-reconciliation-supported="true"/>
            <supported-class is-derived="true" all-attributes-supported="true" name="incident" is-reconciliation-supported="true"/>
        </supported-classes>
        </support-federated-query>
        <support-replicatioin-data>
            <source>
                <push-back-ids/>
                <instance-based-data/>
                <population-queries-resources-locator/>
            </source>
            <target>
                <instance-based-data>true</instance-based-data>
                <push-queries-resources-locator/>
            </target>
        </support-replicatioin-data>
        <general-resources-locator/>
    </adapter-capabilities>
```

You must also implement the `PopulationAdapterConnector` method as follows:



```
/**
 * <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose
 * ths ability to run Population flows.<br>
 * The only Population flow exposed by this adapter is the Simple Flow using {@link #getDataResult(com.hp.ucmdb.federationspi.adapter.federation.
 * which will return the entire data set of data each time, and compare it to the last data set (handled by framework).<p>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}<br>
 * If possible, it's recommended to implement the {@link PopulationChangesAdapter} instead,
 * allowing the adapter to have better performance and control.<p>
 *
 * @see com.hp.ucmdb.federationspi.data.query.topology.TopologyFactory
 * @see com.hp.ucmdb.federationspi.adapter.ChunkTopologyResultGetter
 * @see PopulationAdapter
 * @since 10.10
 *
 */
public interface PopulationAdapter extends BasicSourceDataAdapter{

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the population framework.<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by the framework
     * @return {@link com.hp.ucmdb.federationspi.adapter.federation.FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

Achieving Data Federation using the Generic Adapter

Data federation is achieved by using the following:

Federation Mapping Approach

Federation mapping is achieved by mapping the sub-TQLs used by the UCMDB Federation framework to process a federation request. The general idea is that when a federation request is received by the Generic Adapter, what happens is this:

1. Analyze the dynamic federation TQL query and compare it with a list of static federation TQL queries provided by the federation connector.

2. A static TQL query match is made. This TQL query is used to identify the needed mapping for the given federation request and to create the RTN Structure (a Java object that will illustrate the tree node structure that is needed from the connector) input argument that will be supplied to the federation connector. (For more information, see the **push-interfaces.jar** file)

3. Send the federation call with the TQL argument to the connector.

4. Map the incoming RTN trees sent by the connector in the same way as for population. See "Achieving Data Population using the Generic Adapter" on page 324.

Federation Link Mapping

Federation Link Mapping is performed automatically as is the case for population link mapping. See "Automatic Link Population" on page 329.

Generic Adapter Federation API

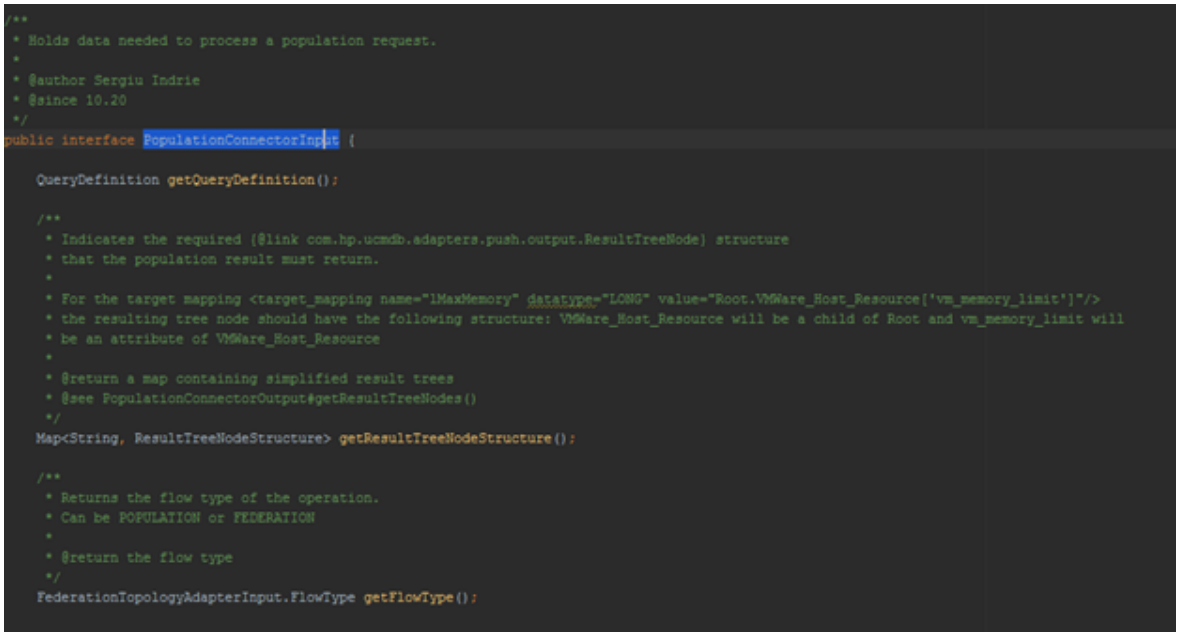
The Generic Adapter Federation API is very similar to the Generic Adapter Population API. This is because the Generic Federation Adapter Java interface is identical to the Generic Population Adapter interface.

```
/**
 * <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose
 * ths ability to run Population flows.<br>
 * The only Population flow exposed by this adapter is the Simple Flow using {@link #getDataResult(com.hp.ucmdb.federationspi.adapter.federation.
 * which will return the entire data set of data each time, and compare it to the last data set (handled by framework).<p>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}<br>
 * If possible, it's recommended to implement the {@link PopulationChangesAdapter} instead,
 * allowing the adapter to have better performance and control.<p>
 *
 * @see com.hp.ucmdb.federationspi.data.query.topology.TopologyFactory
 * @see com.hp.ucmdb.federationspi.adapter.ChunkTopologyResultGetter
 * @see PopulationAdapter
 * @since 10.10
 *
 */
public interface PopulationAdapter extends BasicSourceDataAdapter{

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the population framework.<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by the framework
     * @return {@link com.hp.ucmdb.federationspi.adapter.federation.FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

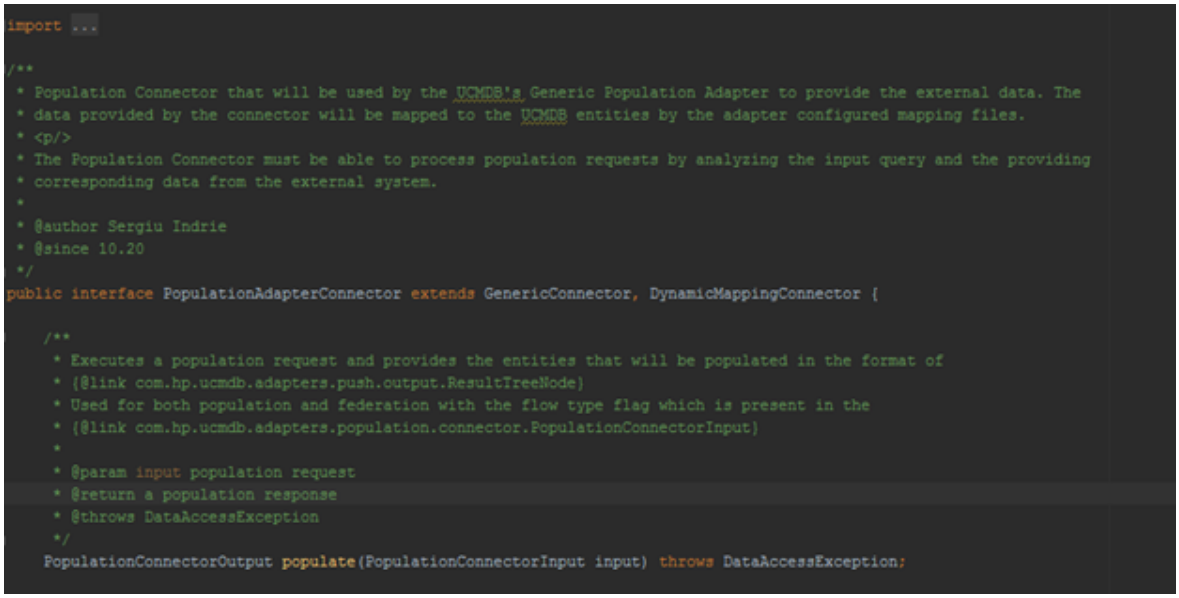
```
import ...

/**
 * <b>Description:</b><br>
 * This Interface is used for implementing
 * an adapter to allow the definition of Integration Points.<br>
 * An adapter that implements this Interface will expose to run Federation flows.<br>
 * The adapter can report status per CI during the flow with {@link com.hp.ucmdb.federationspi.data.replication.UpdateStatus}.<br>
 * It is highly recommended to allow better link validation by implementing {@link com.hp.ucmdb.federationspi.adapter.ReportsLinks}
 * </p>
 *
 */
public interface FederationTopologyDataAdapter extends FTqlDataAdapter {

    /**
     * Retrieves the calculated result of the given {@code QueryDefinition}.<br>
     * <li>
     * This method is called by the federation framework when the user query includes any
     * federated elements(node or link).<br>
     * </li>
     * <p>
     * This method implementation may use the {@code UpdateStatus} to report warnings for specific CIs or Relations.
     * <p/>
     * When implementing this method, it should return the result by being aware to all the conditions
     * that appear on the {@code QueryDefinition} like: topology, cardinality, id conditions and property conditions.
     * </p><br>
     * When implementing this method, one must be aware that different flows may actually be used:<br>
     * 1) A topology only flow (a query definition with conditions or id conditions but with out any layout requested).<br>
     * 2) A layout only flow (a query definition with only ids condition and layout).<br>
     * 3) A full topology flow (a query with property conditions, id conditions and layout)<br>
     *
     * @param input contains the logged in user and queryDefinition that contains the topology, conditions and layout requested by th
     * @return {@link FederationTopologyAdapterOutput} containing the query's calculated result.
     * @throws com.hp.ucmdb.federationspi.exception.DataAccessException
     */
    public FederationTopologyAdapterOutput getDataResult(FederationTopologyAdapterInput input) throws DataAccessException;

}
```

Generic Adapter Connector Interface for Federation

Federation requests use the same method that is used for population requests, so the same population connector implementation can be used. A new attribute was added in the PopulationConnectorInput Java class called **FlowType**. The **FlowType** attribute can have two values, FEDERATION or POPULATION. The Generic Adapter knows the request type based on this attribute.



```
/**
 * Holds data needed to process a population request.
 *
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationConnectorInput {

    QueryDefinition getQueryDefinition();

    /**
     * Indicates the required {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode} structure
     * that the population result must return.
     *
     * For the target mapping <target_mapping name="lMaxMemory" datatype="LONG" value="Root.VMWare_Host_Resource['vm_memory_limit']"/>
     * the resulting tree node should have the following structure: VMWare_Host_Resource will be a child of Root and vm_memory_limit will
     * be an attribute of VMWare_Host_Resource
     *
     * @return a map containing simplified result trees
     * @see PopulationConnectorOutput#getResultTreeNodes()
     */
    Map<String, ResultTreeNodeStructure> getResultTreeNodeStructure();

    /**
     * Returns the flow type of the operation.
     * Can be POPULATION or FEDERATION
     *
     * @return the flow type
     */
    FederationTopologyAdapterInput.FlowType getFlowType();
```



```
import ...

/**
 * Population Connector that will be used by the UCMDB's Generic Population Adapter to provide the external data. The
 * data provided by the connector will be mapped to the UCMDB entities by the adapter configured mapping files.
 * <p/>
 * The Population Connector must be able to process population requests by analyzing the input query and the providing
 * corresponding data from the external system.
 *
 * @author Sergiu Indrie
 * @since 10.20
 */
public interface PopulationAdapterConnector extends GenericConnector, DynamicMappingConnector {

    /**
     * Executes a population request and provides the entities that will be populated in the format of
     * {@link com.hp.ucmdb.adapters.push.output.ResultTreeNode}
     * Used for both population and federation with the flow type flag which is present in the
     * {@link com.hp.ucmdb.adapters.population.connector.PopulationConnectorInput}
     *
     * @param input population request
     * @return a population response
     * @throws DataAccessException
     */
    PopulationConnectorOutput populate(PopulationConnectorInput input) throws DataAccessException;
```

Supported Federation Queries

The federation and population queries are located in different folders. The **PopulationAdapterConnector** Java interface offers the following two methods for indicating the supported population and federation queries:

- **getPopulationQueries** – Returns the collection of queries that the current connector supports for population.
- **getFederationQueries** – Returns the collection of queries that the current connector supports for federation.

```
/**
 * Returns the collection of queries the current connector supports for population
 * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
 * <p/>
 * The method also supports return of queries with their folders path: E.g.
 * "Folder/Secondary Folder/Query Name 1"
 * "Folder/Secondary Folder/Query Name 2"
 *
 * @param env the adapter environment
 * @return a collection of the supported queries
 */
Collection<String> getPopulationQueries(DataAdapterEnvironment env);

/**
 * Returns the collection of queries the current connector supports for federation
 * Note: this method is independent to the adapter life cycle (i.e. start/shutdown methods) and must work at all times.
 * <p/>
 * The method also supports return of queries with their folders path: E.g.
 * "Folder/Secondary Folder/Query Name 1"
 * "Folder/Secondary Folder/Query Name 2"
 *
 * @param env the adapter environment
 * @return a collection of the supported queries
 */
Collection<String> getFederationQueries(DataAdapterEnvironment env);
```

How to Set Up Federation

This section contains:

Configure the Adapter Settings

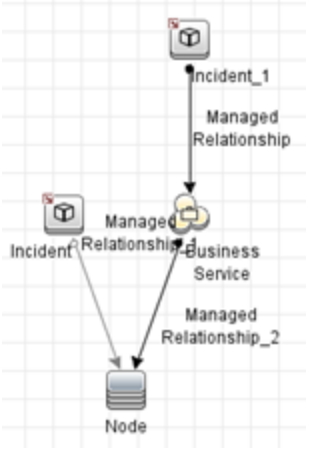
For a given a TQL query, the Generic Adapter needs to declare all the nodes from that TQL query in the **<supported-classes>** tag . For example, if the TQL query has the form of an Incident linked to a Node, then you must declare both the node and the incident as supported classes in the adapter settings xml file located in the adapter package ZIP file in the **discoveryPatterns** folder.


```
<supported-classes>
    <supported-class is-derived="true" all-attributes-supported="true" name="node" is-reconciliation-supported="true"/>
    <supported-class is-derived="true" all-attributes-supported="true" name="incident" is-reconciliation-supported="true"/>
</supported-classes>
```

Set Up Federation Queries from Log Files

Some prerequisites must be fulfilled before using the federation framework . The federation framework makes several requests with different TQL queries to the adapter in order to retrieve the required data. For each TQL query that the federation framework sends, a dynamic TQL query must be created in the adapter, in a similar way to the population flow.

The difference between federation and population is that the federation TQL queries are sent dynamically by the framework so they cannot be known ahead of time. Take the following TQL query as an example:



The framework sends the following queries to the adapter:

- A query with only the incident
- A query with the incident linked to the node with a relation of type connection
- A query with the incident linked to the node with a relation of type membership
- A query with the incident linked to the business service with a relation of type connection
- A query with the incident linked to the business service with a relation of type membership

Note: All queries that the federation engine sends to the adapter and that need to be saved have the name **User mapping union FTQL**.

After the results of the **User mapping union FTQL** query are processed, other calls are made to retrieve the attributes of the objects. These calls contain a query called **objects layout**. The federation engine will try to get all the attributes for a CI, but the connector does not need to provide them all; it is enough to return only the ones required by the mapping file.

The reason for sending the same query with different relations is because in the TQL query there is a **managed_relationship** type link between the node and incident/business service and incident, but the only valid links when trying to link these CI types together are connection and membership.

