Hewlett Packard
Enterprise

HPE File System Connector

Software Version: 11.4

Administration Guide

Document Release Date: June 2017

Software Release Date: June 2017

Legal notices

Warranty

The only warranties for Hewlett Packard Enterprise Development LP 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. HPE 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 HPE 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

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.

This product includes an interface of the 'zlib' general purpose compression library, which is Copyright © 1995-2002 Jean-loup Gailly and Mark Adler.

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 software updates, go to https://downloads.autonomy.com/productDownloads.jsp.

To verify that you are using the most recent edition of a document, go to https://softwaresupport.hpe.com/group/softwaresupport/search-result?doctype=online help.

This site requires that you register for an HPE Passport and sign in. To register for an HPE Passport ID, go to https://hpp12.passport.hpe.com/hppcf/login.do.

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

Support

Visit the HPE Software Support Online web site at https://softwaresupport.hpe.com.

This web site provides contact information and details about the products, services, and support that HPE Software offers.

HPE 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 web site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Access product documentation
- Manage support contracts
- Look up HPE 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 HPE Passport user and sign in. Many also require a support contract.

To register for an HPE Passport ID, go to https://hpp12.passport.hpe.com/hppcf/login.do.

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

To check for recent software updates, go to https://downloads.autonomy.com/productDownloads.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 or read the online Help.

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

Chapter 1: Introduction

This section provides an overview of the HPE File System Connector.

HPE File System Connector

HPE File System Connector is an IDOL connector that retrieves information from file systems on local or network machines.

The HPE File System Connector can:

- Keep IDOL up-to-date with the information in the file system.
- Collect documents from the file system and either write them to disk or send them to IDOL.
- Insert documents into the file system.
- Delete documents from the file system.

Related Topics

- Connector Framework Server, on page 15
- HPE's IDOL Platform, on page 17

Supported Actions

The File System Connector (CFS) supports the following actions:

Action	Supported	Further Information
Synchronize	✓	
Synchronize (identifiers)	✓	The connector's `Synchronize` action supports the optional `identifiers` parameter, which accepts a comma-separated list of document identifiers. If you set this parameter the action synchronizes the specified documents, regardless of whether they have changed.
Synchronize Groups	✗	The `SynchronizeGroups` action is not required because OmniGroupServer can retrieve group information through its LDAP module.

Collect	✓	
Identifiers	✓	
Insert	✓	Insert Files into the File System, on page 53
Delete/Remove	✓	
Hold/ReleaseHold	✗	
Update	✓	Update File Metadata, on page 54
Stub	✓	
GetURI	✓	
View	✓	

Mapped Security

The HPE File System Connector supports mapped security.

The connector can retrieve:

- NT ACLs (the connector must run on Windows).
- Netware ACLs (the connector must run on Windows).
- POSIX ACLs (the connector must run on Linux and the file system must support them).

Display Online Help

You can display the HPE File System Connector Reference by sending an action from your web browser. The HPE File System Connector Reference describes the actions and configuration parameters that you can use with HPE File System Connector.

For HPE File System Connector to display help, the help data file (`help.dat`) must be available in the installation folder.

To display help for HPE File System Connector

1. Start HPE File System Connector.
2. Send the following action from your web browser:

   `http://host:port/action=Help`

   where:

host is the IP address or name of the machine on which HPE File System Connector is installed.

port is the ACI port by which you send actions to HPE File System Connector (set by the `Port` parameter in the `[Server]` section of the configuration file).

For example:

```
http://12.3.4.56:9000/action=help
```

OEM Certification

HPE File System Connector works in OEM licensed environments.

Connector Framework Server

Connector Framework Server (CFS) processes the information that is retrieved by connectors, and then indexes the information into IDOL.

A single CFS can process information from any number of connectors. For example, a CFS might process files retrieved by a File System Connector, web pages retrieved by a Web Connector, and e-mail messages retrieved by an Exchange Connector.

To use the HPE File System Connector to index documents into IDOL Server, you must have a CFS. When you install the HPE File System Connector, you can choose to install a CFS or point the connector to an existing CFS.

For information about how to configure and use Connector Framework Server, refer to the *Connector Framework Server Administration Guide*.

Filter Documents and Extract Subfiles

The documents that are sent by connectors to CFS contain only metadata extracted from the repository, such as the location of a file or record that the connector has retrieved. CFS uses KeyView to extract the file content and file specific metadata from over 1000 different file types, and adds this information to the documents. This allows IDOL to extract meaning from the information contained in the repository, without needing to process the information in its native format.

CFS also uses KeyView to extract and process sub-files. Sub-files are files that are contained within other files. For example, an e-mail message might contain attachments that you want to index, or a Microsoft Word document might contain embedded objects.

Manipulate and Enrich Documents

CFS provides features to manipulate and enrich documents before they are indexed into IDOL. For example, you can:

- add additional fields to a document.
- divide long documents into multiple sections.
- run tasks including Eduction, Optical Character Recognition, or Face Recognition, and add the information that is obtained to the document.
- run a custom Lua script to modify a document.

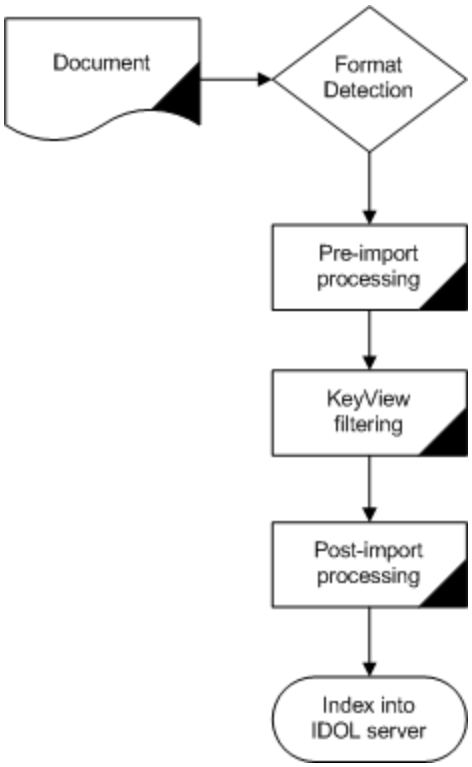
Index Documents

After CFS finishes processing documents, it automatically indexes them into one or more indexes. CFS can index documents into:

- **IDOL Server** (or send them to a *Distributed Index Handler*, so that they can be distributed across multiple IDOL servers).
- **Haven OnDemand**.
- **Vertica**.

Import Process

This section describes the import process for new files that are added to IDOL through CFS.



1. Connectors aggregate documents from repositories and send the files to CFS. A single CFS can process documents from multiple connectors. For example, CFS might receive HTML files from HTTP Connectors, e-mail messages from Exchange Connector, and database records from ODBC Connector.
2. CFS runs pre-import tasks. Pre-Import tasks occur before document content and file-specific metadata is extracted by KeyView.
3. KeyView filters the document content, and extracts sub-files.
4. CFS runs post-import tasks. Post-Import tasks occur after KeyView has extracted document content and file-specific metadata.
5. The data is indexed into IDOL.

HPE's IDOL Platform

At the core of HPE File System Connector is HPE's *Intelligent Data Operating Layer* (IDOL).

IDOL gathers and processes unstructured, semi-structured, and structured information in any format from multiple repositories using IDOL connectors and a global relational index. It can automatically form a contextual understanding of the information in real time, linking disparate data sources together based on the concepts contained within them. For example, IDOL can automatically link concepts contained in an email message to a recorded phone conversation, that can be associated with a stock trade. This information is then imported into a format that is easily searchable, adding advanced retrieval, collaboration, and personalization to an application that integrates the technology.

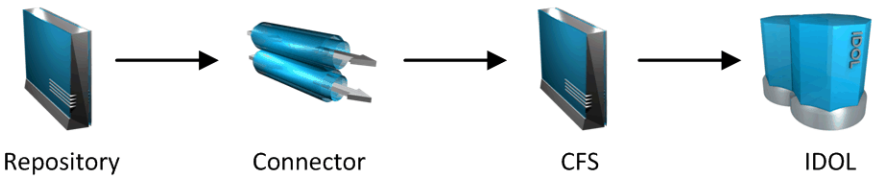
For more information on IDOL, see the *IDOL Getting Started Guide*.

System Architecture

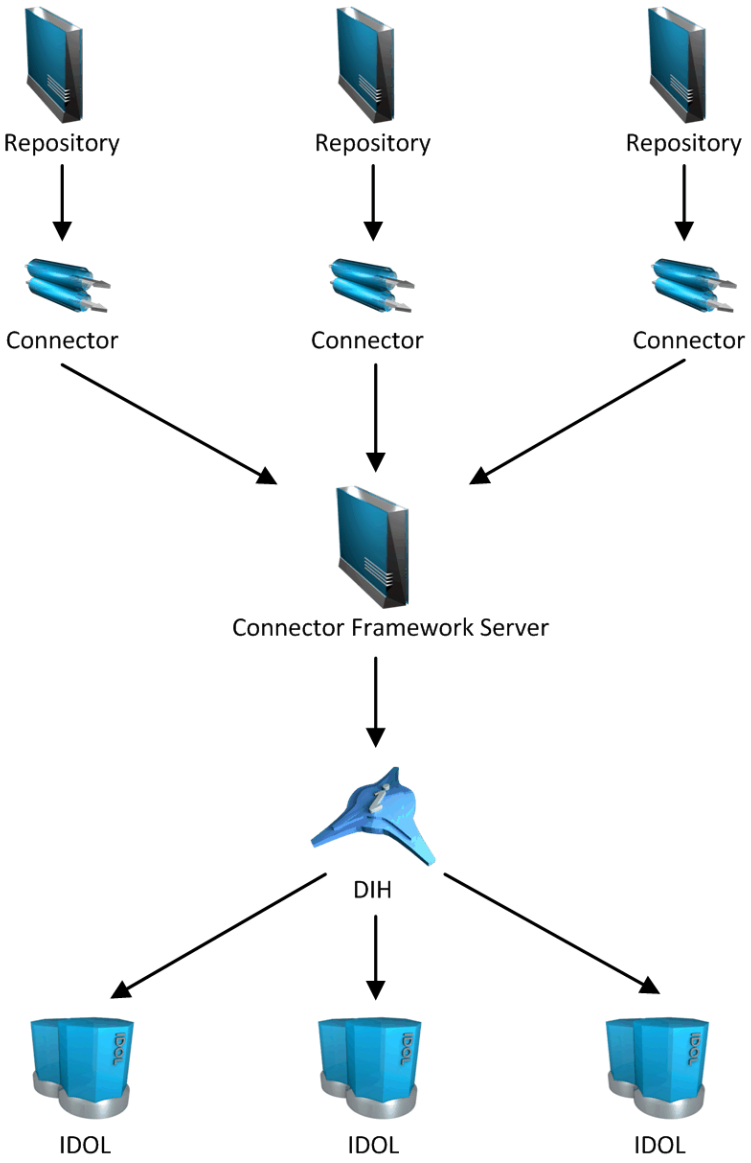
An IDOL infrastructure can include the following components:

- **Connectors**. Connectors aggregate data from repositories and send the data to CFS.
- **Connector Framework Server (CFS)**. Connector Framework Server (CFS) processes and enriches the information that is retrieved by connectors.
- **IDOL Server**. IDOL stores and processes the information that is indexed into it by CFS.
- **Distributed Index Handler (DIH)**. The Distributed Index Handler distributes data across multiple IDOL servers. Using multiple IDOL servers can increase the availability and scalability of the system.
- **License Server**. The License server licenses multiple products.