```
<tql:link from="incident_12" to="node_10050" class="connection" name="connection_1" id="1"/>
<tql:link from="incident_12" to="node_1000050" class="connection" name="connection_2" id="2"/>
<tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
```

```
<tql:link from="incident_12" to="node_10050" class="membership" name="membership_1" id="1"/>
<tql:link from="incident_12" to="node_1000050" class="membership" name="membership_2" id="2"/>
<tql:link from="datacenter_20050" to="node_10050" class="composition" name="composition_30050" id="30050"/>
```

By using this approach we only need to define one static TQL with a generic **managed_relationship** type link, instead of defining two almost identical TQLs with different link types.

```
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >> Received federation call with the following query:
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tql:query name="User mapping union FTQL" is-live="true" priority="low" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleD
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="membership_1"/>
                    <tql:link-ref name="membership_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:link from="incident_16" to="business_service_10050" class="membership" name="membership_1" id="1"/>
    <tql:link from="incident_16" to="business_service_1000050" class="membership" name="membership_2" id="2"/>
</tql:query>
```

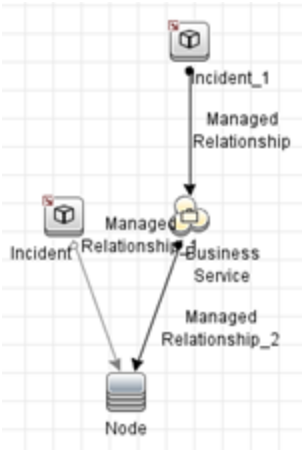
These static TQL queries must be provided by the adapter. To help adapter development, the TQL queries sent by the federation framework are written in the **fcmdb.push.mapping.log** file (with TRACE log level enabled). To ease the development effort, use the setting `<adapter-setting name="dev.mode">true</adapter-setting>`. If this setting is set to True the after running a federation query, the framework will automatically create an empty federation result for the current unmatched TQL query.

Example of a federation query from fcmdb.push.mapping.log:

```
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >> Received federation call with the following query:
2014-08-07 16:43:55,231 [AdHoc:AD_HOC_TASK_PATTERN_ID-179-1407419035224] TRACE -  >>
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tql:query name="User mapping union FTQL" is-live="true" priority="low" xmlns:ns4="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns3="http://www.hp.com/ucmdb/1-0-0/PolicyRuleD
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="membership_1"/>
                    <tql:link-ref name="membership_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>My Bus Serv</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:link from="incident_16" to="business_service_10050" class="membership" name="membership_1" id="1"/>
    <tql:link from="incident_16" to="business_service_1000050" class="membership" name="membership_2" id="2"/>
</tql:query>
```

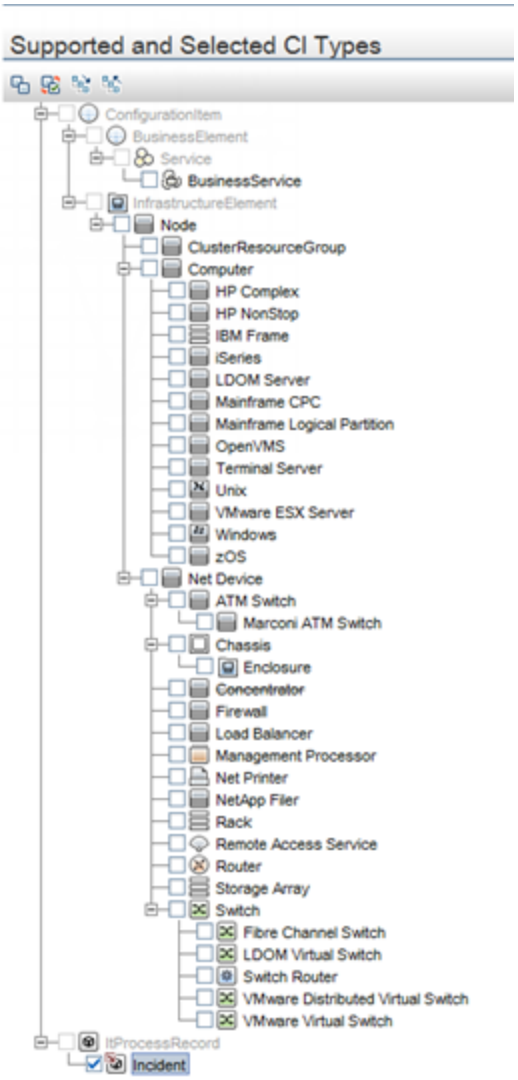
Federation Setup Example

The example will use the following federation TQL:



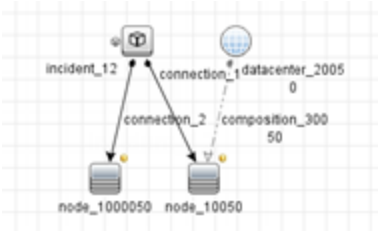
For this TQL query, the adapter must declare the supported classes in the adapter settings XML file located in the adapter package ZIP file in the **discoveryPatterns** folder. The supported classes are **node**, **incident**, and **business_service**.

In the Integration Studio, the incident must be selected on the **Federation** tab, as shown below:

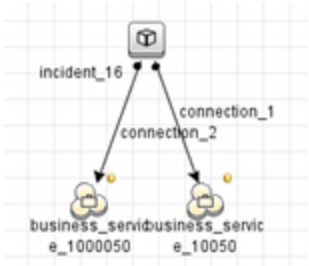


For this TQL query, we must have the following three TQL queries in the adapter:

Incident



Incident linked to Node



Incident linked to business_service

For information about how to obtain static TQLs, see "Set Up Federation Queries from Log Files" on page 345. Although these TQL queriess have conditions dependent on the data present in UCMDB, this should not affect the structure of the TQL queries or how the mapping is performed.

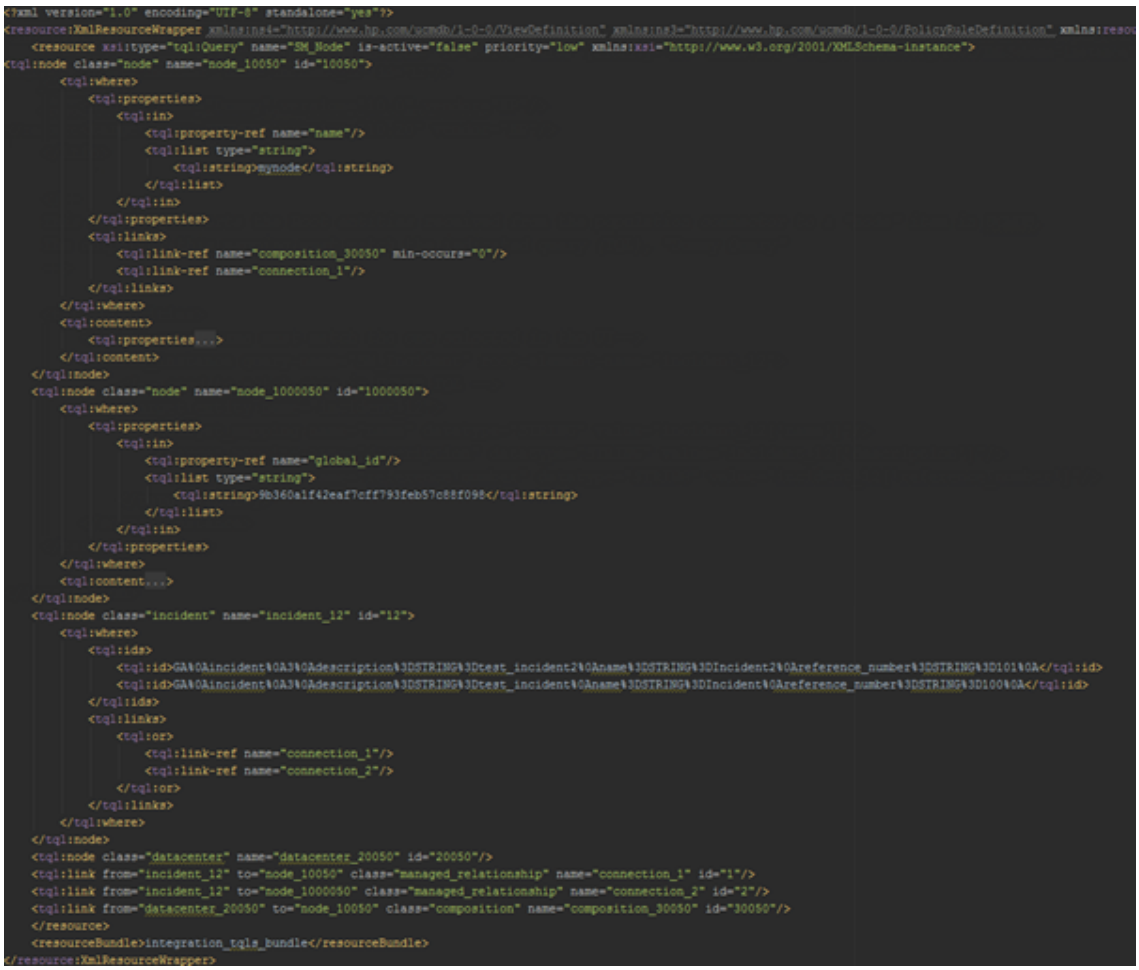
For each of these TQL queries, a mapping file is required in the adapter:

- Incident

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns1="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns2="http://www.hp.com/ucmdb/1-0-0/PolicyRuleDefiniti
    <resource xsi:type="tql:Query" name="SM_Incident" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<tql:node class="incident" name="incident_12" id="12"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

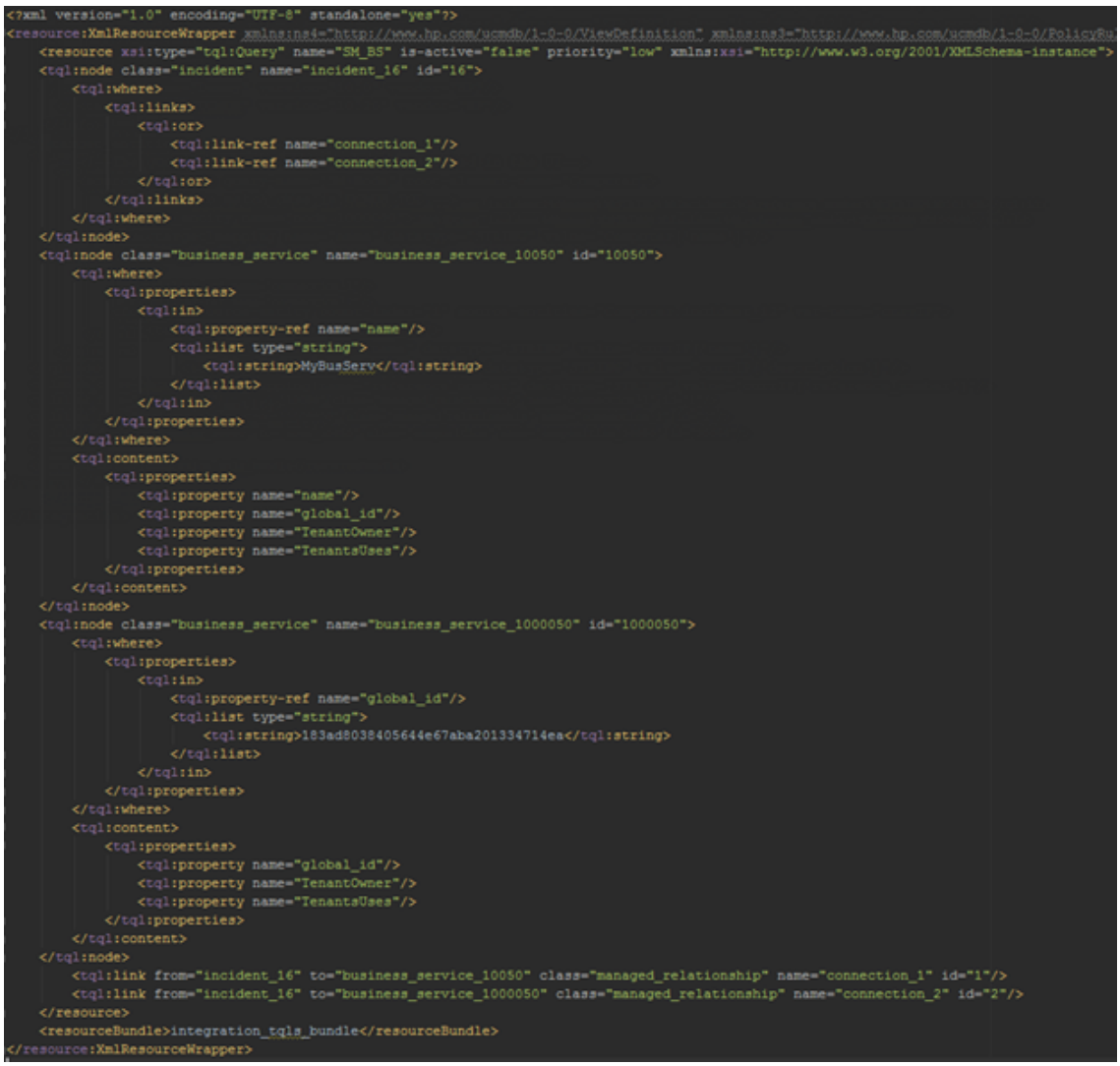
```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>

    <!--
    This mapping converts the Root entities received from the population connector to a "node" item in UCMDB.
    The output of this mapping must match the indicated query (TQL), "Dummy Query"
    -->

    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_Incident" root-element-name="incident_12">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="incident_12">
                <target_mapping name="name" datatype="STRING" value="incident_12['name']"/>
                <target_mapping name="description" datatype="STRING" value="incident_12['description']"/>
                <target_mapping name="reference_number" datatype="STRING" value="incident_12['reference_number']"/>
            </target_entity>
        </source_instance>
    </target_entities>

</integration>
```

- Incident linked to Node



```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>
    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_Node" root-element-name="Computer">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="node_1000050">
                <target_mapping name="name" datatype="STRING" value="Computer['name']"/>
            </target_entity>


            <for-each-source-entity count-index="i" source-entities="Computer.incident_12" var-name="currIP">
                <target_entity name="incident_12">
                    <target_mapping name="name" datatype="STRING" value="currIP['name']"/>
                    <target_mapping name="description" datatype="STRING" value="currIP['description']"/>
                    <target_mapping name="reference_number" datatype="STRING" value="currIP['reference_number']"/>
                </target_entity>
            </for-each-source-entity>


        </source_instance>
    </target_entities>
</integration>
```

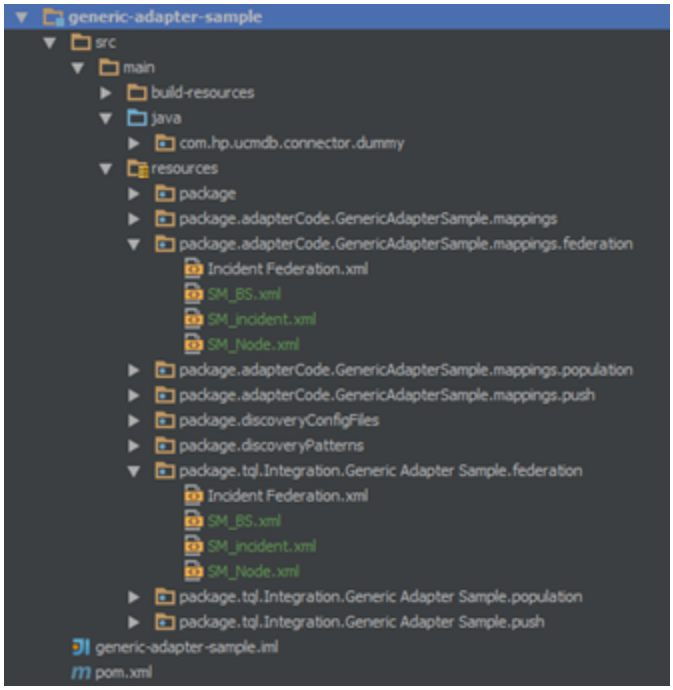
- Incident to business_service



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<resource:XmlResourceWrapper xmlns:ns1="http://www.hp.com/ucmdb/1-0-0/ViewDefinition" xmlns:ns2="http://www.hp.com/ucmdb/1-0-0/PolicyDe
    <resource xsi:type="tql:Query" name="SM_BS" is-active="false" priority="low" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <tql:node class="incident" name="incident_16" id="16">
        <tql:where>
            <tql:links>
                <tql:or>
                    <tql:link-ref name="connection_1"/>
                    <tql:link-ref name="connection_2"/>
                </tql:or>
            </tql:links>
        </tql:where>
    </tql:node>
    <tql:node class="business_service" name="business_service_10050" id="10050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="name"/>
                    <tql:list type="string">
                        <tql:string>MyBusServ</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="name"/>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
    <tql:node class="business_service" name="business_service_1000050" id="1000050">
        <tql:where>
            <tql:properties>
                <tql:in>
                    <tql:property-ref name="global_id"/>
                    <tql:list type="string">
                        <tql:string>183ad8038405644e67aba201334714ea</tql:string>
                    </tql:list>
                </tql:in>
            </tql:properties>
        </tql:where>
        <tql:content>
            <tql:properties>
                <tql:property name="global_id"/>
                <tql:property name="TenantOwner"/>
                <tql:property name="TenantsUses"/>
            </tql:properties>
        </tql:content>
    </tql:node>
        <tql:link from="incident_16" to="business_service_10050" class="managed_relationship" name="connection_1" id="1"/>
        <tql:link from="incident_16" to="business_service_1000050" class="managed_relationship" name="connection_2" id="2"/>
    </resource>
    <resourceBundle>integration_tqls_bundle</resourceBundle>
</resource:XmlResourceWrapper>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<integration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../generic-adapter.xsd">
    <!-- add scheme reference -->
    <info>
        <source name="Dummy" version="10.0" vendor="HP"/>
        <target name="UCMDB" version="10.20" vendor="HP"/>
    </info>
    <target_entities>
        <!--The query name must match the one selected in the UI-->
        <source_instance query-name="SM_BS" root-element-name="BS">
            <!-- need to match case in UCMDB TQL -->
            <target_entity name="business_service_10050">
                <target_mapping name="name" datatype="STRING" value="BS['name']"/>
            </target_entity>
            <for-each-source-entity count-index="i" source-entities="BS.incident_16" var-name="currIP">
                <target_entity name="incident_16">
                    <target_mapping name="name" datatype="STRING" value="currIP['name']"/>
                    <target_mapping name="description" datatype="STRING" value="currIP['description']"/>
                    <target_mapping name="reference_number" datatype="STRING" value="currIP['reference_number']"/>
                </target_entity>
            </for-each-source-entity>
        </source_instance>
    </target_entities>
</integration>
```

These TQL queries and mapping files must be present in the adapter, as shown below:



Reconciliation

When using the Generic Adapter framework to populate or federate data, the CIs must always have the required reconciliation data in order to be accepted into UCMDB. When populating CI types such as Running Software that require a container CI type, always make sure to populate the needed container fields (for example, **root_container_name** and **product_name**) and the container CI (for example, **Node**). To populate CIs that depend on a root container, the CI, its root container, and the link between them

must be created in the same step (either with explicit link population or auto-complete link population between the two CIs).

In addition, when mapping the populated/federated CIs, consider mapping the **global_id** attribute, as that will greatly aid the UCMDB reconciliation engine and should guarantee the exact CI reconciliation.

Generic Adapter API

The API exposed by the Generic Adapter framework is:

```
<UCMDB_Server>\lib\push-interfaces.jar
<UCMDB_Server>\lib\integrationFramework\GenericAdapter\generic-adapter-api-factory.jar
```

The development of your Generic Adapter instance might also require the Federation API:

```
<UCMDB_Server>\lib\federation-api.jar
```

Resource Locator APIs

The resource locator APIs can be used when editing Generic Adapter jobs. Implement the general and population resource locator APIs to help find the adapter resources that are related to a selected job's TQL query.

The following image shows General Resource Locator API in the GenericConnector Java Interface:

```
/**
 * This methods reports general resources used by the adapter.<br>
 * This allows editing these resources directly from the Integration Studio.
 *
 *
 * @param env the Adapter's environment
 * @param input  the requested information
 * @param output the returned resources
 * @see com.hp.ucmdb.federationspi.adapter.resource.GeneralResourcesLocator
 */
void locateGeneralResources(DataAdapterEnvironment env, LocateGeneralResourcesInput input, LocateGeneralResourcesOutput output);
```

The following image shows Population Query Resource Locator API in PopulationAdapterConnector Java Interface:

```
/**
 * This methods reports population queries resources used by the adapter.<br>
 * This allows editing these resources directly from the Integration Studio.
 *
 * @param input  the requested information
 * @param output the returned resources
 * @see com.hp.ucmdb.federationspi.adapter.resource.PushQueriesResourceLocator
 */
void locatePopulationQueriesResources(DataAdapterEnvironment env, LocatePopulationQueriesResourcesInput input,
                                      LocatePopulationQueriesResourcesOutput output);
```

To see the related resources for a job's TQL query:

1. In the Integration Studio, select an integration point.
2. In the Integration Jobs pane, select a job and click **Edit Query Resources**  .

Create a Generic Adapter Package

A Generic Adapter package is similar to an Enhanced Generic Push Adapter package. To create the initial skeleton ZIP archive, it is recommended to copy an existing Generic Adapter package and customize it as required. For more information about the adapter package, see "Achieving Data Push using the Generic Adapter" on page 315.

The differences between an existing Enhanced Generic Push Adapter package and a Generic Adapter package are:

- Adapter XML differences
  - The adapter class is changed from **PushAdapter** to **GenericAdapter**:

```
<className>com.hp.ucmdb.adapters.push.PushAdapter</className>
```

```
<className>com.hp.ucmdb.adapters.GenericAdapter</className>
```

  - The adapter capabilities include population

```
<support-replicatioin-data>
    <source>
        <push-back-ids/>
        <instance-based-data/>
        <population-queries-resources-locator/>
    </source>
```

  - Along with the definition of the Population connector, performed by the adapter setting:

```
<adapter-setting name="PopulationConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPopulationConnector</adapter-setting>
```

    the Generic Adapter (using the population feature) also requires the definition of the Push Connector class:

```
<adapter-setting name="PushConnector.class.name">com.hp.ucmdb.connector.dummy.DummyPushConnector</adapter-setting>
```

- Mapping file folders

  As opposed to the Enhanced Generic Push Adapter (which requires its mapping files to be located in the **<adapter_package_zip>/adapterCode/<*adapter_name*>/mappings** folder, the Generic Adapter requires its mappings to be placed in three separate folders (one each for push, population, and federation). The required folders are:

  <adapter_package_zip>/adapterCode/<*adapter_name*>/mappings/push
  <adapter_package_zip>/adapterCode/<*adapter_name*>/mappings/population
  <adapter_package_zip>/adapterCode/<*adapter_name*>/mappings/federation

  where **<adapter_package_zip>** refers to the zip archive that you will create for the generic adapter package.

Note: Although the Generic Adapter supports all three types of data synchronization (push, population, and federation), a specific Generic Adapter can choose to supply only a subset of those types.

Points to remember when creating a new adapter from an existing adapter

- **TestAdapter\discoveryPatterns\TestAdapter.xml**
- Modify the **TestAdapter.xml** file:
  - `<pattern xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" id="TestAdapter" xsi:noNamespaceSchemaLocation="../../Patterns.xsd" description="…" schemaVersion="9.0" displayName="…">`
  - `<adapter-id>TestAdapter</adapter-id>`
- The ZIP file containing the new adapter should have the same name as the adapter itself - TestAdapter.

Build an Adapter Package

Ensure that the adapter package contains the following folders:

- **adapterCode.** Under this folder, create a folder named **PushExampleAdapter**, which will contain the .jar file we created from the PushExampleAdapter.java. It will also contain a folder named **mappings**, where you can place the mapping file created earlier, **computerIPMapping.xml**. It should also

contain another folder named **scripts** that contains the **PushFunctions.groovy** file.

- **discoveryConfigFiles.** To contain configuration files such as the error codes used when reporting an error using UpdateResult. In this example, the folder is empty.
- **discoveryPatterns.** To contain the **push_example_adapter.xml**.
- **tql.** To contain the TQL query created for the example. This folder is optional, but when the package is deployed, the TQL query is automatically created.

Enable/Disable Attribute and Link Validation at Adapter Level

You can enable or disable attribute and link validation at adapter level for generic adapters by adding the following setting:

```
<adapter-settings>

        <adapter-setting
name="enable.attributes.links.validation">true</adapter-setting>

</adapter-settings>
```

To enable adapter level validation of attribute and link, set the adapter setting **enable.attributes.links.validation** to **true**.

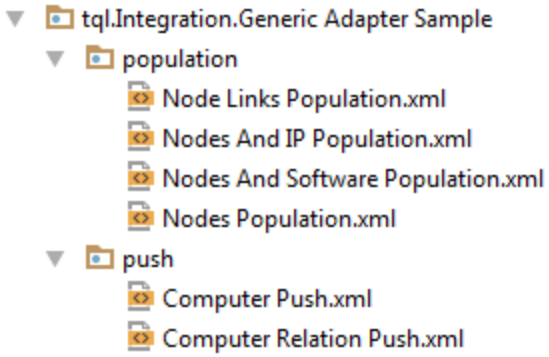
To disable adapter level validation of attribute and link, set the adapter setting **enable.attributes.links.validation** to **false**.

Note: If the setting is not present, its default value is **true**, which means that by default the attribute and link validation is enabled.

Population TQL Queries

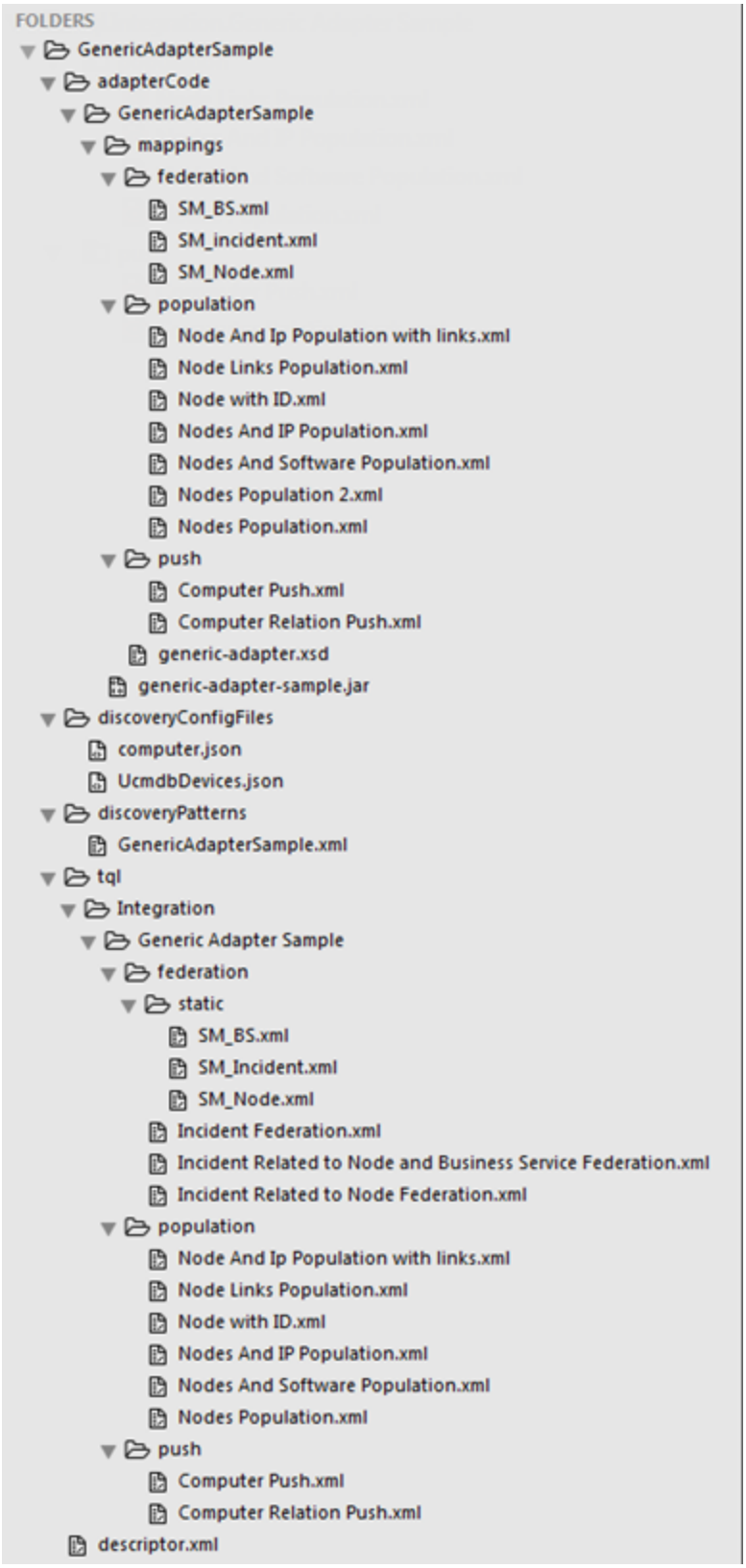
The TQL queries to be used for population jobs must be included in the Generic Adapter's ZIP archive and deployed with the adapter in UCMDB. The indicated TQL query must exist in UCMDB when a population request is made, during the population flow.

These TQL queries must be included in the **<zip>/tql/<folder_1>/../<folder_n>**. Following is an example of the folder structure:



Although population TQL queries are placed in the folder indicated above, the Population connector must also confirm the supported population TQL queries in the corresponding method from the Java interface. For more information, see "The Population Connector" on page 331.

Sample Package


FOLDERS
GenericAdapterSample
adapterCode
GenericAdapterSample
mappings
federation
SM_BS.xml
SM_incident.xml
SM_Node.xml
population
Node And Ip Population with links.xml
Node Links Population.xml
Node with ID.xml
Nodes And IP Population.xml
Nodes And Software Population.xml
Nodes Population 2.xml
Nodes Population.xml
push
Computer Push.xml
Computer Relation Push.xml
generic-adapter.xsd
generic-adapter-sample.jar
discoveryConfigFiles
computer.json
UcmdbDevices.json
discoveryPatterns
GenericAdapterSample.xml
tql
Integration
Generic Adapter Sample
federation
static
SM_BS.xml
SM_Incident.xml
SM_Node.xml
Incident Federation.xml
Incident Related to Node and Business Service Federation.xml
Incident Related to Node Federation.xml
population
Node And Ip Population with links.xml
Node Links Population.xml
Node with ID.xml
Nodes And IP Population.xml
Nodes And Software Population.xml
Nodes Population.xml
push
Computer Push.xml
Computer Relation Push.xml
descriptor.xml

Differences Between Push and Population Mapping

Although both push and population mapping files have the same underlying XML schema, the files have slightly different interpretations. For more information, see "Generic Adapter XML Schema Reference" on page 365.