These components can be installed in many different configurations. The simplest installation consists of a single connector, a single CFS, and a single IDOL Server.



A more complex configuration might include more than one connector, or use a Distributed Index Handler (DIH) to index content across multiple IDOL Servers.



Related Documentation

The following documents provide further information related to HPE File System Connector.

- *HPE File System Connector Reference*

  The *HPE File System Connector Reference* describes the configuration parameters and actions that you can use with the HPE File System Connector.
- *Connector Framework Server Administration Guide*

Connector Framework Server (CFS) processes documents that are retrieved by connectors. CFS then indexes the documents into IDOL Server, Haven OnDemand, or Vertica. The *Connector Framework Server Administration Guide* describes how to configure and use CFS.

- *IDOL Getting Started Guide*

  The *IDOL Getting Started Guide* provides an introduction to IDOL. It describes the system architecture, how to install IDOL components, and introduces indexing and security.

- *IDOL Server Administration Guide*

  The *IDOL Server Administration Guide* describes the operations that IDOL server can perform with detailed descriptions of how to set them up.

- *IDOL Document Security Administration Guide*

  The *IDOL Document Security Administration Guide* describes how to protect the information that you index into IDOL Server.

- *License Server Administration Guide*

  This guide describes how to use a License Server to license multiple services.

Chapter 2: Install HPE File System Connector

This section describes how to install the HPE File System Connector.

System Requirements

HPE File System Connector can be installed as part of a larger system that includes an IDOL Server and an interface for the information stored in IDOL Server. To maximize performance, HPE recommends that you install IDOL Server and the connector on different machines.

For information about the minimum system requirements required to run IDOL components, including HPE File System Connector, refer to the *IDOL Getting Started Guide*.

The connector must be installed on the machine that you want to ingest files from, or on a machine that has access to those files through standard paths.

Netware File Systems

To retrieve files from a Netware file system, the connector must be installed on a machine running Windows. You must also install the Novell client on the connector machine. This allows the connector to access the Netware server.

OmniGroupServer

To extract group information, you must also install OmniGroupServer.

Permissions

This section describes the permissions that must be granted to the connector.

Action	Permissions Required
Synchronize	The connector must be able to: • read the entire directory structure below the folder specified by `DirectoryPathCSVs`. • list files so that it can find the files you want to retrieve. The user account used to run the connector must have read access for the files and

	folders that you want to retrieve. CFS also requires the same permission. **TIP:** Starting the connector by running the executable file uses your user account to run the connector. Unless you have the permissions described above you might need to run the connector using a different user account. To run the connector using a different user account, log on as that user. On Windows platforms, the HPE File System Connector service uses the Local System account to run the connector. To retrieve files from a network share, modify the service so that it uses an account that has the necessary permissions. On Windows, extracting security information requires the "Read permissions" permission.
Collect	The same permissions as the synchronize action. The connector must also have write-access to the collect destination folder.
Insert	The user account used to run the connector must have write access to the path where the files are inserted.
Delete/Remove	The user account used to run the connector must have permission to delete the files.
View	The user account used to run the connector must have read access for the files. Unless the connector communicates directly with the View server, the connector also requires access to the shared path for the IDOL View Server.

Install HPE File System Connector

The HPE File System Connector can be installed using the IDOL Server installer.

For information about installing the HPE File System Connector using this installer, refer to the *IDOL Getting Started Guide*.

Configure the License Server Host and Port

HPE File System Connector is licensed through HPE License Server. In the HPE File System Connector configuration file, specify the information required to connect to the License Server.

To specify the license server host and port

1. Open your configuration file in a text editor.
2. In the `[License]` section, modify the following parameters to point to your License Server.

   `LicenseServerHost` The host name or IP address of your License Server.

   `LicenseServerACIPort` The ACI port of your License Server.

   For example:

```
[License]
LicenseServerHost=licenses
LicenseServerACIPort=20000
```

3. Save and close the configuration file.

Chapter 3: Configure HPE File System Connector

This section describes how to configure the HPE File System Connector.

HPE File System Connector Configuration File

You can configure the HPE File System Connector by editing the configuration file. The configuration file is located in the connector's installation folder. You can modify the file with a text editor.

The parameters in the configuration file are divided into sections that represent connector functionality.

Some parameters can be set in more than one section of the configuration file. If a parameter is set in more than one section, the value of the parameter located in the most specific section overrides the value of the parameter defined in the other sections. For example, if a parameter can be set in "*TaskName* or `FetchTasks` or `Default`", the value in the `TaskName` section overrides the value in the `FetchTasks` section, which in turn overrides the value in the `Default` section. This means that you can set a default value for a parameter, and then override that value for specific tasks.

For information about the parameters that you can use to configure the HPE File System Connector, refer to the *HPE File System Connector Reference*.

Server Section

The `[Server]` section specifies the ACI port of the connector. It can also contain parameters that control the way the connector handles ACI requests.

Service Section

The `[Service]` section specifies the service port of the connector.

Actions Section

The `[Actions]` section contains configuration parameters that specify how the connector processes actions that are sent to the ACI port. For example, you can configure event handlers that run when an action starts, finishes, or encounters an error.

Logging Section

The `[Logging]` section contains configuration parameters that determine how messages are logged. You can use *log streams* to send different types of message to separate log files. The configuration file also contains a section to configure each of the log streams.

Connector Section

The `[Connector]` section contains parameters that control general connector behavior. For example, you can specify a schedule for the fetch tasks that you configure.

Default Section

The `[Default]` section is used to define default settings for configuration parameters. For example, you can specify default settings for the tasks in the `[FetchTasks]` section.

FetchTasks Section

The `[FetchTasks]` section lists the fetch tasks that you want to run. A *fetch task* is a task that retrieves data from a repository. Fetch tasks are usually run automatically by the connector, but you can also run a fetch task by sending an action to the connector's ACI port.

In this section, enter the total number of fetch tasks in the `Number` parameter and then list the tasks in consecutive order starting from 0 (zero). For example:

```
[FetchTasks]
Number=2
0=MyTask0
1=MyTask1
```

[*TaskName*] Section

The `[TaskName]` section contains configuration parameters that apply to a specific task. There must be a `[TaskName]` section for every task listed in the `[FetchTasks]` section.

Ingestion Section

The `[Ingestion]` section specifies where to send the data that is extracted by the connector.

You can send data to a Connector Framework Server, Haven OnDemand, or another connector. For more information about ingestion, see Ingestion, on page 78.

DistributedConnector Section

The `[DistributedConnector]` section configures the connector to operate with the Distributed Connector. The Distributed Connector is an ACI server that distributes actions (`synchronize`, `collect` and so on) between multiple connectors.

For more information about the Distributed Connector, refer to the *Distributed Connector Administration Guide*.

ViewServer Section

The `[ViewServer]` section contains parameters that allow the connector's *view* action to use a View Server. If necessary, the View Server converts files to HTML so that they can be viewed in a web browser.

License Section

The `[License]` section contains details about the License server (the server on which your license file is located).

Document Tracking Section

The `[DocumentTracking]` section contains parameters that enable the tracking of documents through import and indexing processes.

Related Topics

- Modify Configuration Parameter Values, below
- Customize Logging, on page 91

Modify Configuration Parameter Values

You modify HPE File System Connector configuration parameters by directly editing the parameters in the configuration file. When you set configuration parameter values, you must use UTF-8.

CAUTION:
You must stop and restart HPE File System Connector for new configuration settings to take effect.

This section describes how to enter parameter values in the configuration file.

Enter Boolean Values

The following settings for Boolean parameters are interchangeable:

```
TRUE = true = ON = on = Y = y = 1
```

```
FALSE = false = OFF = off = N = n = 0
```

Enter String Values

To enter a comma-separated list of strings when one of the strings contains a comma, you can indicate the start and the end of the string with quotation marks, for example:

ParameterName=**cat,dog,bird,"wing,beak",turtle**

Alternatively, you can escape the comma with a backslash:

ParameterName=**cat,dog,bird,wing\,beak,turtle**

If any string in a comma-separated list contains quotation marks, you must put this string into quotation marks and escape each quotation mark in the string by inserting a backslash before it. For example:

ParameterName=**"<font face=\"arial\" size=\"+1\"><b>","<p>"**

Here, quotation marks indicate the beginning and end of the string. All quotation marks that are contained in the string are escaped.

Include an External Configuration File

You can share configuration sections or parameters between ACI server configuration files. The following sections describe different ways to include content from an external configuration file.

You can include a configuration file in its entirety, specified configuration sections, or a single parameter.

When you include content from an external configuration file, the `GetConfig` and `ValidateConfig` actions operate on the combined configuration, after any external content is merged in.

In the procedures in the following sections, you can specify external configuration file locations by using absolute paths, relative paths, and network locations. For example:

```
../sharedconfig.cfg
K:\sharedconfig\sharedsettings.cfg
\\example.com\shared\idol.cfg
file://example.com/shared/idol.cfg
```

Relative paths are relative to the primary configuration file.