In the following push mapping example, the interpretation is: take the results of the “Computer Push” TQL query (run in UCMDB) and present in the Root tree structure, and create the amComputer entity which will later on be sent to AM.

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Computer Push" root-element-name="Root">
        <target_entity name="amComputer">
            <target_mapping name="TcpIpHostName" datatype="STRING" value="Root['name']"/>
            <target_mapping name="ComputerDesc" datatype="STRING" value="Root['os_description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

In the following population mapping example, the interpretation is: take the results of the “Nodes Population” TQL query (run in the external system) and present in the PC tree structure, and create the UCMDB Root entity (of type Node; as indicated by the TQL query), which will later be added in UCMDB.

```
<target_entities>
    <!--The query name must match the one selected in the UI-->
    <source_instance query-name="Nodes Population" root-element-name="PC">
        <!-- need to match case in UCMDB class model-->
        <target_entity name="Root">
            <target_mapping name="name" datatype="STRING" value="PC['name']"/>
            <target_mapping name="description" datatype="STRING" value="PC['description']"/>
        </target_entity>
    </source_instance>
</target_entities>
```

Generic Adapter Log Files

For troubleshooting and debugging, use the following:

- Adjust logging levels in these files (set the *loglevel* variable to TRACE for the most detailed results):
  - **<UCMDB_DataFlowProbe>\conf\log\fcmdb.push.properties**

    <UCMDB_DataFlowProbe> is the UCMDB Data Flow Probe installation directory.
  - **<UCMDB_Server>\conf\log\reconciliation.properties**

    <UCMDB_Server> is the UCMDB Server installation directory.
- Analyze the following Generic Adapter log files:
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.all.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.configuration.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.connector.all.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.connector.configuration.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.mapping.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\fcmdb.push.all.log**
- Analyze the following generic log files:
  - **<UCMDB_DataFlowProbe>\runtime\log\probe-error.log**
  - **<UCMDB_DataFlowProbe>\runtime\log\WrapperProbeGw.log**
  - **<UCMDB_Server>\runtime\log\error.log**
  - **<UCMDB_Server>\runtime\log\cmdb.reconciliation.log**

Adapters Using the Generic Adapter Framework

As a reference for developing your custom Generic Adapter, refer to the following adapters that shipped with UCMDB as implementation guidelines, which should speed up your adapter development:

- The Asset Manager Adapter
- The Service Manager Adapter

Generic Adapter XML Schema Reference

The Generic Adapter XML Schema can be found in the **cmdb.jar** file under the **schema** directory. The schema file should be referenced while writing Generic Adapter mapping files in external editors. The full path for the XSD file is:

<UCMDB_Server_Install_dir>/lib/cmdb.jar/schema/generic-adapter.xsd

Part II: Using APIs

Chapter 9: Introduction to APIs

This chapter includes:

APIs Overview

The following APIs are included with HP Universal CMDB:

- **UCMDB Java API.** Explains how third-party or custom tools can use the Java API to extract data and calculations and to write data to the UCMDB (Universal Configuration Management database). For details, see "HP Universal CMDB API" on page 368.
- **UCMDB Web Service API.** Enables writing configuration item definitions and topological relations to UCMDB, and querying the information with TQL and ad hoc queries. For details, see "HP Universal CMDB Web Service API" on page 378.
- **Data Flow Management Java API.** Enables managing probes, jobs, triggers and credentials for Data Flow Management. For details, see "Data Flow Management Java API" on page 418.
- **Data Flow Management Web Service API.** Enables managing probes, jobs, triggers and credentials for Data Flow Management. For details, see "Data Flow Management Web Service API" on page 420.

Note: To gain the full value of the API documentation, it is recommended to access the online documentation. The PDF version does not have the links into the API documentation that is generated in html format.

Chapter 10: HP Universal CMDB API

This chapter includes:

Conventions

This chapter uses the following conventions:

- UCMDB refers to the Universal Configuration Management database itself. HP Universal CMDB refers to the application.
- UCMDB elements and method arguments are spelled in the case in which they are specified in the interfaces.

For full documentation on the available APIs, refer to the HP UCMDB API Reference.

These files are located in the following folder:

\\<UCMDB root directory>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\UCMDB_JavaAPI\index.html

Using the HP Universal CMDB API

Note: Use this chapter in conjunction with the API Javadoc, available in the online Documentation Library.

The HP Universal CMDB API is used to integrate applications with the Universal CMDB (CMDB). The API provides methods to:

- Add, remove, and update CIs and relations in the CMDB
- Retrieve information about the class model
- Retrieve information from the UCMDB history
- Run what-if scenarios
- Retrieve information about configuration items and relationships

Methods for retrieving information about configuration items and relationships generally use the Topology Query Language (TQL). For details, see Topology Query Language in the *HP Universal CMDB Modeling Guide*.

Users of the HP Universal CMDB API should be familiar with:

- The Java programming language
- HP Universal CMDB

This section includes the following topics:

Uses of the API

The API is used to fulfill a number of business requirements. For example, a third-party system can query the class model for information about available configuration items (CIs). For more use cases, see "UCMDB API Use Cases" on page 375.

Permissions

The administrator provides login credentials for connecting with the API. The API client needs the user name and password of an integration user defined in the CMDB. These users do not represent human users of the CMDB, but rather applications that connect to the CMDB.

In addition, the user must have the **Access to SDK** general action permission in order to log in.

Caution: The API client can also work with regular users as long as they have API authentication

permission. However, this option is not recommended.

For details, see "Create an Integration User" on page 372.

General Structure of an Application

There is only one static factory, the UcmdbServiceFactory. This factory is the entry point for an application. The UcmdbServiceFactory exposes getServiceProvider methods. These methods return an instance of the **UcmdbServiceProvider** interface.

The client creates other objects using interface methods. For example, to create a new query definition, the client:

1. Gets the query service from the main CMDB service object.
2. Gets a query factory object from the service object.
3. Gets a new query definition from the factory.

```
UcmdbServiceProvider provider =
   UcmdbServiceFactory.getServiceProvider(HOST_NAME, PORT);
UcmdbService ucmdbService =
   provider.connect(provider.createCredentials(USERNAME,
   PASSWORD), provider.createClientContext("Test"));
TopologyQueryService queryService = ucmdbService.getTopologyQueryService();
TopologyQueryFactory factory = queryService.getFactory();
QueryDefinition queryDefinition = factory.createQueryDefinition("Test
Query");
queryDefinition.addNode("Node").ofType("host");
Topology topology = queryService.executeQuery(queryDefinition);
System.out.println("There are " + topology.getAllCIs().size() + " hosts in
uCMDB");
```

The services available from **UcmdbService** are:

Service Methods	Use
getAuthorizationModelService	Perform authorization operations (create users and user groups, assign roles to users and groups, and so on).
getClassModelService	Information about types of CIs and relations

Service Methods	Use
getConfigurationService	Infrastructure settings management, for server configuration
getDataStoreMgmtService	Query data store information including which CIs and attributes will be federated.
getDDMConfigurationService	Configure the Data Flow Management system
getDDMManagementService	Analyze and view the progress, results, and errors of the Data Flow Management system
getDDMZoneService	Import and export management zones (with their activities).
getHistoryService	Information about history of monitored CIs (changes, removals, and so on)
getImpactAnalysisService	Run impact analysis scenario (also known as **correlation**).
getLicensingService	Query information about licenses installed in the system.
getMultipleCMDBService	Convert between global IDs and UCMDB IDs.
getMultiTenancyService	Create, read, update, and delete tenants.
getPersistencyService	Persist binary data into key-value pairs.
getQueryManagementService	Manage access to queries - save, delete, list existing. Also provides query validation and discovery of queries dependencies.
getReconciliationService	Supplies identification and merging capabilities.
getResourceBundleManagementService	Resource tagging ("bundling" services). Allows explicit creation of new tags and removal of tags from all tagged resources.
getResourceManagementService	Deploy resource packages (of TQL queries, views, users, and so on) to the system.
getSecurityService	Verify whether credentials are valid.
getServerService	Query generic information about the system.
getSnapshotService	Provide services for managing snapshots (get, save, compare, and so on)

Service Methods	Use
getSoftwareSignatureService	Define software items to be discovered by the Data Flow Management system
getStateService	Provide services for managing states (list, add, remove, and so on)
getSystemHealthService	Provide system health services (basic system performance indicators, capacity and availability metrics)
getTopologyQueryService	Get information about the IT universe
getTopologyUpdateService	Change information in the IT universe
getUcmdbVersion	Query UCMDB and content pack versions and build information.
getViewArchiveService	View result archiving services. Allows saving the current view result and retrieving previously saved results.
getViewService	View execution service (execute definition, execute saved) and management service (save, delete, list existing). Also provides view validation and dependencies discovery.

The client communicates with the server over HTTP(S).

Put the API Jar File in the Classpath

The use of this API set requires the file **ucmdb-api.jar**. You can download the file by entering `http://<localhost>:8080` in a Web browser where `localhost` is the machine where UCMDB is installed and clicking the **API Client Download** link.

Put the **.jar** file in the classpath before compiling or running your application.

Note: Use of the UCMDB Java API Jar requires you to have JRE version 6 or later installed.

Create an Integration User

You can create a dedicated user for integrations between other products and UCMDB. This user enables a product that uses the UCMDB client SDK to be authenticated in the server SDK and execute the APIs. Applications written with this API set must log on with integration user credentials.

Caution: It is also possible to connect with a regular UCMDB user (for instance, admin). However, this option is not recommended. To connect with a UCMDB user, you must grant the user API authentication permission.

To create an integration user:

1. Launch the Web browser and enter the server address, as follows:

   `http://localhost:8080/jmx-console`

   You may have to log in with a user name and password.

2. Under UCMDB, click **service=UCMDB Authorization Services**.

3. Locate the **createUser** operation. This method accepts the following parameters:

   - **customerId**. The customer ID.
   - **username**. The integration user's name.
   - **userDisplayName**. The integration user's display name.
   - **userLoginName**. The integration user's login name.
   - **password**. The integration user's password.

     The default password policy requires the UCMDB password to include at least one of each of the four following types of characters:

     - Uppercase alphabetic characters
     - Lowercase alphabetic characters
     - Numeric characters
     - Symbol characters ,\:/. _?&%=+-[]()|

     It also requires the password to adhere to the minimum length, which is set by the **Password minimum length** setting.

4. Click **Invoke**.

5. In a single-tenant environment, locate the **setRolesForUser** method and enter the following

parameters:

- **userName**. The integration user's name.
- **roles.** SuperAdmin.

Click **Invoke**.

6. In a multi-tenant environment, locate the **grantRolesToUserForAllTenants** method and enter the following parameters to assign the role in connection with all tenants:

   - **userName**. The integration user's name.
   - **roles.** SuperAdmin.

   Click **Invoke**.

   Alternatively, to assign the role in connection with specific tenants, invoke the **grantRolesToUserForTenants** method, using the same userName and roles parameter values. For the **tenantNames** parameter, enter the required tenants.

7. Either create more users, or close the JMX console.
8. Log on to UCMDB as an administrator.
9. From the **Administration** tab, run **Package Manager**.
10. Click the **Create custom package** icon.
11. Enter a name for the new package, and click **Next**.
12. In the Resource Selection tab, under **Settings**, click **Users**.
13. Select a user or users that you created using the JMX console.
14. Click **Next** and then **Finish**. Your new package appears in the Package Name list in Package Manager.
15. Deploy the package to the users who will run the API applications.

    For details, see the section "How to Deploy a Package" in the *HP Universal CMDB Administration Guide*.

Note: The integration user is per customer. To create a stronger integration user for cross-customer usage, use a **systemUser** with the **isSuperIntegrationUser** flag set to **true**. Use the **systemUser** methods (**removeUser**, **resetPassword**, **UserAuthenticate**, and so on).

There are two out-of-the-box system users. It is recommended to change their passwords after installation using the **resetPassword** method.

- **sysadmin/sysadmin**
- **UISysadmin/UISysadmin** (this user is also the **SuperIntegrationUser**).

  If you change the UISysadmin password using **resetPassword**, you must do the following:

  i. In the JMX Console, locate the **UCMDB-UI:name=UCMDB Integration** service.

  ii. Run **setCMDBSuperIntegrationUser** with the user name and new password of the integration user.

UCMDB API Use Cases

The use cases listed in this section assume two systems:

- HP Universal CMDB server
- A third-party system that contains a repository of configuration items

This section includes the following topics:

Populating the CMDB

Use cases:

- A third-party asset management updates the CMDB with information available only in asset management
- A number of third-party systems populate the CMDB to create a central CMDB that can track changes and perform impact analysis
- A third-party system creates Configuration Items and Relations according to third-party business logic, to leverage the UCMDB query capabilities

Querying the CMDB

Use cases:

- A third-party system gets the Configuration Items and Relations that represent the SAP system by retrieving the results of the SAP TQL
- A third-party system gets the list of Oracle servers that have been added or changed in the last five hours
- A third-party system gets the list of servers whose host name contains the **lab** substring
- A third-party system finds the elements related to a given CI by getting its neighbors

Querying the Class Model

Use cases:

- A third-party system enables users to specify the set of data to be retrieved from the CMDB. A user interface can be built over the class model to show users the possible properties and prompt them for required data. The user can then choose the information to be retrieved.
- A third-party system explores the class model when the user cannot access the UCMDB user interface.

Analyzing Change Impact

Use case:

- A third-party system outputs a list of the business services that could be impacted by a change on a specified host.

Examples

See the following code samples:

- Create a Connection
- Create and Execute an Ad Hoc Query
- Create and Execute a View
- Add and Delete Data
- Execute an Impact Analysis
- Query the Class Model
- Query a History Sample

These files are located in the following directory:

\\<UCMDB root directory>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\JavaSDK_Samples

Chapter 11: HP Universal CMDB Web Service API

This chapter includes:

Conventions

This chapter uses the following conventions:

- UCMDB refers to the Universal Configuration Management database itself. HP Universal CMDB refers to the application.
- UCMDB elements and method arguments are spelled in the case in which they are specified in the schema. An element or argument to a method is not capitalized. For example, a `relation` is an element of type `Relation` passed to a method.

For full documentation on the request and response structures, refer to the HP UCMDB Web Service API Reference. These files are located in the following folder:

<UCMDB root directory>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\CMDB_Schema\webframe.html

HP Universal CMDB Web Service API Overview

Note: Use this chapter in conjunction with the UCMDB schema documentation, available in the online Documentation Library.

The HP Universal CMDB Web Service API is used to integrate applications with the HP Universal CMDB (UCMDB). The API provides methods to:

- Add, remove, and update CIs and relations in the CMDB
- Retrieve information about the class model
- Retrieve impact analyses
- Retrieve information about configuration items and relationships
- Manage credentials: view, add, update, and remove
- Manage jobs: view status, activate, and deactivate
- Manage probe ranges: view, add, and update
- Manage triggers: add or remove a trigger CI, and add, remove, or disable a trigger TQL
- View general data on domains and Probes

Methods for retrieving information about configuration items and relationships generally use the Topology Query Language (TQL). For details, see Topology Query Language in the *HP Universal CMDB Modeling Guide*.

Users of the HP Universal CMDB Web Service API should be familiar with:

- The SOAP specification
- An object-oriented programming language such as C++, C# or Java
- HP Universal CMDB
- Data Flow Management

This section includes the following topics:

Uses of the API

The UCMDB Web Services API is used to fulfill a number of business requirements. For example:

- A third-party system can query the class model for information about available configuration items (CIs).
- A third-party asset management tool can update the CMDB with information available only to that tool, thereby unifying its data with data collected by HP applications.
- A number of third-party systems can populate the CMDB to create a central CMDB that can track changes and perform impact analysis.
- A third-party system can create entities and relations according to its business logic, and then write the data to the CMDB to take advantage of the CMDB query capabilities.
- Other systems, such as the Release Control (CCM) system, can use the Impact Analysis methods for change analysis.

Permissions

To access the WSDL file for the web service, go to: **http://localhost:8080/axis2/services/UcmdbService?wsdl**. You will need to provide server administrator user credentials to view the WSDL file.

Note: The Axis2 administration console is not accessible.

The user must have the **Run Legacy API** general action permission in order to log in.

The following table displays the additional required permissions for each Web Service API command:

Web Service API command	Required Permissions
addCIsAndRelations deleteCIsAndRelations updateCIsAndRelations	General Action: **Data Update**

Web Service API command	Required Permissions
executeTopologyQueryByName(AdHoc) executeTopologyQueryByNameWithParameters(AdHoc) executeTopologyQueryWithParameters(AdHoc)	General Action: **Run Query by Definition** For each query: **View** permission
getTopologyQueryExistingResultByName getTopologyQueryResultCountByName releaseChunks pullTopologyMapChunks getCINeighbours getFilteredCIsByType getCIsById getCIsByType getRelationsById	General Action: **View CIs** For each query: **View** permission
getQueryNameOfView	General Action: **View CIs** For each view: **View** permission
getChangedCIs	General Action: **View History**, **View CIs**
calculateImpact getImpactPath getImpactRulesByGroupName getImpactRulesByNamePrefix	General Action: **Run Impact Analysis**
getAllClassesHierarchy getClassAncestors getCmdbClassDefinition	None

Note: When the root context has been changed in UCMDB, follow these steps to enable access to the Web Service API:

1. Open the **\UCMDB\UCMDBServer\deploy\axis2\WEB-INF\web.xml** configuration file and locate the following section:

   ```
   <servlet-class>
   org.apache.axis2.transport.http.AxisServlet
   </servlet-class>
   ```

   Add the following lines after that:

```
<init-param>
<param-name>axis2.find.context</param-name>
<param-value>false</param-value>
</init-param>
```

2. Open the **\UCMDB\UCMDBServer\deploy\axis2\WEB-INF\conf\axis2.xml** configuration file and locate the following line:

   ```
   <parameter name="enableSwA" locked="false">false</parameter>
   ```

   Add the following line after it:

   ```
   <parameter name="contextRoot" locked="false">test1/setup1/axis2
   </parameter>
   ```

   where **test1/setup1** is your root context.

   (To remove the root context, remove the text added to the path.)

3. Restart the UCMDB server.

Call the HP Universal CMDB Web Service

Use standard SOAP programming techniques in the HP Universal CMDB Web Service API to enable calling server-side methods. If the statement cannot be parsed or if there is a problem invoking the method, the API methods throw a `SoapFault` exception. When a `SoapFault` exception is thrown, the UCMDB populates one or more of the error message, error code, and exception message fields. If there is no error, the results of the invocation are returned.

SOAP programmers can access the WSDL at:

http://<server>[:port]/axis2/services/UcmdbService?wsdl

The port specification is only necessary for non-standard installations. Consult your system administrator for the correct port number.

The URL for calling the service is:

http://<server>[:port]/axis2/services/UcmdbService

For examples of connecting to the CMDB, see "UCMDB Web Service API Use Cases" on page 415.

Query the CMDB

The CMDB is queried using the APIs described in " UCMDB Query Methods" on page 395. The queries and the returned CMDB elements always contain real UCMDB IDs. For examples of using query methods, see Query Example.

This section includes the following topics:

Just In Time Response Calculation

For all query methods, the UCMDB server calculates the values requested by the query method when the request is received, and returns results based on the latest data. The result is always calculated at the time the request is received, even if the TQL query is active and there is a previously calculated result. Therefore, the results of running a query returned to the client application may be different from the results of the same query displayed on the user interface.

Tip: If your application uses the results of a given query more than once and the data is not expected to change significantly between uses of the result data, you can improve performance by having the client application store the data rather than repeatedly running the query.

Processing Large Responses

The response to a query always includes the structures for the data requested by the query method, even if no actual data is being transmitted. For many methods where the data is a collection or map, the response also includes the `ChunkInfo` structure, comprised of `chunksKey` and `numberOfChunks`. The `numberOfChunks` field indicates the number of chunks containing data that must be retrieved.

The maximum transmission size of data is set by the system administrator. If the data returned from the query is larger than the maximum size, the data structures in the first response contain no meaningful information, and the value of the `numberOfChunks` field is `2` or greater. If the data is not larger than the maximum, the `numberOfChunks` field is 0 (zero), and the data is transmitted in the first response. Therefore, in processing a response, check the `numberOfChunks` value first. If it is greater than `1`, discard the data in the transmission and request the chunks of data. Otherwise, use the data in the response.

For information on handling chunked data, see "pullTopologyMapChunks" on page 405 and "releaseChunks" on page 407.

Specifying Properties to Return

CIs and relations generally have many properties. Some methods that return collections or graphs of these items accept input parameters that specify which property values to return with each item that matches the query. The CMDB does not return empty properties. Therefore, the response to a query may have fewer properties than requested in the query.

This section describes the types of sets used to specify the properties to return.

Properties can be referenced in two ways:

- By their names
- By using names of predefined properties rules. Predefined properties rules are used by the CMDB to create a list of real property names.

When an application references properties by name, it passes a `PropertiesList` element.

> **Tip:** Whenever possible, use `PropertiesList` to specify the names of the properties in which you are interested, rather than a rule-based set. The use of predefined properties rules usually results in returning more properties than needed, and incurs a price in performance.

There are two types of predefined properties: qualifier properties and simple properties.

- **Qualifier properties.** Use this when the client application should pass a `QualifierProperties` element (a list of qualifiers that can be applied to properties). The CMDB converts the list of qualifiers passed by the client application to the list of the properties to which at least one of the qualifiers applies. The values of these properties are returned with the `CI` or `Relation` elements.
- **Simple properties.** To use simple rule-based properties, the client application passes a `SimplePredefinedProperty` or `SimpleTypedPredefinedProperty` element. These elements

contain the name of the rule by which the CMDB generates the list of properties to return. The rules that can be specified in a `SimplePredefinedProperty` or `SimpleTypedPredefinedProperty` element are `CONCRETE`, `DERIVED`, and `NAMING`.

Concrete Properties

Concrete properties are the set of properties defined for the specified CIT. The properties added by derived classes are not returned for instances of those derived classes.

A collection of instances returned by a method may consist of instances of a CIT specified in the method invocation and instances of CITs that inherit from that CIT. The derived CITs inherit the properties of the specified CIT. In addition, the derived CITs extend the parent CIT by adding properties.

Example of Concrete Properties:

CIT `T1` has properties `P1` and `P2`. CIT `T11` inherits from `T1` and extends `T1` with properties `P21` and `P22`.

The collection of CIs of type `T1` includes the instances of `T1` and `T11`. The concrete properties of all instances in this collection are `P1` and `P2`.

Derived Properties

Derived properties are the set of properties defined for the specified CIT and, for each derived CIT, the properties added by the derived CIT.

Example of Derived Properties:

Continuing the example from concrete properties, the derived properties of instances of `T1` are `P1` and `P2`. The derived properties of instances of `T11` are `P1`, `P2`, `P21`, and `P22`.

Naming Properties

The naming properties are `display_label` and `data_name`.

Other Property Specification Elements

- **PredefinedProperties**

`PredefinedProperties` can contain a `QualifierProperties` element and a `SimplePredefinedProperty` element for each of the other possible rules. A `PredefinedProperties` set does not necessarily contain all types of lists.

- **PredefinedTypedProperties**

  `PredefinedTypedProperties` is used to apply a different set of properties to each CIT. `PredefinedTypedProperties` can contain a `QualifierProperties` element and a `SimpleTypedPredefinedProperty` element for each of the other applicable rules. Because `PredefinedTypedProperties` is applied to each CIT individually, derived properties are not relevant. A `PredefinedProperties` set does not necessarily contain all applicable types of lists.

- **CustomProperties**

  `CustomProperties` can contain any combination of the basic `PropertiesList` and the rule-based property lists. The properties filter is the union of all the properties returned by all the lists.

- **CustomTypedProperties**

  `CustomTypedProperties` can contain any combination of the basic `PropertiesList` and the applicable rule-based property lists. The properties filter is the union of all the properties returned by all the lists.

- **TypedProperties**

  `TypedProperties` is used to pass a different set of properties for each CIT. `TypedProperties` is a collection of pairs composed of type names and properties sets of all types. Each properties set is applied only to the corresponding type.

Update the CMDB

You update the CMDB with the update APIs. For details of the API methods, see " UCMDB Update Methods" on page 407.

This task includes the following steps:

- "UCMDB Update Parameters" on the next page
- "Use of ID Types with Update Methods" on the next page

UCMDB Update Parameters

This topic describes the parameters used only by the service's update methods.

- **CIsAndRelationsUpdates**

  The `CIsAndRelationsUpdates` type consists of `CIsForUpdate`, `relationsForUpdate`, `referencedRelations`, and `referencedCIs`. A `CIsAndRelationsUpdates` instance does not necessarily include all three elements.

  `CIsForUpdate` is a CIs collection. `relationsForUpdate` is a `Relations` collection. The `CI` and `relation` elements in the collections have a `props` element. When creating a CI or relation, properties that have either the `required` attribute or the `key` attribute in the CI Type definition must be populated with values. The items in these collections are updated or created by the method.

  `referencedCIs` and `referencedRelations` are collections of CIs that are already defined in the CMDB. The elements in the collection are identified with a temporary ID in conjunction with all the key properties. These items are used to resolve the identities of CIs and relations for update. They are never created or updated by the method.

  Each of the `CI` and `relation` elements in these collections has a properties collection. New items are created with the property values in these collections.

Use of ID Types with Update Methods

The following describes ID CITs, and CIs and relations. When the ID is not a real CMDB ID, the type and key attributes are required.

- **Deleting or Updating Configuration Items**

  A temporary or empty ID may be used by the client when calling a method to delete or update an item. In this case, the CI type and the "Key Attributes" that identify the CI must be set.

- **Deleting or Updating Relations**

  When deleting or updating relations, the relation ID can be empty, temporary, or real.

  If a CI's ID is temporary, the CI must be passed in the `referencedCIs` collection and its key attributes must be specified. For details, see referencedCIs in "CIsAndRelationsUpdates" above.

- **Inserting New Configuration Items into the CMDB**

It is possible to use either an empty ID or a temporary ID to insert a new CI. However, if the ID is empty, the server cannot return the real CMDB ID in the structure `createIDsMap` because there is no `clientID`. For details, see "addCIsAndRelations" on page 408 and " UCMDB Query Methods" on page 395.

- **Inserting New Relations into the CMDB**

  The relation ID can be either temporary or empty. However, if the relation is new but the configuration items on either end of the relation are already defined in the CMDB, then those CIs that already exist must be identified by a real CMDB ID or be specified in a `referencedCIs` collection.

Query the UCMDB Class Model

The class model methods return information about CITs and relations. The class model is configured using the CI Type Manager. For details, see CI Type Manager in the *HP Universal CMDB Modeling Guide*.

This section provides information on the following methods that return information about CITs and relations:

- "getClassAncestors" below
- "getAllClassesHierarchy" on the next page
- "getCmdbClassDefinition" on the next page

getClassAncestors

The `getClassAncestors` method retrieves the path between the given CIT and its root, including the root.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
className	The type name. For details, see "Type Name" on page 392.

Output

Parameter	Comment
classHierarchy	A collection of pairs of class names and parent class name.
comments	For internal use only.

getAllClassesHierarchy

The `getAllClassesHierarchy` method retrieves the entire class model tree.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.

Output

Parameter	Comment
classesHierarchy	A collection of pairs of class name and parent class name.
comments	For internal use only.

getCmdbClassDefinition

The `getCmdbClassDefinition` method retrieves information about the specified class.

If you use `getCmdbClassDefinition` to retrieve the key attributes, you must also query the parent classes up to the base class. `getCmdbClassDefinition` identifies as key attributes only those attributes with the `ID_ATTRIBUTE` set in the class definition specified by `className`. Inherited key attributes are not recognized as key attributes of the specified class. Therefore, the complete list of key attributes for the specified class is the union of all the keys of the class and of all its parents, up to the root.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on the next page.
className	The type name. For details, see "UCMDB General Parameters " below.

Output

Parameter	Comment
cmdbClass	The class definition, consisting of `name`, `classType`, `displayLabel`, `description`, `parentName`, qualifiers, and attributes.
comments	For internal use only.

Query for Impact Analysis

The `Identifier` in the impact analysis methods points to the service's response data. It is unique for the current response and is discarded from the server's memory cache after 10 minutes of non-use.

For examples of the use of the impact analysis methods, see Impact Analysis Example.

UCMDB General Parameters

This section describes the most common parameters of the service's methods.

This section includes the following topics:

- "Configuration Item (CI)" on the next page
- "Relation" on page 393

CmdbContext

All UCMDB Web Service API service invocations require a `CmdbContext` argument. `CmdbContext` is a `callerApplication` string that identifies the application that invokes the service. `CmdbContext` is used for logging and troubleshooting.

ID

Every `CI` and Relation has an `ID` field. It consists of a case-sensitive ID string and an optional `temp` flag, indicating whether the ID is temporary.

Key Attributes

For identifying a `CI` or `Relation` in some contexts, key attributes can be used in place of a CMDB ID. Key attributes are those attributes with the ID_ATTRIBUTE set in the class definition.

In the user interface, the key attributes have a key icon next to them in the list of Configuration Item Type attributes in the user interface. For details, see Add/Edit Attribute Dialog Box in the *HP Universal CMDB Modeling Guide*. For information about identifying the key attributes from within the API client application, see "getCmdbClassDefinition" on page 389.

ID Types

An `ID` element can contain a real ID, or a temporary ID.

A real ID is a string assigned by the CMDB that identifies an entity in the database. A temporary ID can be any string that is unique in the current request.

A temporary ID can be assigned by the client and often represents the ID of the CI as stored by the client. It does not necessarily represent an entity already created in the CMDB. When a temporary ID is passed by the client, if the CMDB can identify an existing data configuration item using the CI key properties, that CI is used as appropriate for the context as if it had been identified with a real ID.

CIProperties

A `CIProperties` element is composed of collections, each containing a sequence of name-value elements that specify properties of the type indicated by the collection name. None of the collections are required, so the `CIProperties` element can contain any combination of collections.

CIProperties are used by CI and Relation elements. For details, see "Configuration Item (CI)" below and "Relation" on the next page.

The properties collections are:

- datePropsCollection of DateProp elements
- doublePropsCollection of DoubleProp elements
- floatPropsCollection of FloatProp elements
- intListPropsCollection of intListProp elements
- intPropsCollection of IntProp elements
- strPropsCollection of StrProp elements
- strListPropsCollection of StrListProp elements
- longPropsCollection of LongProp elements
- bytesPropsCollection of BytesProp elements
- xmlPropsCollection of XmlProp elements

Type Name

The type name is the class name of a configuration item type or relation type. The type name is used in code to refer to the class. It should not be confused with the display name, which is seen on the user interface where the class is mentioned, but which is meaningless in code.