> **NOTE:**
> You can use nested inclusions, for example, you can refer to a shared configuration file that references a third file. However, the external configuration files must not refer back to your original configuration file. These circular references result in an error, and HPE File System Connector does not start.
>
> Similarly, you cannot use any of these methods to refer to a different section in your primary configuration file.

Include the Whole External Configuration File

This method allows you to import the whole external configuration file at a specified point in your configuration file.

To include the whole external configuration file

1. Open your configuration file in a text editor.
2. Find the place in the configuration file where you want to add the external configuration file.
3. On a new line, type a left angle bracket (<), followed by the path to and name of the external configuration file, in quotation marks (""). You can use relative paths and network locations. For example:

   ```
   < "K:\sharedconfig\sharedsettings.cfg"
   ```

4. Save and close the configuration file.

Include Sections of an External Configuration File

This method allows you to import one or more configuration sections from an external configuration file at a specified point in your configuration file. You can include a whole configuration section in this way, but the configuration section name in the external file must exactly match what you want to use in your file. If you want to use a configuration section from the external file with a different name, see Merge a Section from an External Configuration File, on the next page.

To include sections of an external configuration file

1. Open your configuration file in a text editor.
2. Find the place in the configuration file where you want to add the external configuration file section.
3. On a new line, type a left angle bracket (<), followed by the path to and name of the external configuration file, in quotation marks (""). You can use relative paths and network locations. After the configuration file name, add the configuration section name that you want to include. For example:

   ```
   < "K:\sharedconfig\extrasettings.cfg" [License]
   ```

   > **NOTE:**
   > You cannot include a section that already exists in your configuration file.

4. Save and close the configuration file.

Include a Parameter from an External Configuration File

This method allows you to import a parameter from an external configuration file at a specified point in your configuration file. You can include a section or a single parameter in this way, but the value in the external file must exactly match what you want to use in your file.

To include a parameter from an external configuration file

1. Open your configuration file in a text editor.
2. Find the place in the configuration file where you want to add the parameter from the external configuration file.
3. On a new line, type a left angle bracket (<), followed by the path to and name of the external configuration file, in quotation marks (""). You can use relative paths and network locations. After the configuration file name, add the name of the configuration section name that contains the parameter, followed by the parameter name. For example:

   ```
   < "license.cfg" [License] LicenseServerHost
   ```

   To specify a default value for the parameter, in case it does not exist in the external configuration file, specify the configuration section, parameter name, and then an equals sign (=) followed by the default value. For example:

   ```
   < "license.cfg" [License] LicenseServerHost=localhost
   ```

4. Save and close the configuration file.

Merge a Section from an External Configuration File

This method allows you to include a configuration section from an external configuration file as part of your HPE File System Connector configuration file. For example, you might want to specify a standard SSL configuration section in an external file and share it between several servers. You can use this method if the configuration section that you want to import has a different name to the one you want to use.

To merge a configuration section from an external configuration file

1. Open your configuration file in a text editor.
2. Find or create the configuration section that you want to include from an external file. For example:

   ```
   [SSLOptions1]
   ```

3. After the configuration section name, type a left angle bracket (<), followed by the path to and name of the external configuration file, in quotation marks (""). You can use relative paths and network locations. For example:

   ```
   [SSLOptions1] < "../sharedconfig/ssloptions.cfg"
   ```

   If the configuration section name in the external configuration file does not match the name that you want to use in your configuration file, specify the section to import after the configuration file name. For example:

```
[SSLOptions1] < "../sharedconfig/ssloptions.cfg" [SharedSSLOptions]
```

In this example, HPE File System Connector uses the values in the `[SharedSSLOptions]` section of the external configuration file as the values in the `[SSLOptions1]` section of the HPE File System Connector configuration file.

> **NOTE:**
> You can include additional configuration parameters in the section in your file. If these parameters also exist in the imported external configuration file, HPE File System Connector uses the values in the local configuration file. For example:
>
> ```
> [SSLOptions1] < "ssloptions.cfg" [SharedSSLOptions]
> SSLCACertificatesPath=C:\IDOL\HTTPConnector\CACERTS\
> ```

4. Save and close the configuration file.

Encrypt Passwords

HPE recommends that you encrypt all passwords that you enter into a configuration file.

Create a Key File

A key file is required to use AES encryption.

To create a new key file

1. Open a command-line window and change directory to the HPE File System Connector installation folder.
2. At the command line, type:

   ```
   autpassword -x -tAES -oKeyFile=./MyKeyFile.ky
   ```

   A new key file is created with the name `MyKeyFile.ky`

> **CAUTION:**
> To keep your passwords secure, you must protect the key file. Set the permissions on the key file so that only authorized users and processes can read it. HPE File System Connector must be able to read the key file to decrypt passwords, so do not move or rename it.

Encrypt a Password

The following procedure describes how to encrypt a password.

To encrypt a password

1. Open a command-line window and change directory to the HPE File System Connector installation folder.
2. At the command line, type:

`autpassword -e -t`*`EncryptionType`* `[-oKeyFile] [-c`*`FILE`* `-s`*`SECTION`* `-p`*`PARAMETER`*`] `*`PasswordString`*

where:

Option	Description
`-t` *`EncryptionType`*	The type of encryption to use: • **`Basic`** • **`AES`** For example: **`-tAES`** **NOTE:** AES is more secure than basic encryption.
`-o`*`KeyFile`*	AES encryption requires a key file. This option specifies the path and file name of a key file. The key file must contain 64 hexadecimal characters. For example: **`-oKeyFile=./key.ky`**
`-cFILE -sSECTION -pPARAMETER`	(Optional) You can use these options to write the password directly into a configuration file. You must specify all three options. • **`-c`**. The configuration file in which to write the encrypted password. • **`-s`**. The name of the section in the configuration file in which to write the password. • **`-p`**. The name of the parameter in which to write the encrypted password. For example: **`-c./Config.cfg -sMyTask -pPassword`**
`PasswordString`	The password to encrypt.

For example:

`autpassword -e -tBASIC `*`MyPassword`*

`autpassword -e -tAES -oKeyFile=./key.ky `*`MyPassword`*

`autpassword -e -tAES -oKeyFile=./key.ky -c./Config.cfg -sDefault -pPassword `*`MyPassword`*

The password is returned, or written to the configuration file.

Decrypt a Password

The following procedure describes how to decrypt a password.

To decrypt a password

1. Open a command-line window and change directory to the HPE File System Connector installation folder.

2. At the command line, type:

   `autpassword -d -t`*`EncryptionType`* `[-oKeyFile]` *`PasswordString`*

   where:

Option	Description
`-t` *`EncryptionType`*	The type of encryption: • **`Basic`** • **`AES`** For example: **`-tAES`**
`-oKeyFile`	AES encryption and decryption requires a key file. This option specifies the path and file name of the key file used to decrypt the password. For example: **`-oKeyFile=./key.ky`**
`PasswordString`	The password to decrypt.

   For example:

   ```
   autpassword -d -tBASIC 9t3M3t7awt/J8A
   ```

   ```
   autpassword -d -tAES -oKeyFile=./key.ky 9t3M3t7awt/J8A
   ```

   The password is returned in plain text.

Configure Client Authorization

You can configure HPE File System Connector to authorize different operations for different connections.

Authorization roles define a set of operations for a set of users. You define the operations by using the `StandardRoles` configuration parameter, or by explicitly defining a list of allowed actions in the `Actions` and `ServiceActions` parameters. You define the authorized users by using a client IP address, SSL identities, and GSS principals, depending on your security and system configuration.

For more information about the available parameters, see the *HPE File System Connector Reference*.

To configure authorization roles

1. Open your configuration file in a text editor.
2. Find the `[AuthorizationRoles]` section, or create one if it does not exist.
3. In the `[AuthorizationRoles]` section, list the user authorization roles that you want to create. For example:

   ```
   [AuthorizationRoles]
   0=AdminRole
   1=UserRole
   ```

4. Create a section for each authorization role that you listed. The section name must match the name that you set in the `[AuthorizationRoles]` list. For example:

```
[AdminRole]
```

5. In the section for each role, define the operations that you want the role to be able to perform. You can set `StandardRoles` to a list of appropriate values, or specify an explicit list of allowed actions by using `Actions` and `ServiceActions`. For example:

```
[AdminRole]
StandardRoles=Admin,Index,ServiceControl,ServiceStatus

[UserRole]
Actions=GetVersion
ServiceActions=GetStatus
```

> **NOTE:**
> The standard roles do not overlap. If you want a particular role to be able to perform all actions, you must include all the standard roles, or ensure that the clients, SSL identities, and so on, are assigned to all relevant roles.

6. In the section for each role, define the access permissions for the role, by setting `Clients`, `SSLIdentities`, and `GSSPrincipals`, as appropriate. If an incoming connection matches one of the allowed clients, principals, or SSL identities, the user has permission to perform the operations allowed by the role. For example:

```
[AdminRole]
StandardRoles=Admin,Index,ServiceControl,ServiceStatus
Clients=localhost
SSLIdentities=admin.example.com
```

7. Save and close the configuration file.
8. Restart HPE File System Connector for your changes to take effect.

Register with a Distributed Connector

To receive actions from a Distributed Connector, a connector must register with the Distributed Connector and join a *connector group*. A connector group is a group of similar connectors. The connectors in a group must be of the same type (for example, all HTTP Connectors), and must be able to access the same repository.

To configure a connector to register with a Distributed Connector, follow these steps. For more information about the Distributed Connector, refer to the *Distributed Connector Administration Guide*.

To register with a Distributed Connector

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. In the `[DistributedConnector]` section, set the following parameters:

`RegisterConnector`	(Required) To register with a Distributed Connector, set this parameter to **`true`**.

`HostN`	(Required) The host name or IP address of the Distributed Connector.
`PortN`	(Required) The ACI port of the Distributed Connector.
`DataPortN`	(Optional) The data port of the Distributed Connector.
`ConnectorGroup`	(Required) The name of the connector group to join. The value of this parameter is passed to the Distributed Connector.
`ConnectorPriority`	(Optional) The Distributed Connector can distribute actions to connectors based on a priority value. The lower the value assigned to `ConnectorPriority`, the higher the probability that an action is assigned to this connector, rather than other connectors in the same connector group.
`SharedPath`	(Optional) The location of a shared folder that is accessible to all of the connectors in the `ConnectorGroup`. This folder is used to store the connectors' datastore files, so that whichever connector in the group receives an action, it can access the information required to complete it. If you set the `DataPortN` parameter, the datastore file is streamed directly to the Distributed Connector, and this parameter is ignored.