Configuration Item (CI)

A CI element is composed of an ID, a type, and a props collection.

When using " UCMDB Update Methods" to update a CI, the ID element can contain a real CMDB ID or a client-assigned temporary ID. If a temporary ID is used, set the temp flag to true. When deleting an item, the ID can be empty. " UCMDB Query Methods" take real IDs as input parameters and return real IDs in the query results.

The type can be any type name defined in the CI Type Manager. For details, see CI Type Manager in the HP Universal CMDB Modeling Guide.

The props element is a CIProperties collection. For details, see "UCMDB General Parameters " on page 390.

Relation

A `Relation` is an entity that links two configuration items. A `Relation` element is composed of an `ID`, a `type`, the identifiers of the two items being linked `(end1ID` and `end2ID),` and a `props` collection.

When using " UCMDB Update Methods" to update a `Relation`, the value of the `Relation`'s ID can be a real CMDB ID or a temporary ID. When deleting an item, the `ID` can be empty. " UCMDB Query Methods" take real `ID`s as input parameters and return real `ID`s in the query results.

The relation type is the `Type Name` of the UCMDB class from which the relation is instantiated. The type can be any of the relation types defined in the CMDB. For further information on classes or types, see "Query the UCMDB Class Model " on page 388.

For details, see CI Type Manager in the *HP Universal CMDB Modeling Guide*.

The two relation end IDs must not be empty IDs because they are used to create the ID of the current relation. However, they both can have temporary IDs assigned to them by the client.

The `props` element is a CIProperties collection. For details, see "CIProperties" on page 391.

UCMDB Output Parameters

This section describes the most common output parameters of the service methods. For more details, refer to the online schema documentation.

This section includes the following topics:

CIs

CIs is a collection of CI elements.

ShallowRelation

A ShallowRelation is an entity that links two configuration items, composed of an ID, a type, and the identifiers of the two items being linked (end1ID and end2ID). The relation type is the Type Name of the CMDB class from which the relation is instantiated. The type can be any of the relation types defined in the CMDB.

Topology

Topology is a graph of CI elements and relations. A Topology consists of a CIs collection and a Relations collection containing one or more Relation elements.

CINode

CINode is composed of a CIs collection with a label. The label in the CINode is the label defined in the node of the TQL used in the query.

RelationNode

RelationNode is a set of Relation collections with a label. The label in the RelationNode is the label defined in the node of the TQL used in the query.

TopologyMap

TopologyMap is the output of a query calculation that matches a TQL query. The labels in the TopologyMap are the node labels defined in the TQL used in the query.

The data of TopologyMap is returned in the following form:

- CINodes. This is one or more CINode (see "CINode" above).
- relationNodes. This is one or more RelationNode (see "RelationNode" above).

The labels in these two structures order the lists of configuration items and relations.

ChunkInfo

When a query returns a large amount of data, the server stores the data, divided into segments called chunks. The information the client uses to retrieve the chunked data is located in the ChunkInfo

structure returned by the query. `ChunkInfo` is composed of the `numberOfChunks` that must be retrieved and the `chunksKey`. The `chunksKey` is a unique identifier of the data on the server for this specific query invocation.

For more information, see "Processing Large Responses" on page 383.

UCMDB Query Methods

This section provides information on the following methods:

executeTopologyQueryByNameWithParameters

The `executeTopologyQueryByNameWithParameters` method retrieves a `topologyMap` element that matches the specified parameterized query.

The values for the query parameters are passed in the `parameterizedNodes` argument. The specified TQL must have unique labels defined for each `CINode` and each `relationNode` or the method invocation fails.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
queryName	The name of the parameterized TQL in the CMDB for which to get the map.
parameterizedNodeList	The conditions each node must meet to be included in the query results.
queryTypedProperties	A collection of sets of properties to retrieve for items of a specific Configuration Item Type.

Output

Parameter	Comment
topologyMap	For details, see "TopologyMap" on page 394.
chunkInfo	For details, see "ChunkInfo" on page 394 and "Processing Large Responses" on page 383.

executeTopologyQueryWithParameters

The `executeTopologyQueryWithParameters` method retrieves a `topologyMap` element that matches the parameterized query.

The query is passed in the `queryXML` argument. The values for the query parameters are passed in the `parameterizedNodeList` argument. The TQL must have unique labels defined for each `CINode` and each `relationNode`.

The `executeTopologyQueryWithParameters` method is used to pass ad hoc queries, rather than accessing a query defined in the CMDB. You can use this method when you do not have access to the UCMDB user interface to define a query, or when you do not want to save the query to the database.

To use an exported TQL as the input of this method, do the following:

1. Launch the Web browser and enter the following address: **http://localhost:8080/jmx-console**.

   You may have to log in with a user name and password.

2. Click **UCMDB:service=TQL Services**.
3. Locate the **exportTql** operation.
   - In the **customerId** parameter box, enter **1** (the default).
   - In the **patternName** parameter box, enter a valid TQL name.
4. Click **Invoke**.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
queryXML	An XML string representing a TQL without resource tags.
parameterizedNodeList	The conditions each node must meet to be included in the query results.

Output

Parameter	Comment
topologyMap	For details, see "TopologyMap" on page 394.
chunkInfo	For details, see "ChunkInfo" on page 394 and "Processing Large Responses" on page 383.

getChangedCIs

The `getChangedCIs` method returns the change data for all CIs related to the specified CIs.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
ids	The list of the IDs of the root CIs whose related CIs are checked for changes. Only real CMDB IDs are valid in this collection.

Parameter	Comment
fromDate	The beginning of the period in which to check if CIs changed.
toDate	The end of the period in which to check if CIs changed.

Output

Parameter	Comment
getChangedCIsResponseList	Zero or more collections of `ChangedDataInfo` elements.

getCINeighbours

The `getCINeighbours` method returns the immediate neighbors of the specified CI.

For example, if the query is on the neighbors of CI `A`, and CI `A` contains CI `B` which uses CI `C`, CI `B` is returned, but CI `C` is not. That is, only neighbors of the specified type are returned.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
ID	The ID of the CI for which to retrieve the neighbors. This must be a real CMDB or global ID.
neighbourType	The CIT name of the neighbors to retrieve. Neighbors of the specified type and of types derived from that type are returned. For details, see "Type Name" on page 392.
CIProperties	The data to be returned on each configuration item, called the Query Layout in the user interface. For details, see "TypedProperties" on page 386.
relationProperties	The data to be returned on each relation (called the Query Layout in the user interface). For details, see "TypedProperties" on page 386.

Output

Parameter	Comment
topology	For details, see "Topology" on page 394.

Parameter	Comment
comments	For internal use only.

getClsByID

The `getCIsByID` method retrieves configuration items by their CMDB or global IDs.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
ClsTypedProperties	A typed properties collection. For details, see "Other Property Specification Elements" on page 385.
IDs	Only real CMDB or global IDs are valid in this collection.

Output

Parameter	Comment
CIs	A collection of CI elements.
chunkInfo	For details, see "ChunkInfo" on page 394 and "Processing Large Responses" on page 383.

getClsByType

The `getCIsByType` method returns the collection of configuration items of the specified type and of all types that inherit from the specified type.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
type	The class name. For details, see "Type Name" on page 392.
properties	The data to be returned on each configuration item. For details, see "CustomProperties" on page 386.

Output

Parameter	Comment
CIs	A collection of CI elements.
chunkInfo	For details, see: "ChunkInfo" on page 394 and "Processing Large Responses" on page 383.

getFilteredCIsByType

The `getFilteredCIsByType` method retrieves the CIs of the specified type that meet the conditions used by the method. A condition is comprised of:

- A name field containing the name of a property
- An operator field containing a comparison operator
- An optional value field containing a value or list of values

Together, they form a Boolean expression:
`<item>.property.value [operator] <condition>.value`

For example, if the condition name is `root_actualdeletionperiod`, the condition value is `40` and the operator is `Equal`, the Boolean statement is:

`<item>.root_actualdeletionperiod.value = = 40`

The query returns all items whose `root_actualdeletionperiod` is `40`, assuming there are no other conditions.

If the `conditionsLogicalOperator` argument is `AND`, the query returns the items that meet all conditions in the `conditions` collection. If `conditionsLogicalOperator` is `OR`, the query returns the items that meet at least one of the conditions in the `conditions` collection.

The following table lists the comparison operators:

Operator	Type of Condition/Comments
ChangedDuring	Date This is a range check. The condition value is specified in hours. If the value of the date property lies in the range of the time the method is invoked plus or minus the condition value, the condition is true. For example, if the condition value is `24`, the condition is true if the value of the date property is between yesterday at this time and tomorrow at this time. **Note:** The name `ChangedDuring` is kept to preserve backward compatibility. In previous versions, the operator was used only with create and modify time properties.
Equal	String and numerical
EqualIgnoreCase	String
Greater	Numerical
GreaterEqual	Numerical
In	String, numerical, and list The condition's value is a list. The condition is true if the value of the property is one of the values in the list.
InList	List The condition's value and the property's value are lists. The condition is true if all the values in the condition's list also appear in the item's property list. There can be more property values than specified in the condition without affecting the truth of the condition.
IsNull	String, numerical, and list The item's property has no value. When operator `IsNull` is used, the value of the condition is ignored, and in some cases can be nil.
Less	Numerical
LessEqual	Numerical

Operator	Type of Condition/Comments
Like	String The condition's value is a substring of the value of the property's value. The condition's value must be bracketed with percentage signs (%). For example, `%Bi%` matches `Bismark` and `Bay of Biscay`, but not `biscuit`.
LikeIgnoreCase	String Use the `LikeIgnoreCase` operator as you use the `Like` operator. The match, however is not case-sensitive. Therefore, `%Bi%` matches `biscuit`.
NotEqual	String and numerical
UnchangedDuring	Date This is a range check. The condition value is specified in hours. If the value of the date property is in the range of the time the method is invoked plus or minus the condition value, the condition is false. If it lies outside that range, the condition is true. For example, if the condition value is `24`, the condition is true if the value of the date property is before yesterday at this time or after tomorrow at this time. **Note:** The name `UnchangedDuring` is kept to preserve backward compatibility. In previous versions, the operator was used only with create and modify time properties.

Example of Setting Up a Condition:

```
FloatCondition fc = new FloatCondition();
FloatProp fp = new FloatProp();
fp.setName("attr_name");
fp.setValue(11f);
fc.setCondition(fp);
fc.setFloatOperator(FloatCondition.FloatOperator.EQUAL);
```

Example of Querying for Inherited Properties:

The target CI is `sample` which has two attributes, `name` and `size`. `sampleII` extends the CI with two attributes, `level` and `grade`. This example sets up a query for the properties of `sampleII` that were inherited from `sample` by specifying them by name.

```
GetFilteredCIsByType request = new GetFilteredCIsByType()
request.setCmdbContext(cmdbContext)
request.setType("sampleII");
CustomProperties customProperties = new CustomProperties();
```

```
PropertiesList propertiesList = new PropertiesList();
propertiesList.setPropertyNames(Arrays.asList("name","size"));
customProperties.setPropertiesList(propertiesList);
request.setProperties(customProperties);
```

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
type	The class name. For details, see "Type Name" on page 392. The type can be any of the types defined using the CI Type Manager. For details, see CI Type Manager in the *HP Universal CMDB Modeling Guide*.
properties	The data to be returned on each CI (called the Query Layout in the user interface). For details, see "CustomProperties" on page 386.
conditions	A collection of name-value pairs and the operators that relate one to the other. For example, `host_hostname like QA`.
conditionsLogicalOperator	• **AND.** All the conditions must be met. • **OR**. At least one of the conditions must be met.

Output

Parameter	Comment
CIs	Collection of CI elements.
chunkInfo	For details, see "ChunkInfo" on page 394 and "Processing Large Responses" on page 383.

getQueryNameOfView

The `getQueryNameOfView` method retrieves the name of the TQL on which the specified view is based.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
viewName	The name of a view, which is a sub-set of the class model in the CMDB.

Output

Parameter	Comment
queryName	The name of the TQL in the CMDB on which the view is based.

getTopologyQueryExistingResultByName

The `getTopologyQueryExistingResultByName` method retrieves the most recent result of running the specified TQL. The call does not run the TQL. If there are no results from a previous run, nothing is returned.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
queryName	The name of a TQL.
queryTypedProperties	A collection of sets of properties to retrieve for items of a specific Configuration Item Type.

Output

Parameter	Comment
topologyMap	For details, see "TopologyMap" on page 394.
chunkInfo	For details, see "ChunkInfo" on page 394 and "Processing Large Responses" on page 383.

getTopologyQueryResultCountByName

The `getTopologyQueryResultCountByName` method retrieves the number of instances of each node that matches the specified query.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
queryName	The name of a TQL.
countInvisible	If true, the output includes CIs defined as invisible in the query.

Output

Parameter	Comment
getTopologyQueryResultCountByNameResponse	The number of instances matching the query.

pullTopologyMapChunks

The `pullTopologyMapChunks` method retrieves one of the chunks that contain the response to a method.

Each chunk contains a `topologyMap` element that is part of the response. The first chunk is numbered `1`, so the retrieval loop counter iterates from 1 to *<response object>*`.getChunkInfo().getNumberOfChunks()`.

For details, see "ChunkInfo" on page 394 and "Query the CMDB" on page 383.

The client application must be able to handle the partial maps.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.

Parameter	Comment
ChunkRequest	The number of the chunk to retrieve and the `ChunkInfo` that is returned by the query method.
queryTypedProperties	A collection of sets of properties to retrieve for items of a specific CI type.

Output

Parameter	Comment
topologyMap	For details, see "TopologyMap" on page 394.
comments	For internal use only.

Example of Handling Chunks:

```
GetCIsByType request =
     new GetCIsByType(cmdbContext, typeName, customProperties);
GetCIsByTypeResponse response =
     ucmdbService.getCIsByType(request);
ChunkRequest chunkRequest = new ChunkRequest();
chunkRequest.setChunkInfo(response.getChunkInfo());
for(int j=1; j<=response.getChunkInfo().getNumberOfChunks(); j++){
    chunkRequest.setChunkNumber(j);
    PullTopologyMapChunks req =new PullTopologyMapChunks
(cmdbContext,chunkRequest);
    PullTopologyMapChunksResponse res =
        ucmdbService.pullTopologyMapChunks(req);
        for(int m=0 ;
            m < res.getTopologyMap().getCINodes().sizeCINodeList();
            m++) {
            CIs cis =
            res.getTopologyMap().getCINodes().getCINode(m).getCIs();
            for(int i=0 ; i < cis.sizeCIList() ; i++) {
                // your code to process the CIs
            }
        }
}
```

```
GetCIsByType request =
     new GetCIsByType(cmdbContext, typeName, customProperties);
GetCIsByTypeResponse response =
     ucmdbService.getCIsByType(request);
ChunkRequest chunkRequest = new ChunkRequest();
chunkRequest.setChunkInfo(response.getChunkInfo());
for(int j=1 ; j <= response.getChunkInfo().getNumberOfChunks() ; j++) {
    chunkRequest.setChunkNumber(j);
```

```
    PullTopologyMapChunks req = new PullTopologyMapChunks(cmdbContext,
chunkRequest);
    PullTopologyMapChunksResponse res =
        ucmdbService.pullTopologyMapChunks(req);
        for(int m=0 ;
            m < res.getTopologyMap().getCINodes().getCINodes().size();
            m++) {
            CIs cis =
            res.getTopologyMap().getCINodes().getCINodes().get(m).getCIs();
            for(int i=0 ; i < cis.getCIs().size(); i++) {
                // your code to process the CIs
            }
        }
}
```

releaseChunks

The `releaseChunks` method frees the memory of the chunks that contain the data from the query.

Tip: The server discards the data after ten minutes. Calling this method to discard the data as soon as it has been read conserves server resources.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
chunksKey	The identifier of the data on the server that was chunked. The key is an element of `ChunkInfo`.

UCMDB Update Methods

This section provides information on the following methods:

- "removeCustomer" on page 410
- "updateCIsAndRelations" on page 410

addCIsAndRelations

The `addCIsAndRelations` method adds or updates CIs and relations.

If the CIs or relations do not exist in the CMDB, they are added and their properties are set according to the contents of the `CIsAndRelationsUpdates` argument.

If the CIs or relations do exist in the CMDB, they are updated with the new data, if `updateExisting` is **true**.

If `updateExisting` is **false**, `CIsAndRelationsUpdates` cannot reference existing configuration items or relations. Any attempt to reference existing items when `updateExisting` is false results in an exception.

If `updateExisting` is **true**, the add or update operation is performed without validating the CIs, regardless of the value of `ignoreValidation`.

If `updateExisiting` is **false** and `ignoreValidation` is **true**, the add operation is performed without validating the CIs.

If `updateExisiting` is **false** and `ignoreValidation` is **false**, the CIs are validated before the add operation.

Relations are never validated.

`CreatedIDsMap` is a map or dictionary of type `ClientIDToCmdbID` that connects the client's temporary IDs with the corresponding real CMDB IDs.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
updateExisting	Set to **true** to update items that already exist in the CMDB. Set to **false** to throw an exception if any item already exists.
CIsAndRelationsUpdates	The items to update or create. For details, see "CIsAndRelationsUpdates" on page 387.

Parameter	Comment
ignoreValidation	If true, no check is performed before updating the CMDB.
dataStore	Changer information.

Output

Parameter	Comment
createdIDsMapList	The list of client IDs mapped to CMDB IDs. For details, see the description above.
comments	For internal use only.

addCustomer

The `addCustomer` method adds a customer.

Input

Parameter	Comment
CustomerID	The numeric ID of the customer.

deleteCIsAndRelations

The `deleteCIsAndRelations` method removes the specified configuration items and relations from the CMDB.

When a CI is deleted and the CI is one end of one or more `Relation` items, those `Relation` items are also deleted.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
CIsAndRelationsUpdates	The items to delete. For details, see "CIsAndRelationsUpdates" on page 387
dataStore	Changer information.

removeCustomer

The `removeCustomer` method deletes a customer record.

Input

Parameter	Comment
CustomerID	The numeric ID of the customer.

updateCIsAndRelations

The `updateCIsAndRelations` method updates the specified CIs and relations.

Update uses the property values from the `CIsAndRelationsUpdates` argument. If any of the CIs or relations do not exist in the CMDB, an exception is thrown.

`CreatedIDsMap` is a map or dictionary of type `ClientIDToCmdbID` that connects the client's temporary IDs with the corresponding real CMDB IDs.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
CIsAndRelationsUpdates	The items to update. For details, see "CIsAndRelationsUpdates" on page 387.
ignoreValidation	If true, no check is performed before updating the CMDB.
dataStore	Changer information.

Output

Parameter	Comment
createdIDsMapList	The list of client IDs mapped to CMDB IDs. For details, see "addCIsAndRelations" on page 408.

UCMDB Impact Analysis Methods

This section provides information on the following methods:

calculateImpact

The `calculateImpact` method calculates which CIs are affected by a given CI according to the rules defined in the CMDB.

This shows the effect of an event triggering of the rule. The `identifier` output of `calculateImpact` is used as input for "getImpactPath" on the next page.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
impactCategory	The type of event that would trigger the rule being simulated.
IDs	A collection of CMDB or global ID elements.
impactRulesNames	A collection of `ImpactRuleName` elements.
severity	The severity of the triggering event.

Output

Parameter	Comment
impactTopology	For details, see "Topology" on page 394.
identifier	The key to the server response.

getImpactPath

The `getImpactPath` method retrieves the topology graph of the path between the affected CI and the CI that affects it.

The `identifier` output of "calculateImpact" on the previous page is used as the `identifier` input argument of `getImpactPath`.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
identifier	The key to the server response that was returned by calculateImpact.
relation	A Relation based on one of the "ShallowRelation"s returned by calculateImpact in the `impactTopology` element.

Output

Parameter	Comment
impactPathTopology	A `CIs` collection and an `ImpactRelations` collection.
comments	For internal use only.

An `ImpactRelations` element consists of an `ID`, `type`, `end1ID`, `end2ID`, `rule`, and `action`.

getImpactRulesByNamePrefix

The `getImpactRulesByNamePrefix` method retrieves rules using a prefix filter.

This method applies to impact rules that are named with a prefix that indicates the context to which they apply, for example, `SAP_myrule`, `ORA_myrule`, and so on. This method filters all impact rule names for those beginning with the prefix specified by the `ruleNamePrefixFilter` argument.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
ruleNamePrefixFilter	A string containing the first letters of the rule names to match.

Output

Parameter	Comment
impactRules	`impactRules` is composed of zero or more `impactRule`. An `impactRule`, which specifies the effect of a change, is composed of `ruleName`, `description`, `queryName`, and `isActive`.

Actual State Web Service API

The Actual State Web Service API is used primarily by the Service Manager to retrieve Actual State information for a specific CMDB ID or Global ID and a specific customer ID. The API finds a matching query under the folder **Integration/SM Query** and executes the TQL with the CMDB ID or Global ID as a condition, and returns the output of the query.

Web Service URL: http://[machine_name]:8080/axis2/services/ucmdbSMService

Web Service Schema: http://[machine_name]:8080/axis2/services/ucmdbSMService?xsd=xsd0

Flow

When the API method is called, it tries to find an appropriate Query in the **Integration/SM Query** folder. It tries to match the type of the requested CMDBID/GlobalID with one of the queries in the latter folder first by looking for a **QueryElement** with the name **Root**, and if one is not found it tries to use any **QueryNode** of the same type as the requested CMDBID/GlobalID. Once an appropriate Query and QueryNode are found, it puts the CMDBID/GlobalID as a condition on the QueryNode and executes the Query. The result is then returned to the caller of the API.

Manipulating the Result Using Transformations

In some cases you may want to apply additional transformations to the resulting XML (for example, to sum up the sizes of all the disks and add that sum as an additional attribute to the CI). To add additional transformations on the TQL results, place a resource called **[tql_name].xslt** in the adapter

configuration as follows: **Adapter Management > ServiceDeskAdapter7-1 >Configuration Files > [tql_name].xslt.**

Logs for the Actual State Web Service API

The log configuration for UCMDB resides at: **UCMDBServer/Conf/log** in the various ***.properties** files.

To view logs of the SM Actual State flow:

1. Open the **cmdb_soaapi.properties** file and change the log level to DEBUG as follows: **loglevel=DEBUG**.
2. Open the **fcmdb.properties** file and change the log level to DEBUG as follows: **loglevel=DEBUG**.
3. Wait 1 minute for the server to retrieve the changes.
4. Run the Actual State from the SM.
5. View the following log files at **UCMDBServer/Runtime/log**:
   - cmdb.soaapi.log
   - fcmdb.log

Enabling Actual State of Replicated CIs after Changing Root Context

If you have changed the root context that is used to access UCMDB, you must make the following configuration changes to enable Actual State of Replicated CIs:

1. Under **UCMDBServer\deploy\axis2\WEB-INF**, open the file **web.xml**.
2. Add the following **servlet init** parameter to AxisServlet (paste these four lines after line 28):

   ```
   <init-param>
   <param-name>axis2.find.context</param-name>
   <param-value>false</param-value>
   </init-param>
   ```

   This setting prevents Axis2 from trying to calculate the context root and tells it to look for it explicitly in **axis2.xml**.
3. Under **UCMDBServer\deploy\axis2\WEB-INF\conf**, open the file **axis2.xml**.

4. At line 58, remove comments from the parameter **contextRoot**, and edit it as follows:

   ```
   <parameter name="contextRoot" locked="false">test/axis2</parameter>
   ```

   (where **test** is the new root context in **cmdb.xml**).

   Note: There is no slash at the beginning of **test/axis2**.

UCMDB Web Service API Use Cases

The following use cases assume two systems:

- HP Universal CMDB server
- A third-party system that contains a repository of configuration items

This section includes the following topics:

- "Populating the CMDB" below
- "Querying the CMDB" on the next page
- "Querying the Class Model" on the next page
- "Analyzing Change Impact" on the next page

Populating the CMDB

Use cases:

- A third-party asset management updates the CMDB with information available only in asset management
- A number of third-party systems populate the CMDB to create a central CMDB that can track changes and perform impact analysis
- A third-party system creates Configuration Items and Relations according to third-party business logic to leverage the CMDB query capabilities

Querying the CMDB

Use cases:

- A third-party system gets the Configuration Items and Relations that represent the SAP system by getting the results of the SAP TQL
- A third-party system gets the list of Oracle servers that have been added or changed in the last five hours
- A third-party system gets the list of servers whose host name contains the substring *lab*
- A third-party system finds the elements related to a given CI by getting its neighbors

Querying the Class Model

Use cases:

- A third-party system enables users to specify the set of data to be retrieved from the CMDB. A user interface can be built over the class model to show users the possible properties and prompt them for required data. The user can then choose the information to be retrieved.
- A third-party system explores the class model when the user cannot access the UCMDB user interface.

Analyzing Change Impact

Use case:

A third-party system outputs a list of the business services that could be impacted by a change on a specified host.

Examples

See the following code samples:

- The Example Base Class
- Query Example
- Update Example

- Class Model Example
- Impact Analysis Example

These files are located in the following directory:

\\<UCMDB root directory>\hp\UCMDB\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\WebServiceAPI_Samples

Chapter 12: Data Flow Management Java API

This chapter includes:

Using the Data Flow Management Java API

Note: Use this chapter in conjunction with the DFM API Javadoc, available in the online Documentation Library.

This chapter explains how third-party or custom tools can use the HP Data Flow Management Java API to manage Data Flow.The API provides methods to:

- **Manage credentials**. View, add, update, and remove.
- **Manage jobs**. View status, activate, and deactivate.
- **Manage probe ranges**. View, add, and update.
- **Manage triggers**. Add or remove a trigger CI, and add, remove, or disable a trigger TQL.
- **View general data**. Data on domains and probes.

The following services are available in the Discovery Services package:

- **DDMConfigurationService.** Services to configure the Data Flow Probes, clusters, IP Ranges, and Credentials.The Universal Discovery server can be configured with an XML file or via the Data Flow Probe.
- **DDMManagementService.** Services to analyze and view the progress, results, and errors of the Universal Discovery run.
- **DDMSoftwareSignatureService.** Services to define software items to be discovered by the Data Flow Probe components. The definitions are system-wide. If more than one Data Flow probe component is defined, the definitions apply to all of them.
- **DDMZoneService.** Services to manage zone-based discovery.

In addition to these services, there are Data Flow Management client APIs, which are used in creating Jython adapters. For details, see "Developing Jython Adapters" on page 41.

Permissions

The administrator provides login credentials for connecting with the API. The API client needs the user name and password of an integration user defined in the CMDB. These users do not represent human users of the CMDB, but rather applications that connect to the CMDB.

In addition, the user must have the **Access to SDK** general action permission in order to log in.

> **Caution:** The API client can also work with regular users as long as they have API authentication permission. However, this option is not recommended.

For details, see "Create an Integration User" on page 372.

Chapter 13: Data Flow Management Web Service API

This chapter includes:

Data Flow Management Web Service API Overview

This chapter explains how third-party or custom tools can use the HP Data Flow Management Web Service API to manage Data Flow.

The HP Data Flow Management Web Service API is used to integrate applications with HP Universal CMDB. The API provides methods to:

- **Manage credentials**. View, add, update, and remove.
- **Manage jobs**. View status, activate, and deactivate.
- **Manage probe ranges**. View, add, and update.
- **Manage triggers**. Add or remove a trigger CI, and add, remove, or disable a trigger TQL.
- **View general data**. Data on domains and probes.

Users of the HP Data Flow Management Web Service should be familiar with:

- The SOAP specification
- An object-oriented programming language such as C++, C# or Java
- HP Universal CMDB
- Data Flow Management

Note:

- A user must have the **Run Legacy API** general action permission in order to log in.
- The logged-in user must have the **Run Discovery and Integrations** general action permission to access any of the methods.

For full documentation on the available operations, see *HP Universal Discovery Schema Reference*. These files are located in the following folder:

<UCMDB root directory>\UCMDBServer\deploy\ucmdb-docs\docs\eng\APIs\DDM_Schema\webframe.html

Conventions

This chapter uses the following conventions:

- This style `Element` indicates that an item is an entity in the database or an element defined in the schema, including structures passed to or returned by methods. Plain text indicates that the item is being discussed in a general context.
- Data Flow Management elements and method arguments are spelled in the case in which they are specified in the schema. This usually means that a class name or generic reference to an instance of the class is capitalized. An element or argument to a method is not capitalized. For example, a `credential` is an element of type `Credential` passed to a method.

Call the HP Data Flow Management Web Service

The HP Data Flow Management Web Service API enables calling server-side methods using standard SOAP programming techniques. If the statement cannot be parsed or if there is a problem invoking the method, the API methods throw a `SoapFault` exception. When a `SoapFault` exception is thrown, the service populates one or more of the error message, error code, and exception message fields. If there is no error, the results of the invocation are returned.

To call the service, use:

- `Protocol: http or https (depending on server configuration)`
- `URL: <UCMDB server>:8080/axis2/services/DiscoveryService`

- `Default password: "admin"`
- `Default username: "admin"`

SOAP programmers can access the WSDL at:

- `axis2/services/DiscoveryService?wsdl`

Data Flow Management Methods and Data Structures

This section lists the Data Flow Management Web Service API methods and data structures, and provides a brief summary of their uses. For full documentation of the request and response for each operation, see *HP Universal Discovery Schema Reference*.

This section includes the following topics:

Data Structures

These are some of the data structures used in the Data Flow Management Web Service API.

CIProperties

`CIProperties` is a collection of collections. Each collection contains properties of a different data type. For example, there can be a `dateProps` collection, a `strListProps` collection, an `xmlProps` collection, and so on.

Each type collection contains individual properties of the given type. The names of these properties elements is the same as the container, but in singular. For example, `dateProps` contains `dateProp` elements. Each property is a name-value pair.

See CIProperties in the *HP Universal Discovery Schema Reference*.

IPList

A list of `IP` elements, each of which contains an IPv4 or IPv6 Address.

See IPList in the *HP Universal Discovery Schema Reference*.

IPRange

An `IPRange` has two elements, `Start` and `End`. Each element contains an `Address` element, which is an IPv4 or IPv6 Address.

See IPRange in the *HP Universal Discovery Schema Reference*.

Scope

Two `IPRanges`. `Exclude` is a collection of `IPRanges` to exclude from the job. `Include` is a collection of `IPRanges` to include in the job.

See Scope in the *HP Universal Discovery Schema Reference*

Managing Discovery Job Methods

activateJob

Activates the specified job.

See "Code Sample" on page 434.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	The name of the job.

deactivateJob

Deactivates the specified job.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	The name of the job.

dispatchAdHocJob

Dispatches a job on the probe ad hoc. The job must be active and contain the specified trigger CI.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	The name of the job.
CIID	The ID of the trigger CI.
ProbeName	The name of the probe.
Timeout	In milliseconds

getDiscoveryJobsNames

Returns the list of job names.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.