4. Save and close the configuration file.
5. Start the connector.

   The connector registers with the Distributed Connector. When actions are sent to the Distributed Connector for the connector group that you configured, they are forwarded to this connector or to another connector in the group.

Set Up Secure Communication

You can configure Secure Socket Layer (SSL) connections between the connector and other ACI servers.

Configure Outgoing SSL Connections

To configure the connector to send data to other components (for example Connector Framework Server) over SSL, follow these steps.

To configure outgoing SSL connections

1. Open the HPE File System Connector configuration file in a text editor.
2. Specify the name of a section in the configuration file where the SSL settings are provided:
   - To send data to an ingestion server over SSL, set the `IngestSSLConfig` parameter in the `[Ingestion]` section. To send data from a single fetch task to an ingestion server over SSL, set `IngestSSLConfig` in a `[`*`TaskName`*`]` section.
   - To send data to a Distributed Connector over SSL, set the `SSLConfig` parameter in the `[DistributedConnector]` section.

- To send data to a View Server over SSL, set the `SSLConfig` parameter in the `[ViewServer]` section.

You can use the same settings for each connection. For example:

```
[Ingestion]
IngestSSLConfig=SSLOptions

[DistributedConnector]
SSLConfig=SSLOptions
```

3. Create a new section in the configuration file. The name of the section must match the name you specified in the `IngestSSLConfig` or `SSLConfig` parameter. Then specify the SSL settings to use.

`SSLMethod`	The SSL protocol to use.
`SSLCertificate`	(Optional) The SSL certificate to use (in PEM format).
`SSLPrivateKey`	(Optional) The private key for the SSL certificate (in PEM format).

For example:

```
[SSLOptions]
SSLMethod=TLSV1.2
SSLCertificate=host1.crt
SSLPrivateKey=host1.key
```

4. Save and close the configuration file.
5. Restart the connector.

Related Topics

- Start and Stop the Connector, on page 40

Configure Incoming SSL Connections

To configure a connector to accept data sent to its ACI port over SSL, follow these steps.

To configure an incoming SSL Connection

1. Stop the connector.
2. Open the configuration file in a text editor.
3. In the `[Server]` section set the `SSLConfig` parameter to specify the name of a section in the configuration file for the SSL settings. For example:

```
[Server]
SSLConfig=SSLOptions
```

4. Create a new section in the configuration file (the name must match the name you used in the `SSLConfig` parameter). Then, use the SSL configuration parameters to specify the details for the connection. You must set the following parameters:

`SSLMethod`	The SSL protocol to use.

`SSLCertificate` The SSL certificate to use (in PEM format).

`SSLPrivateKey` The private key for the SSL certificate (in PEM format).

For example:

```
[SSLOptions]
SSLMethod=TLSV1.2
SSLCertificate=host1.crt
SSLPrivateKey=host1.key
```

5. Save and close the configuration file.
6. Restart the connector.

Related Topics

- Start and Stop the Connector, on page 40

Backup and Restore the Connector's State

After configuring a connector, and while the connector is running, you can create a backup of the connector's state. In the event of a failure, you can restore the connector's state from the backup.

To create a backup, use the `backupServer` action. The `backupServer` action saves a ZIP file to a path that you specify. The backup includes:

- a copy of the `actions` folder, which stores information about actions that have been queued, are running, and have finished.
- a copy of the configuration file.
- a copy of the connector's datastore files, which contain information about the files, records, or other data that the connector has retrieved from a repository.

Backup a Connector's State

To create a backup of the connectors state

- In the address bar of your Web browser, type the following action and press **ENTER**:

  `http://host:port/action=backupServer&path=path`

  where,

  host The host name or IP address of the machine where the connector is running.

  port The connector's ACI port.

  path The folder where you want to save the backup.

  For example:

  `http://localhost:1234/action=backupServer&path=./backups`

Restore a Connector's State

To restore a connector's state

- In the address bar of your Web browser, type the following action and press **ENTER**:

  `http://host:port/action=restoreServer&filename=filename`

  where,

host	The host name or IP address of the machine where the connector is running.
port	The connector's ACI port.
filename	The path of the backup that you created.

  For example:

  `http://localhost:1234/action=restoreServer&filename=./backups/filename.zip`

Validate the Configuration File

You can use the `ValidateConfig` service action to check for errors in the configuration file.

> **NOTE:**
> For the `ValidateConfig` action to validate a configuration section, HPE File System Connector must have previously read that configuration. In some cases, the configuration might be read when a task is run, rather than when the component starts up. In these cases, `ValidateConfig` reports any unread sections of the configuration file as unused.

To validate the configuration file

- Send the following action to HPE File System Connector:

  `http://Host:ServicePort/action=ValidateConfig`

  where:

Host	is the host name or IP address of the machine where HPE File System Connector is installed.
ServicePort	is the service port, as specified in the `[Service]` section of the configuration file.

Example Configuration File

```
[License]
LicenseServerHost=licenses
LicenseServerACIPort=20000
```

```
[Server]
Port=7002
XSLTemplates=TRUE

[AuthorizationRoles]
0=AdminRole
1=QueryRole

[AdminRole]
StandardRoles=admin,servicecontrol,query,servicestatus
Clients=::1,127.0.0.1

[QueryRole]
StandardRoles=query,servicestatus
Clients=*

[logging]
LogLevel=NORMAL
LogDirectory=logs
0=ApplicationLogStream
1=ActionLogStream
2=SynchronizeLogStream
3=CollectLogStream
4=ViewLogStream

[ApplicationLogStream]
LogFile=application.log
LogTypeCSVs=application

[ActionLogStream]
LogFile=action.log
LogTypeCSVs=action

[SynchronizeLogStream]
LogFile=synchronize.log
LogTypeCSVs=synchronize

[CollectLogStream]
LogFile=collect.log
LogTypeCSVs=collect

[ViewLogStream]
LogFile=view.log
LogTypeCSVs=view

[Service]
ServicePort=17002

[Ingestion]
```

```
IngesterType=CFS
IngestHost=localhost
IngestPort=7000
IndexDatabase=FileSystem

[Connector]
EnableIngestion=true
EnableExtraction=true
EnableFieldNameStandardization=true
TempDirectory=Temp
EnableScheduledTasks=true
ScheduleCycles=-1

[FetchTasks]
Number=1
0=MyTask

[MyTask]
DirectoryPathCSVs=C:\MyFiles\,D:\MyFiles
DirectoryRecursive=TRUE
```

Chapter 4: Start and Stop the Connector

This section describes how to start and stop the HPE File System Connector.

> **NOTE:**
> You must start and stop the Connector Framework Server separately from the HPE File System Connector.

Start the Connector

After you have installed and configured a connector, you are ready to run it. Start the connector using one of the following methods.

Start the Connector on Windows

To start the connector using Windows Services

1. Open the Windows Services dialog box.
2. Select the connector service, and click **Start**.
3. Close the Windows Services dialog box.

To start the connector by running the executable

- In the connector installation directory, double-click the connector executable file.

Start the Connector on UNIX

To start the connector on a UNIX operating system, follow these steps.

To start the connector using the UNIX start script

1. Change to the installation directory.
2. Enter the following command:

   ```
   ./startconnector.sh
   ```

3. If you want to check the HPE File System Connector service is running, enter the following command:

   ```
   ps -aef | grep ConnectorInstallName
   ```

This command returns the HPE File System Connector service process ID number if the service is running.

Verify that HPE File System Connector is Running

After starting HPE File System Connector, you can run the following actions to verify that HPE File System Connector is running.

- GetStatus
- GetLicenseInfo

GetStatus

You can use the `GetStatus` service action to verify the HPE File System Connector is running. For example:

```
http://Host:ServicePort/action=GetStatus
```

> **NOTE:**
> You can send the `GetStatus` action to the ACI port instead of the service port. The `GetStatus` ACI action returns information about the HPE File System Connector setup.

GetLicenseInfo

You can send a `GetLicenseInfo` action to HPE File System Connector to return information about your license. This action checks whether your license is valid and returns the operations that your license includes.

Send the `GetLicenseInfo` action to the HPE File System Connector ACI port. For example:

```
http://Host:ACIport/action=GetLicenseInfo
```

The following result indicates that your license is valid.

```
<autn:license>
       <autn:validlicense>true</autn:validlicense>
</autn:license>
```

As an alternative to submitting the `GetLicenseInfo` action, you can view information about your license, and about licensed and unlicensed actions, on the **License** tab in the Status section of IDOL Admin.

Stop the Connector

You must stop the connector before making any changes to the configuration file.

To stop the connector using Windows Services

1. Open the Windows Services dialog box.
2. Select the ***ConnectorInstallName*** service, and click **Stop**.
3. Close the Windows Services dialog box.

To stop the connector by sending an action to the service port

- Type the following command in the address bar of your Web browser, and press ENTER:

  `http://host:ServicePort/action=stop`

host	The IP address or host name of the machine where the connector is running.
ServicePort	The connector's service port (specified in the `[Service]` section of the connector's configuration file).

Chapter 5: Send Actions to HPE File System Connector

This section describes how to send actions to HPE File System Connector.

Send Actions to HPE File System Connector

HPE File System Connector actions are HTTP requests, which you can send, for example, from your web browser. The general syntax of these actions is:

`http://host:port/action=action&parameters`

where:

host	is the IP address or name of the machine where HPE File System Connector is installed.
port	is the HPE File System Connector ACI port. The ACI port is specified by the `Port` parameter in the `[Server]` section of the HPE File System Connector configuration file. For more information about the `Port` parameter, see the *HPE File System Connector Reference*.
action	is the name of the action you want to run.
parameters	are the required and optional parameters for the action.

> **NOTE:**
> Separate individual parameters with an ampersand (&). Separate parameter names from values with an equals sign (=). You must percent-encode all parameter values.

For more information about actions, see the *HPE File System Connector Reference*.

Asynchronous Actions

When you send an asynchronous action to HPE File System Connector, the connector adds the task to a queue and returns a token. HPE File System Connector performs the task when a thread becomes available. You can use the token with the `QueueInfo` action to check the status of the action and retrieve the results of the action.

Most of the features provided by the connector are available through `action=fetch`, so when you use the `QueueInfo` action, query the `fetch` action queue, for example:

```
/action=QueueInfo&QueueName=Fetch&QueueAction=GetStatus
```

Check the Status of an Asynchronous Action

To check the status of an asynchronous action, use the token that was returned by HPE File System Connector with the `QueueInfo` action. For more information about the `QueueInfo` action, refer to the *HPE File System Connector Reference*.

To check the status of an asynchronous action

- Send the `QueueInfo` action to HPE File System Connector with the following parameters.

`QueueName`	The name of the action queue that you want to check.
`QueueAction`	The action to perform. Set this parameter to **`GetStatus`**.
`Token`	(Optional) The token that the asynchronous action returned. If you do not specify a token, HPE File System Connector returns the status of every action in the queue.

For example:

```
/action=QueueInfo&QueueName=fetch&QueueAction=getstatus&Token=...
```

Cancel an Asynchronous Action that is Queued

To cancel an asynchronous action that is waiting in a queue, use the following procedure.

To cancel an asynchronous action that is queued

- Send the `QueueInfo` action to HPE File System Connector with the following parameters.

`QueueName`	The name of the action queue that contains the action to cancel.
`QueueAction`	The action to perform . Set this parameter to **`Cancel`**.
`Token`	The token that the asynchronous action returned.

For example:

```
/action=QueueInfo&QueueName=fetch&QueueAction=Cancel&Token=...
```

Stop an Asynchronous Action that is Running

You can stop an asynchronous action at any point.

To stop an asynchronous action that is running

- Send the `QueueInfo` action to HPE File System Connector with the following parameters.

`QueueName`	The name of the action queue that contains the action to stop.
`QueueAction`	The action to perform. Set this parameter to **`Stop`**.
`Token`	The token that the asynchronous action returned.

For example:

```
/action=QueueInfo&QueueName=fetch&QueueAction=Stop&Token=...
```

Store Action Queues in an External Database

HPE File System Connector provides asynchronous actions. Each asynchronous action has a queue to store requests until threads become available to process them. You can configure HPE File System Connector to store these queues either in an internal database file, or in an external database hosted on a database server.

The default configuration stores queues in an internal database. Using this type of database does not require any additional configuration.

You might want to store the action queues in an external database so that several servers can share the same queues. In this configuration, sending a request to any of the servers adds the request to the shared queue. Whenever a server is ready to start processing a new request, it takes the next request from the shared queue, runs the action, and adds the results of the action back to the shared database so that they can be retrieved by any of the servers. You can therefore distribute requests between components without configuring a Distributed Action Handler (DAH).

> **NOTE:**
> You cannot use multiple servers to process a single request. Each request is processed by one server.

> **NOTE:**
> Although you can configure several connectors to share the same action queues, the connectors do not share fetch task data. If you share action queues between several connectors and distribute synchronize actions, the connector that processes a synchronize action cannot determine which items the other connectors have retrieved. This might result in some documents being ingested several times.

Prerequisites

- Supported databases:
  - PostgreSQL 9.0 or later.
  - MySQL 5.0 or later.
- If you use PostgreSQL, you must set the PostgreSQL ODBC driver setting `MaxVarChar` to **`0`** (zero).

If you use a DSN, you can configure this parameter when you create the DSN. Otherwise, you can set the `MaxVarcharSize` parameter in the connection string.

Configure HPE File System Connector

To configure HPE File System Connector to use a shared action queue, follow these steps.

To store action queues in an external database

1. Stop HPE File System Connector, if it is running.
2. Open the HPE File System Connector configuration file.
3. Find the relevant section in the configuration file:
   - To store queues for all asynchronous actions in the external database, find the `[Actions]` section.
   - To store the queue for a single asynchronous action in the external database, find the section that configures that action.
4. Set the following configuration parameters.

`AsyncStoreLibraryDirectory`	The path of the directory that contains the library to use to connect to the database. Specify either an absolute path, or a path relative to the server executable file.
`AsyncStoreLibraryName`	The name of the library to use to connect to the database. You can omit the file extension. The following libraries are available: • **`postgresAsyncStoreLibrary`** - for connecting to a PostgreSQL database. • **`mysqlAsyncStoreLibrary`** - for connecting to a MySQL database.

`ConnectionString`	The connection string to use to connect to the database. The user that you specify must have permission to create tables in the database. For example:
	`ConnectionString=DSN=my_ASYNC_QUEUE`
	or
	`ConnectionString=Driver={PostgreSQL}; Server=10.0.0.1; Port=9876; Database=SharedActions; Uid=user; Pwd=password; MaxVarcharSize=0;`

   For example:

```
[Actions]
AsyncStoreLibraryDirectory=acidlls
AsyncStoreLibraryName=postgresAsyncStoreLibrary
ConnectionString=DSN=ActionStore
```

5. If you are using the same database to store action queues for more than one type of component,

set the following parameter in the `[Actions]` section of the configuration file.

`DatastoreSharingGroupName`	The group of components to share actions with. You can set this parameter to any string, but the value must be the same for each server in the group. For example, to configure several HPE File System Connectors to share their action queues, set this parameter to the same value in every HPE File System Connector configuration. HPE recommends setting this parameter to the name of the component. **CAUTION:** Do not configure different components (for example, two different types of connector) to share the same action queues. This will result in unexpected behavior.

```
For example:

[Actions]
...
DatastoreSharingGroupName=ComponentType
```

6. Save and close the configuration file.

   When you start HPE File System Connector it connects to the shared database.

Store Action Queues in Memory

HPE File System Connector provides asynchronous actions. Each asynchronous action has a queue to store requests until threads become available to process them. These queues are usually stored in a datastore file or in a database hosted on a database server, but in some cases you can increase performance by storing these queues in memory.

> **NOTE:**
> Storing action queues in memory improves performance only when the server receives large numbers of actions that complete quickly. Before storing queues in memory, you should also consider the following:
>
> - The queues (including queued actions and the results of finished actions) are lost if HPE File System Connector stops unexpectedly, for example due to a power failure or the component being forcibly stopped. This could result in some requests being lost, and if the queues are restored to a previous state some actions could run more than once.
> - Storing action queues in memory prevents multiple instances of a component being able to share the same queues.
> - Storing action queues in memory increases memory use, so please ensure that the server has sufficient memory to complete actions and store the action queues.

If you stop HPE File System Connector cleanly, HPE File System Connector writes the action queues from memory to disk so that it can resume processing when it is next started.

To configure HPE File System Connector to store asynchronous action queues in memory, follow these steps.

To store action queues in memory

1. Stop HPE File System Connector, if it is running.
2. Open the HPE File System Connector configuration file and find the `[Actions]` section.
3. If you have set any of the following parameters, remove them:
   - `AsyncStoreLibraryDirectory`
   - `AsyncStoreLibraryName`
   - `ConnectionString`
   - `UseStringentDatastore`
4. Set the following configuration parameters.

`UseInMemoryDatastore`	A Boolean value that specifies whether to keep the queues for asynchronous actions in memory. Set this parameter to **`TRUE`**.
`InMemoryDatastoreBackupIntervalMins`	(Optional) The time interval (in minutes) at which the action queues are written to disk. Writing the queues to disk can reduce the number of queued actions that would be lost if HPE File System Connector stops unexpectedly, but configuring a frequent backup will increase the load on the datastore and might reduce performance.

   For example:

```
[Actions]
UseInMemoryDatastore=TRUE
InMemoryDatastoreBackupIntervalMins=30
```

5. Save and close the configuration file.

   When you start HPE File System Connector, it stores action queues in memory.

Use XSL Templates to Transform Action Responses

You can transform the action responses returned by HPE File System Connector using XSL templates. You must write your own XSL templates and save them with either an `.xsl` or `.tmpl` file extension.

After creating the templates, you must configure HPE File System Connector to use them, and then apply them to the relevant actions.

To enable XSL transformations

1. Ensure that the `autnxslt` library is located in the same directory as your configuration file. If the library is not included in your installation, you can obtain it from HPE Support.
2. Open the HPE File System Connector configuration file in a text editor.
3. In the `[Server]` section, ensure that the `XSLTemplates` parameter is set to **`true`**.

> **CAUTION:**
> If `XSLTemplates` is set to **`true`** and the `autnxslt` library is not present in the same directory as the configuration file, the server will not start.

4. (Optional) In the `[Paths]` section, set the `TemplateDirectory` parameter to the path to the directory that contains your XSL templates. The default directory is `acitemplates`.
5. Save and close the configuration file.
6. Restart HPE File System Connector for your changes to take effect.

To apply a template to action output

- Add the following parameters to the action:

`Template`	The name of the template to use to transform the action output. Exclude the folder path and file extension.
`ForceTemplateRefresh`	(Optional) If you modified the template after the server started, set this parameter to **`true`** to force the ACI server to reload the template from disk rather than from the cache.

For example:

```
action=QueueInfo&QueueName=Fetch
                &QueueAction=GetStatus
                &Token=...
                &Template=myTemplate
```

In this example, HPE File System Connector applies the XSL template `myTemplate` to the response from a `QueueInfo` action.

> **NOTE:**
> If the action returns an error response, HPE File System Connector does not apply the XSL template.

Example XSL Templates

HPE File System Connector includes the following sample XSL templates, in the `acitemplates` folder:

XSL Template	Description
`FetchIdentifiers`	Transforms the output from the `Identifiers` fetch action, to show what is in the repository.
`LuaDebug`	Transforms the output from the `LuaDebug` action, to assist with debugging Lua scripts.

Chapter 6: Use the Connector

This section describes how to retrieve data from a file system. You can configure the connector to retrieve files that match specific criteria. For example, you can retrieve files that match a specific file type or were modified after a specific date.

To retrieve data from a file system, run the `synchronize` fetch action. You can do this manually, or by creating a *fetch task* in the connector's configuration file. The connector runs fetch tasks automatically, based on a schedule that you define.

Retrieve Information from a File System

To automatically retrieve files from a file system, create a new fetch task by following these steps. The connector runs each fetch task automatically, based on the schedule that is configured in the configuration file.