Output

Parameter	Comment
strList	The list of job names.

isJobActive

Checks whether the job is active.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	The name of the job to check.

Output

Parameter	Comment
JobState	True if the job is active.

Managing Trigger Methods

addTriggerCI

Adds a new trigger CI to the specified job.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	The name of the job.
CIID	The ID of the trigger CI.

addTriggerTQL

Adds a new trigger TQL to the specified job.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	The name of the job.
TqlName	The name of the TQL to add.

disableTriggerTQL

Prevents the TQL from triggering the job, but does not permanently remove it from the list of queries that trigger the job.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	The name of the job.

removeTriggerCI

Removes the specified CI from the list of CIs that trigger the job.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	The job name.
CIID	The ID of the trigger CI.

removeTriggerTQL

Removes the specified TQL from the list of queries that trigger the job.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
JobName	Collection of job names to check.
CIID	The ID of the TQL to remove.

setTriggerTQLProbesLimit

Restrict the probes in which the TQL is active in the job to the specified list.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.

Parameter	Comment
JobName	The name of the job.
tqlName	The TQL name.
probesLimit	The list of probes for which the TQL is active.

Domain and Probe Data Methods

getDomainType

Returns the domain type.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The name of the domain.

Output

Parameter	Comment
domainType	The domain type.

getDomainsNames

Returns the names of the current domains.

See "Code Sample" on page 434.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.

Output

Parameter	Comment
domainNames	The list of domain names.

getProbeIPs

Returns the IP addresses of the specified probe.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain to check.
probeName	The name of the probe used on that domain.

Output

Parameter	Comment
probeIPs	The "IPList" of the addresses in the probe.

getProbesNames

Returns the names of the probes in the specified domain.

See "Code Sample" on page 434.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain to check.

Output

Parameter	Comment
probesName	The list of probes on the domain.

getProbeScope

Returns the scope definition of the specified probe.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain to check.
probeName	The name of the probe.

Output

Parameter	Comment
probeScope	The "Scope" of the probe.

isProbeConnected

Checks whether the specified probe is connected.

See "Code Sample" on page 434.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain to check.
probeName	The probe to check

Output

Parameter	Comment
isConnected	True if the probe is connected.

updateProbeScope

Sets the scope of the specified probe, overriding the existing scope.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.

Parameter	Comment
domainName	The domain.
probeName	The probe to update.
newScope	The "Scope" to set for the probe.

Credentials Data Methods

addCredentialsEntry

Adds a credentials entry to the specified protocol for the specified domain.

See "Code Sample" on page 434.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain to update.
protocolName	The name of the protocol.
credentialsEntryParameters	The "CIProperties" collection of the new credentials.

Output

Parameter	Comment
credentialsEntryID	The CI ID of the new credential entry.

getCredentialsEntriesIDs

Returns the IDs of the credentials defined for the specified protocol.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain for which to get the credentials.
protocolName	The name of a protocol used on that domain.

Output

Parameter	Comment
credentialsEntryIDs	The list of credential IDs for the protocol on the domain.

getCredentialsEntry

Returns the credentials defined for the specified protocol. Encrypted attributes are returned empty.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain for which to get the credentials.
protocolName	The name of a protocol used on that domain.
credentialsEntryID	The credential ID to get.

Output

Parameter	Comment
credentialsEntryParameters	The "CIProperties" collection of the credentials.

removeCredentialsEntry

Removes the specified credentials from the protocol.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain.
protocolName	The name of a protocol used on the domain.
credentialsEntryID	The ID of the credential to remove.

updateCredentialsEntry

Sets new values for properties of the specified credentials entry.

The existing properties are deleted and these properties are set. Any property whose value is not set in this call is left undefined.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
domainName	The domain in which to update credentials.
protocolName	The name of a protocol used on the domain.
credentialsEntryID	The ID of the credentials to update.
credentialsEntryParameters	The "CIProperties" collection to set as properties for the credentials.

Data Refresh Methods

rediscoverCIs

Locates the triggers that discovered the specified CI objects and reruns those triggers. **rediscoverCIs** runs asynchronously. Call **checkDiscoveryProgress** to determine when the rediscovery is complete.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
CmdbIDs	Collection of IDs of the objects to rediscover.

Output

Parameter	Comment
isSucceed	True if the CIs rediscovery succeeded.

checkDiscoveryProgress

Returns the progress of the most recent **rediscoverCIs** call on the specified IDs. The response is a value between 0-1. When the response is 1, the **rediscoverCIs** call has completed.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
CmdbIDs	Collection of IDs of the objects in the rediscover call to track.

Output

Parameter	Comment
progress	A completed job has a progress of 1. Jobs that have not completed have a fraction less than 1.

rediscoverViewCIs

Locates the triggers that created the data to populate the specified view, and reruns those triggers. **rediscoverViewCIs** runs asynchronously. Call **checkViewDiscoveryProgress** to determine when the rediscovery is complete.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
viewName	The views to check.

Output

Parameter	Comment
isSucceed	True if CIs rediscovery succeeded.

checkViewDiscoveryProgress

Returns the progress of the most recent **rediscoverViewCIs** call on the specified view. The response is a value from 0 to 1. When the response is 1, the **rediscoverCIs** call has completed.

Input

Parameter	Comment
cmdbContext	For details, see "CmdbContext" on page 391.
viewName	The collection of views to check.

Output

Parameter	Comment
progress	A completed job has a progress of 1. Jobs that have not completed have a fraction less than 1.

Code Sample

```
import java.net.URL;
import org.apache.axis2.transport.http.HTTPConstants;
import org.apache.axis2.transport.http.HttpTransportProperties;
import com.hp.ucmdb.generated.params.discovery.*;
import com.hp.ucmdb.generated.services.*;
import com.hp.ucmdb.generated.types.*;
public class test {
    static final String HOST_NAME = "<my_hostname>";
    static final int PORT = 8080;
    private static final String PROTOCOL = "http";
    private static final String FILE = "/axis2/services/DiscoveryService";

    private static final String PASSWORD = "<my_password>";
    private static final String USERNAME = "<my_username>";

    private static CmdbContext cmdbContext = new CmdbContext("ws tests");

    public static void main(String[] args) throws Exception {
        // Get the stub object
        DiscoveryService discoveryService = getDiscoveryService();

        // Activate Job
        discoveryService.activateJob(new ActivateJobRequest(
            "Range IPs by ICMP", cmdbContext));

        // Get domain & probes info
        getProbesInfo(discoveryService);
        // Add credentilas entry for ntcmd protcol
        addNTCMDCredentialsEntry();
    }

    public static void addNTCMDCredentialsEntry() throws Exception {
        DiscoveryService discoveryService = getDiscoveryService();

        // Get domain name
        StrList domains =
            discoveryService.getDomainsNames(
                new GetDomainsNamesRequest(cmdbContext)).
                getDomainNames();
```

```
        if (domains.sizeStrValueList() == 0) {
            System.out.println("No domains were found, can't create credentials");
            return;
        }
        String domainName = domains.getStrValue(0);
        // Create propeties with one byte param
        CIProperties newCredsProperties = new CIProperties();

        // Add password property - this is of type bytes
        newCredsProperties.setBytesProps(new BytesProps());
        setPasswordProperty(newCredsProperties);

        // Add user & domain properties - these are of type string
        newCredsProperties.setStrProps(new StrProps());
        setStringProperties("protocol_username", "test user", newCredsProperties);
        setStringProperties("ntadminprotocol_ntdomain",
            "test doamin", newCredsProperties);

        // Add new credentials entry
        discoveryService.addCredentialsEntry(
            new AddCredentialsEntryRequest(domainName,
            "ntadminprotocol",  newCredsProperties, cmdbContext));
        System.out.println("new credentials craeted for domain: " + domainName + "
in ntcmd protocol");
    }

    private static void setPasswordProperty(CIProperties newCredsProperties) {
        BytesProp bProp = new BytesProp();
        bProp.setName("protocol_password");
        bProp.setValue(new byte[] {101,103,102,104});
        newCredsProperties.getBytesProps().addBytesProp(bProp);
    }

    private static void setStringProperties(String propertyName, String value,
CIProperties newCredsProperties) {
        StrProp strProp = new StrProp();
        strProp.setName(propertyName);
        strProp.setValue(value);
        newCredsProperties.getStrProps().addStrProp(strProp);
    }


    private static void getProbesInfo(DiscoveryService discoveryService) throws
Exception {
        GetDomainsNamesResponse result = discoveryService.getDomainsNames(new
GetDomainsNamesRequest(cmdbContext ));
        // Go over all the domains
        if (result.getDomainNames().sizeStrValueList() > 0) {
            String domainName =
            result.getDomainNames().getStrValue(0);
```

```
            GetProbesNamesResponse probesResult =
                discoveryService.getProbesNames(
                    new GetProbesNamesRequest(domainName, cmdbContext));
            // Go over all the probes
            for (int i=0; i<probesResult.getProbesNames().sizeStrValueList(); i++)
{
                String probeName = probesResult.getProbesNames().getStrValue(i);
                // Check if connected
                IsProbeConnectedResponce connectedRequest =
                    discoveryService.isProbeConnected(
                        new IsProbeConnectedRequest(
                        domainName, probeName, cmdbContext));
                Boolean isConnected = connectedRequest.getIsConnected();
                // Do something ...
                System.out.println("probe " + probeName + " isconnect=" +
isConnected);
            }
        }
    }

    private static DiscoveryService getDiscoveryService() throws Exception {
        DiscoveryService discoveryService = null;
        try {
            // Create service
            URL url = new URL(PROTOCOL,HOST_NAME,PORT, FILE);
            DiscoveryServiceStub serviceStub =
                new DiscoveryServiceStub(url.toString());

            // Authenticate info
            HttpTransportProperties.Authenticator auth =
                new HttpTransportProperties.Authenticator();
            auth.setUsername(USERNAME);
            auth.setPassword(PASSWORD);
            serviceStub._getServiceClient().getOptions().setProperty(
                HTTPConstants.AUTHENTICATE,auth);

            discoveryService = serviceStub;
        } catch (Exception e) {
            throw new Exception("cannot create a connection to service ", e);
        }
        return discoveryService;
    }
}
```

Adding Credentials Example

```
import java.net.URL;
import org.apache.axis2.transport.http.HTTPConstants;
import org.apache.axis2.transport.http.HttpTransportProperties;
import com.hp.ucmdb.generated.params.discovery.*;
import com.hp.ucmdb.generated.services.DiscoveryService;
import com.hp.ucmdb.generated.services.DiscoveryServiceStub;
import com.hp.ucmdb.generated.types.BytesProp;
import com.hp.ucmdb.generated.types.BytesProps;
import com.hp.ucmdb.generated.types.CIProperties;
import com.hp.ucmdb.generated.types.CmdbContext;
import com.hp.ucmdb.generated.types.StrList;
import com.hp.ucmdb.generated.types.StrProp;
import com.hp.ucmdb.generated.types.StrProps;

public class test {
    static final String HOST_NAME = "hostname";
    static final int PORT = 8080;
    private static final String PROTOCOL = "http";
    private static final String FILE = "/axis2/services/DiscoveryService";

    private static final String PASSWORD = "admin";
    private static final String USERNAME = "admin";

    private static CmdbContext cmdbContext = new CmdbContext("ws tests");

    public static void main(String[] args) throws Exception {
        // Get the stub object
        DiscoveryService discoveryService = getDiscoveryService();

        // Activate Job
        discoveryService.activateJob(new ActivateJobRequest("Range IPs by ICMP",
cmdbContext));

        // Get domain & probes info
        getProbesInfo(discoveryService);
        // Add credentilas entry for ntcmd protcol
        addNTCMDCredentialsEntry();
    }

    public static void addNTCMDCredentialsEntry() throws Exception {
        DiscoveryService discoveryService = getDiscoveryService();

        // Get domain name
        StrList domains =
            discoveryService.getDomainsNames(new GetDomainsNamesRequest
(cmdbContext)).getDomainNames();
```

```
        if (domains.sizeStrValueList() == 0) {
            System.out.println("No domains were found, can't create credentials");
            return;
        }
        String domainName = domains.getStrValue(0);
        // Create propeties with one byte param
        CIProperties newCredsProperties = new CIProperties();

        // Add password property - this is of type bytes
        newCredsProperties.setBytesProps(new BytesProps());
        setPasswordProperty(newCredsProperties);

        // Add user & domain properties - these are of type string
        newCredsProperties.setStrProps(new StrProps());
        setStringProperties("protocol_username", "test user", newCredsProperties);
        setStringProperties("ntadminprotocol_ntdomain", "test doamin",
newCredsProperties);

        // Add new credentials entry
        discoveryService.addCredentialsEntry(new AddCredentialsEntryRequest
(domainName,  "ntadminprotocol",  newCredsProperties, cmdbContext));
        System.out.println("new credentials craeted for domain: " + domainName + "
in ntcmd protocol");
    }

    private static void setPasswordProperty(CIProperties newCredsProperties) {
        BytesProp bProp = new BytesProp();
        bProp.setName("protocol_password");
        bProp.setValue(new byte[] {101,103,102,104});
        newCredsProperties.getBytesProps().addBytesProp(bProp);
    }

    private static void setStringProperties(String propertyName, String value,
CIProperties newCredsProperties) {
        StrProp strProp = new StrProp();
        strProp.setName(propertyName);
        strProp.setValue(value);
        newCredsProperties.getStrProps().addStrProp(strProp);
    }

    private static void getProbesInfo(DiscoveryService discoveryService) throws
Exception {
        GetDomainsNamesResponse result = discoveryService.getDomainsNames(new
GetDomainsNamesRequest(cmdbContext ));
        // Go over all the domains
        if (result.getDomainNames().sizeStrValueList() > 0) {
            String domainName = result.getDomainNames().getStrValue(0);
            GetProbesNamesResponse probesResult =
                discoveryService.getProbesNames(new GetProbesNamesRequest
(domainName, cmdbContext));
```

```
            // Go over all the probes
            for (int i=0; i<probesResult.getProbesNames().sizeStrValueList(); i++)
{
                String probeName = probesResult.getProbesNames().getStrValue(i);
                // Check if connected
                IsProbeConnectedResponce connectedRequest =
                    discoveryService.isProbeConnected(new IsProbeConnectedRequest
(domainName, probeName, cmdbContext));
                Boolean isConnected = connectedRequest.getIsConnected();
                // Do something ...
                System.out.println("probe " + probeName + " isconnect=" +
isConnected);
            }
        }
    }

    private static DiscoveryService getDiscoveryService() throws Exception {
        DiscoveryService discoveryService = null;
        try {
            // Create service
            URL url = new URL(PROTOCOL,HOST_NAME,PORT, FILE);
            DiscoveryServiceStub serviceStub = new DiscoveryServiceStub
(url.toString());

            // Authenticate info
            HttpTransportProperties.Authenticator auth = new
HttpTransportProperties.Authenticator();
            auth.setUsername(USERNAME);
            auth.setPassword(PASSWORD);
            serviceStub._getServiceClient().getOptions().setProperty
(HTTPConstants.AUTHENTICATE,auth);

            discoveryService = serviceStub;
        } catch (Exception e) {
            throw new Exception("cannot create a connection to service ", e);
        }
        return discoveryService;
    }
} // End class
```

Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on Developer Reference Guide (Universal CMDB 10.20)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to cms-doc@hp.com.

We appreciate your feedback!