To create a new Fetch Task

1. Stop the connector.
2. Open the configuration file in a text editor.
3. In the `[FetchTasks]` section of the configuration file, specify the number of fetch tasks using the `Number` parameter. If you are configuring the first fetch task, type **`Number=1`**. If one or more fetch tasks have already been configured, increase the value of the `Number` parameter by one (1). Below the `Number` parameter, specify the names of the fetch tasks, starting from zero (0). For example:

   ```
   [FetchTasks]
   Number=1
   0=MyTask
   ```

4. Below the `[FetchTasks]` section, create a new *TaskName* section. The name of the section must match the name of the new fetch task. For example:

   ```
   [FetchTasks]
   Number=1
   0=MyTask

   [MyTask]
   ```

5. To specify the folders that the connector checks for files, set the `DirectoryPathCSVs` and `DirectoryRecursive` parameters.

`DirectoryPathCSVs`	A comma-separated list of folders from which you want to retrieve files.
`DirectoryRecursive`	A Boolean that specifies whether the connector retrieves files in sub-folders.

```
[MyTask]
DirectoryPathCSVs=C:\MyFiles\,D:\MyFiles
DirectoryRecursive=True
```

> **TIP:**
> Using the `DirectoryPathCSVs` parameter, you can specify paths that cause the connector to retrieve files from different types of file system (for example, NT and Netware). However, HPE recommends that you create separate fetch tasks for retrieving files from different types of file system. This ensures that Access Control Lists can be retrieved correctly.

6. (Optional) To limit the folders that the connector checks for files, you can set the `PathCrawlRegex` and `PathNoCrawlRegex` parameters. These parameters can be useful when the location specified by `DirectoryPathCSVs` contains large folder structures that you do not want to index. For information about these parameters, refer to the *File System Connector (CFS) Reference*.
7. (Optional) To limit the files that are retrieved, you can use the file selection parameters. For information about these parameters, refer to the *File System Connector (CFS) Reference*. A file is retrieved only if it matches all of the criteria that are set. For example:

```
[MyTask]
DirectoryPathCSVs=C:\MyFiles\,D:\MyFiles
DirectoryRecursive=True
DirectoryFileMatch=*.pdf,*.doc
DirectoryFileModifiedSince=2012-JUN-15 00:00:01
```

8. Save and close the configuration file. You can now start the connector.

> **NOTE:**
> The connector saves or updates a data file for each completed fetch task. If you make changes to the configuration and want to reset the connector so that it retrieves all of your data again, see Reset the Connector, on page 59.

Related Topics

- Start and Stop the Connector, on page 40
- Schedule Fetch Tasks, below

Schedule Fetch Tasks

The connector automatically runs the fetch tasks that you have configured, based on the schedule in the configuration file. To modify the schedule, follow these steps.

To schedule fetch tasks

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. Find the `[Connector]` section.
4. The `EnableScheduleTasks` parameter specifies whether the connector should automatically run the fetch tasks that have been configured in the `[FetchTasks]` section. To run the tasks, set this parameter to `true`. For example:

   ```
   [Connector]
   EnableScheduledTasks=True
   ```

5. In the `[Connector]` section, set the following parameters:

`ScheduleStartTime`	The start time for the fetch task, the first time it runs after you start the connector. The connector runs subsequent synchronize cycles after the interval specified by `ScheduleRepeatSecs`. Specify the start time in the format H[H][:MM][:SS]. To start running tasks as soon as the connector starts, do not set this parameter or use the value **`now`**.
`ScheduleRepeatSecs`	The interval (in seconds) from the start of one scheduled `synchronize` cycle to the start of the next. If a previous synchronize cycle is still running when the interval elapses, the connector queues a maximum of one action.
`ScheduleCycles`	The number of times that each fetch task is run. To run the tasks continuously until the connector is stopped, set this parameter to **`-1`**. To run each task only one time, set this parameter to **`1`**.

   For example:

   ```
   [Connector]
   EnableScheduledTasks=True
   ScheduleStartTime=15:00:00
   ScheduleRepeatSecs=3600
   ScheduleCycles=-1
   ```

6. (Optional) To run a specific fetch task on a different schedule, you can override these parameters in a *TaskName* section of the configuration file. For example:

   ```
   [Connector]
   EnableScheduledTasks=TRUE
   ScheduleStartTime=15:00:00
   ScheduleRepeatSecs=3600
   ScheduleCycles=-1

   ...

   [FetchTasks]
   Number=2
   ```

```
0=MyTask0
1=MyTask1
...

[MyTask1]
ScheduleStartTime=16:00:00
ScheduleRepeatSecs=60
ScheduleCycles=-1
```

In this example, `MyTask0` follows the schedule defined in the `[Connector]` section, and `MyTask1` follows the scheduled defined in the `[MyTask1]` *TaskName* section.

7. Save and close the configuration file. You can now start the connector.

Related Topics

- Start and Stop the Connector, on page 40

Synchronize from Identifiers

The connector's `synchronize` action searches a repository for document updates and sends these updates for ingestion (for example, to CFS, for indexing into IDOL Server).

You can use the `identifiers` parameter to synchronize a specific set of documents, whether they have been updated or not, and ignore other files. For example:

`/action=fetch&fetchaction=synchronize&identifiers=<identifiers>`

(where `<identifiers>` is a comma-separated list of identifiers that specifies the documents to synchronize).

For example, if some documents fail the ingestion process, and are indexed into an IDOL Error Server, you can use the `identifiers` parameter with the `synchronize` action to retrieve those documents again. You can retrieve a list of identifiers for the failed documents by sending a query to the IDOL Error Server. For more information about IDOL Error Server, refer to the *IDOL Error Server Technical Note*. For more information about the `synchronize` action, refer to the *HPE File System Connector Reference*.

Insert Files into the File System

The connector's `insert` fetch action can insert files into the file system. To use the `insert` action, you must construct some XML that specifies where to add each file, and the information to insert.

To insert a file, set the `reference` to the path where you want to insert the file. Use the `file` element to specify the content to insert. There are several ways that you can do this, for example specifying the path to a file or providing the content base-64 encoded. For more information about how to specify the source file, refer to the *HPE File System Connector Reference*.

The following example would insert a file at the location `C:\files\file.txt`. The file would contain the content "This is my file".

```
<insertXML>
  <insert>
    <reference>C:\files\file.txt</reference>
    <file>
      <type>content</type>
      <content>VGhpcyBpcyBteSBmaWxl</content>
    </file>
    <insert_id>012345</insert_id>
  </insert>
</insertXML>
```

Add this XML to the insert fetch action as the value of the `insertXML` action parameter. The XML must be URL encoded before being used in the action command. For example:

```
http://host:port/action=Fetch&FetchAction=Insert
                            &ConfigSection=MyTask
                            &InsertXML=[URL encoded XML]
```

For more information about using the `insert` fetch action, refer to the *HPE File System Connector Reference*.

Update File Metadata

The connector's `update` fetch action updates the metadata of files in the file system. You can change the following metadata:

Windows When the connector is running on Windows and the file system is on a Windows server you can modify:

- The Access Control List (ACL)
- File creation time
- Last access time
- Last change time (the last time the security or sharing settings were changed)
- Last modification time

UNIX/Linux

- Last access time
- Last modification time

To use the `update` fetch action, the connector must have permission to modify files in the file system. If you are running the connector as a service and the user running the service does not have adequate permissions, set the configuration parameters `UpdateUsername` and `UpdatePassword` in the connector's configuration file. These parameters specify the user name and password of a user account to use to perform the updates.

To use the `update` fetch action, you must construct some XML that specifies the identifiers of the files to update, and provides the new values for any metadata fields that you want to change. Use the XML as the value of the action parameter `identifiersXML`. The XML must be URL-encoded before being used in the action command. For example:

```
http://host:port/action=Fetch&FetchAction=update
                            &IdentifiersXML=URL-encoded XML
```

For information about how to construct the XML, see Construct XML to Update Access Control Lists, below and Construct XML to Update Dates, on page 58.

Construct XML to Update Access Control Lists

To update the Access Control Lists of files in a file system, you must construct some XML that specifies the identifiers of the files to update, and provides information about how to change the ACL.

```
<identifiersXML>
   <identifier value="...">
     <acl_update>
       ...
     </acl_update>
   </identifier>
 </identifiersXML>
```

In the identifier `value` attribute, replace "..." with the document identifier of the file that you want to update. A document identifier can be found in the `AUTN_IDENTIFIER` field of an indexed document.

You can update the ACLs of several files by including more than one `identifier` element in your XML:

```
<identifiersXML>
   <identifier value="...">
      <acl_update>
         ...
      </acl_update>
   </identifier>
   <identifier value="...">
      <acl_update>
         ...
      </acl_update>
   </identifier>
</identifiersXML>
```

The following table describes the XML elements that you can use in the `acl_update` element to specify how to change the ACL:

XML Element	Description	Permitted

		Occurrences
<break_inheritance/>	Add this element to your XML to prevent ACL settings being inherited from the parent object in the file system. If specified, this element must be the first child of `acl_update`.	0 or 1
<enable_inheritance/>	Add this element to your XML to inherit ACL settings from the parent object in the file system. If specified, this element must be the first child of `acl_update`. This element accepts an optional `revert_acl` attribute that specifies whether to remove all non-inherited entries in the ACL. Set this attribute to `true` or `false` (which is the default). For example: `<enable_inheritance revert_acl="true" />`	0 or 1
<ace action="...">	Add or remove an entry from the ACL. The `action` attribute must be specified and accepts the value `add` or `remove`. The following child elements must all appear exactly once: • `principal` - the user or group whose permissions you want to modify in the ACL. You can specify a domain user name or an SID. • `principalType` - the type of principal specified by the `principal` element ◦ `DomainUser` ◦ `SID` • `level` - a comma-separated list of permissions to add or remove. ◦ `All` (equivalent to "full control") ◦ `Read` ◦ `Write` ◦ `Execute` ◦ `Delete` ◦ `DenyAll` ◦ `DenyRead` ◦ `DenyWrite` ◦ `DenyExecute` ◦ `DenyDelete`	0 or more

The following example demonstrates how to change the ACL for a file:

- grant read permission to `MYDOMAIN\user1`
- grant read, write, execute, and delete permissions to `MYDOMAIN\user2`
- remove all "allow" permissions from `MYDOMAIN\user3`

```
<identifiersXML>
   <identifier value="...">
     <acl_update>
       <break_inheritance/>
       <ace action="add">
         <principal>MYDOMAIN\user1</principal>
         <principalType>DomainUser</principalType>
         <level>Read</level>
       </ace>
       <ace action="add">
         <principal>MYDOMAIN\user2</principal>
         <principalType>DomainUser</principalType>
         <level>Read, Write, Execute, Delete</level>
       </ace>
       <ace action="remove">
         <principal>MYDOMAIN\user3</principal>
         <principalType>DomainUser</principalType>
         <level>All</level>
       </ace>
     </acl_update>
   </identifier>
 </identifiersXML>
```

The following example demonstrates how to change the ACL for a file, so that ACL entries are inherited from the parent object in the file system and all non-inherited entries are removed:

```
<identifiersXML>
   <identifier value="...">
     <acl_update>
       <enable_inheritance revert_acl="true"/>
     </acl_update>
   </identifier>
 </identifiersXML>
```

> **NOTE:**
> When you update an ACL the file's last change time is updated to the current time (unless you provide a value for the `LASTCHANGED` metadata field, in which case that value is used instead). For more information about changing the last change time, see Construct XML to Update Dates, on the next page.

Construct XML to Update Dates

To use the `update` fetch action, you must construct some XML that specifies the identifiers of the files to update, and provides the new values for any metadata fields that you want to change:

```
<identifiersXML>
  <identifier value="...">
    <metadata name="CREATED" value="1427302848"/>
    <metadata name="LASTMODIFIED" value="1427302848"/>
    <metadata name="LASTACCESSED" value="1427302848"/>
    <metadata name="LASTCHANGED" value="1427302848"/>
  </identifier>
</identifiersXML>
```

In the identifier `value` attribute, replace "..." with the document identifier of the item that you want to update. A document identifier can be found in the `AUTN_IDENTIFIER` field of an indexed document.

You can update the metadata of several files by including more than one `identifier` element in your XML:

```
<identifiersXML>
   <identifier value="...">
      <metadata name="CREATED" value="1427302848"/>
   </identifier>
   <identifier value="...">
      <metadata name="LASTMODIFIED" value="1427302848"/>
   </identifier>
</identifiersXML>
```

You can set date/time values using the following formats:

- Epoch seconds (for example 1427302848)
- YYYY-SHORTMONTH-DD HH:NN:SS (for example 2015-MAR-25 17:01:30)
- YYYY-MM-DDTHH:NN:SS (for example 2015-03-25T17:01:30)
- YYYYMMDDTHHNNSS (for example 20150325T170130)

TIP:
The connector passes the times to the operating system without any consideration of time zones.

NOTE:
The connector updates any of the fields that you specify. If you do not set a value for one of the fields, the value is not updated. The last change time is *not* updated unless you explicitly set a value for that field, even though updating the other metadata fields would usually result in the last change time being updated to the current time.

Reset the Connector

When the connector runs the `synchronize` action, it updates a datastore file that stores information about the data retrieved from the repository. The next time the connector runs the `synchronize` action, it retrieves only data that is new or has been modified. The connector can also determine whether files or records have been deleted, so that related documents can be removed from the IDOL index.

When you are configuring the connector and you make a change to the configuration, you might want to purge all information from the datastore so that the connector retrieves all of your data again.

To purge the datastore for a fetch task

- To purge the datastore for a fetch task, use the `PurgeDatastore` action. Specify the name of the task as the value of the `section` parameter, for example:

  ```
  /action=PurgeDatastore&section=MyTask
  ```

In some cases you might want to delete all queued actions.

To delete the actions queue

1. Stop the connector.
2. Delete the `actions` folder. This ensures that information about incomplete and queued actions is deleted.
3. Restart the connector.

Troubleshoot the Connector

This section describes how to troubleshoot common problems that might occur when you set up the File System Connector.

Files are not retrieved

If the File System Connector does not retrieve the files that you expect, check whether the files are excluded by the configuration parameters that you have set, particularly those that accept regular expressions.

If you encounter problems with permissions, ensure you run the connector as a user who has the necessary permissions for the files that you want to retrieve. For more information about permissions, see Permissions, on page 21.

NSF or PST files are not retrieved

The File System Connector can retrieve files that are read-only. However, due to limitations with the MAPI and Notes APIs, KeyView cannot process read-only `.nsf` and `.pst` files.

To resolve this issue, you can:

- remove the read-only attribute from these files (the files will appear to be modified but the data, such as e-mail messages, are not changed).

- configure CFS to copy the files to a temporary directory (the temporary copies are not read-only). To do this, set the `WorkingDirectory` parameter in the CFS configuration file.

Files are ingested repeatedly

When performing extraction from `.nsf` and `.pst` files, KeyView can change the last modified date of a file. This causes the connector to re-ingest the document on the next `synchronize` action. You can prevent this happening by setting the `EnableExtractionCopy` parameter to `true`. When this parameter is `true`, the connector makes a copy of the original document and KeyView performs extraction on the copy.

The connector does not retrieve all items from a Netware ACL

If the connector does not retrieve all of the security information from ACLs in a Netware file system, make sure that the connector's Windows service is run by a user who has administrative permissions on the connector machine.

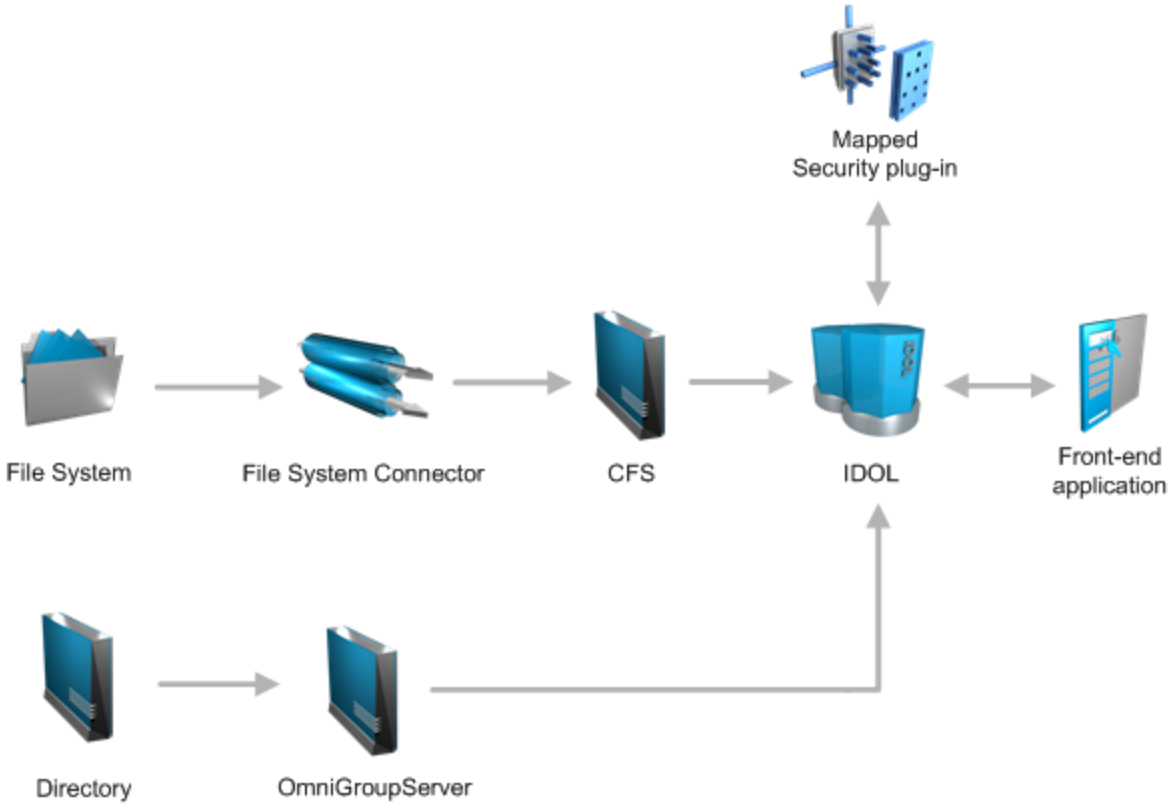
Chapter 7: Mapped Security

This section describes how to set up mapped security for information that is extracted from a file system.

Introduction

The mapped security architecture includes the following components:

- The file system.
- HPE File System Connector (CFS)
- HPE OmniGroupServer
- HPE IDOL server
- HPE IDOL Mapped Security plug-in
- A front-end application



Files in the file system have an Access Control List (ACL) that lists the users and groups who are permitted, and are not permitted, to view the file.

The File System Connector retrieves files from the file system and sends documents to CFS to be indexed into IDOL. The connector extracts the ACL for each item, and writes it to a document field. Each time the connector synchronizes with the repository, it extracts updated ACLs.

IDOL needs the ACL to determine whether a user can view a document that is returned as a result to a query. However, IDOL must also consider the groups that the user belongs to. A user might not be permitted to view a document, but they could be a member of a group that has permission. This means that IDOL requires the user and group information associated with the files.

OmniGroupServer extracts the security group information using LDAP, and stores it.

When a user logs on to a front-end application, the application requests the user's security information and group memberships from IDOL server. IDOL returns a token containing the information. The front-end application includes this token in all queries the user sends to IDOL.

After a user submits a query, IDOL sends the result documents and the user's security token to the Mapped Security plug-in. The Mapped Security plug-in compares the user's security information and group memberships to each document's ACL. The plug-in determines which documents the user is permitted to view and returns the results. IDOL server then sends only the documents that the user is permitted to view to the front-end application.

Set up Mapped Security

To use mapped security to protect information that was extracted from a file system by the HPE File System Connector, set up the following components:

- IDOL server. You must set up IDOL server to process the security information contained in each document. You must also configure user security, so that IDOL sends user and group information to the front-end application when a user logs on. For information about how to set up IDOL server, refer to the *IDOL Document Security Administration Guide*.
- HPE File System Connector. You must set up the HPE File System Connector to include security information (Access Control Lists) in the documents that are indexed into IDOL server. The connector encrypts the ACL and adds it to a document field named `AUTONOMYMETADATA`. You must also add a field to each document that identifies the security type. For information about how to do this, see Retrieve and Index Access Control Lists, below.
- OmniGroupServer. You must set up OmniGroupServer to retrieve group information. HPE recommends retrieving the group information through LDAP. For information about how to configure OmniGroupServer, see Retrieve Security Group Information, on page 64.
- A front-end application for querying IDOL server.

Retrieve and Index Access Control Lists

To configure the File System Connector to retrieve and index Access Control Lists (ACLs), follow these steps.

To retrieve and index Access Control Lists

1. If the connector is running, stop the connector.
2. Open the connector's configuration file.
3. Set the `MappedSecurity` parameter to `true`.
   - To index ACLs for all fetch tasks, set this parameter in the `[FetchTasks]` section.
   - To index ACLs for a single fetch task, set this parameter in the `TaskName` section for the task.

   For example:

   ```
   [FetchTasks]
   MappedSecurity=True
   ```
4. Add a field to each document to specify the security type. To do this, create an ingest action or run a Lua script. For example:

   ```
   [Ingestion]
   IngestActions=META:SecurityType=NT
   ```

   > **NOTE:**
   > The field name and value that you specify must match the name and value you used to identify the security type in your IDOL Server configuration file.
5. Save and close the configuration file.

Retrieve ACLs from a Netware File System

This section describes how to retrieve Netware ACLs for files retrieved from a Netware file system.

Requirements

- To retrieve files from a Netware file system, the connector must be installed on a machine running Windows. You must also install the Novell client on the connector machine. This allows the connector to access the Netware server.
- Run the connector's Windows service as a user who has administrative permissions on the connector machine.
- It is possible to retrieve files from an NT file system and a Netware file system using the same fetch task. However, to retrieve Access Control Lists for the documents you must use separate fetch tasks. For example:

  ```
  [FetchTasks]
  Number=2
  0=MyTask
  1=Netware
  MappedSecurity=True

  [MyTask]
  // Task for extracting files from an NT file system
  ```

```
DirectoryPathCSVs=\\server.domain.com\folder,\\server.domain.com\anotherfolder

[Netware]
// Task for extracting files from Netware
DirectoryPathCSVs=\\netware\sys\public
NetwareServer=10.0.0.7
NetwareUsername=CN=ADMIN.O=ORG
NetwarePassword=password
```

- The `NetwareUsername` parameter must be set and must contain a distinguished name, for example:

  `NetwareUsername=CN=ADMIN.O=ORG`
- The `DirectoryPathCSVs` configuration parameter must be configured using a full path, for example:

  `DirectoryPathCSVs=\\netware\sys\public`
- To ensure that CFS has access to the files retrieved from a Netware file system, it might be necessary to make a temporary copy of the files by setting the `IngestSharedPath` parameter. For example:

  ```
  [Ingestion]
  IngestSharedPath=\\server\IngestShared\
  ```

Troubleshooting

If the connector fails with the following error, this usually indicates that the value specified for the `NetwareUsername` configuration parameter is not known or is not a valid distinguished name:

```
<TASK>: Netware error 889a (NWDSLogin)
Failed to complete <ACTION> for task '<TASK>' [<TOKEN>]: Netware initialization or
logon failed
```

Retrieve Security Group Information

To retrieve the security groups using OmniGroupServer, follow these steps.

To retrieve security groups

1. Open the OmniGroupServer configuration file.
2. In the `[Repositories]` section, create a repository to store the groups. For example:

   ```
   [Repositories]
   Number=1
   0=LDAP
   ```
3. In the new section, configure a task to extract the information from the directory using LDAP. You can use the following configuration parameters (for a complete list of configuration parameters, refer to the *OmniGroupServer Reference*).

`GroupServerLibrary`	The path (including the file name) to the library file that allows the group server to access the repository. Use the LDAP group server library.

`LDAPServer`	The host name or IP address of the machine that hosts the LDAP directory.
`LDAPPort`	The port to use to access the LDAP directory.
`LDAPBase`	The distinguished name of the search base.
`LDAPType`	The type of LDAP server (for example, Microsoft Active Directory).
`LDAPSecurityType`	The type of security to use when communicating with the LDAP server (for example, `SSL` or `TLS`).
`LDAPBindMethod`	The type of authentication to use to access the LDAP directory. To log on as the same user that is running OmniGroupServer, set this parameter to `NEGOTIATE`.

For example:

```
[LDAP]
GroupServerLibrary=ogs_ldap.dll
LDAPServer=myLDAPserver
LDAPPort=636
LDAPBase=DC=DOMAIN,DC=COM
LDAPType=MAD
LDAPSecurityType=SSL
LDAPBindMethod=NEGOTIATE
```

4. (Optional) You can set further parameters to define the schedule for the task. You can set these parameters in the task section (to schedule only the current task), or in the `[Default]` section (to provide a default schedule for all OmniGroupServer tasks).

`GroupServerStartTime`	The time when a task starts.
`GroupServerRepeatSecs`	The number of seconds that elapse before the Group Server repeats a task.

For example:

```
[LDAP]
GroupServerLibrary=ogs_ldap.dll
LDAPServer=myLDAPserver
LDAPPort=636
LDAPBase=DC=DOMAIN,DC=COM
LDAPType=MAD
LDAPSecurityType=SSL
LDAPBindMethod=NEGOTIATE
GroupServerStartTime=12:00
GroupServerRepeatSecs=3600
```

5. Save and close the OmniGroupServer configuration file.

Chapter 8: Manipulate Documents

This section describes how to manipulate documents that are created by the connector and sent for ingestion.

Introduction

IDOL Connectors retrieve data from repositories and create documents that are sent to Connector Framework Server, another connector, or Haven OnDemand. You might want to manipulate the documents that are created. For example, you can:

- Add or modify document fields, to change the information that is indexed into IDOL Server or Haven OnDemand.
- Add fields to a document to customize the way the document is processed by CFS.
- Convert information into another format so that it can be inserted into another repository by a connector that supports the `Insert` action.

When a connector sends documents to CFS, the documents only contain metadata extracted from the repository by the connector (for example, the location of the original files). To modify data extracted by KeyView, you must modify the documents using CFS. For information about how to manipulate documents with CFS, refer to the *Connector Framework Server Administration Guide*.

Add a Field to Documents using an Ingest Action

To add a field to all documents retrieved by a fetch task, or all documents sent for ingestion, you can use an Ingest Action.

> **NOTE:**
> To add a field only to selected documents, use a Lua script (see Run Lua Scripts, on page 73). For an example Lua script that demonstrates how to add a field to a document, see Add a Field to a Document, on page 76.

To add a field to documents using an Ingest Action

1. Open the connector's configuration file.
2. Find one of the following sections in the configuration file:

- To add the field to all documents retrieved by a specific fetch task, find the `[TaskName]` section.
- To add a field to all documents that are sent for ingestion, find the `[Ingestion]` section.

> **NOTE:**
> If you set the `IngestActions` parameter in a `[TaskName]` section, the connector does not run any `IngestActions` set in the `[Ingestion]` section for documents retrieved by that task.

3. Use the `IngestActions` parameter to specify the name of the field to add, and the field value. For example, to add a field named `AUTN_NO_EXTRACT`, with the value `SET`, type:

   ```
   IngestActions0=META:AUTN_NO_EXTRACT=SET
   ```

4. Save and close the configuration file.

Customize Document Processing

You can add the following fields to a document to control how the document is processed by CFS. Unless stated otherwise, you can add the fields with any value.

AUTN_FILTER_META_ONLY

Prevents KeyView extracting file content from a file. KeyView only extracts metadata and adds this information to the document.

AUTN_NO_FILTER

Prevents KeyView extracting file content and metadata from a file. You can use this field if you do not want to extract text from certain file types.

AUTN_NO_EXTRACT

Prevents KeyView extracting subfiles. You can use this field to prevent KeyView extracting the contents of ZIP archives and other container files.

AUTN_NEEDS_MEDIA_SERVER_ANALYSIS

Identifies media files (images, video, and documents such as PDF files that contain embedded images) that you want to send to Media Server for analysis, using a `MediaServerAnalysis` import task. You do not need to add this field if you are using a Lua script to run media analysis. For more information about running analysis on media, refer to the *Connector Framework Server Administration Guide*.

AUTN_NEEDS_TRANSCRIPTION

Identifies audio and video assets that you want to send to an IDOL Speech Server for speech-to-text processing, using an `IdolSpeech` import task. You do not need to add this field if you are using a Lua script to run speech-to-text. For more information about running speech-to-text on documents, refer to the *Connector Framework Server Administration Guide*.

AUTN_FORMAT_CORRECT_FOR_TRANSCRIPTION

To bypass the transcoding step of an `IdolSpeech` import task, add the field `AUTN_FORMAT_CORRECT_FOR_TRANSCRIPTION`. Documents that have this field are not sent to a Transcode Server. For more

information about the IdolSpeech task, refer to the *Connector Framework Server Administration Guide*.

AUTN_AUDIO_LANGUAGE

To bypass the language identification step of an IdolSpeech import task add the field AUTN_AUDIO_LANGUAGE. The value of the field must be the name of the IDOL Speech Server language pack to use for extracting speech. Documents that have this field are not sent to the IDOL Speech Server for language identification. For more information about the IdolSpeech task, refer to the *Connector Framework Server Administration Guide*.

Standardize Field Names

Field standardization modifies documents so that they have a consistent structure and consistent field names. You can use field standardization so that documents indexed into IDOL through different connectors use the same fields to store the same type of information.

For example, documents created by the File System Connector can have a field named FILEOWNER. Documents created by the Documentum Connector can have a field named owner_name. Both of these fields store the name of the person who owns a file. Field standardization renames the fields so that they have the same name.

Field standardization only modifies fields that are specified in a dictionary, which is defined in XML format. A standard dictionary, named dictionary.xml, is supplied in the installation folder of every connector. If a connector does not have any entries in the dictionary, field standardization has no effect.

Configure Field Standardization

IDOL Connectors have several configuration parameters that control field standardization. All of these are set in the [Connector] section of the configuration file:

- EnableFieldNameStandardization specifies whether to run field standardization.
- FieldNameDictionaryPath specifies the path of the dictionary file to use.
- FieldNameDictionaryNode specifies the rules to use. The default value for this parameter matches the name of the connector, and HPE recommends that you do not change it. This prevents one connector running field standardization rules that are intended for another.

To configure field standardization, use the following procedure.

> **NOTE:**
> You can also configure CFS to run field standardization. To standardize all field names, you must run field standardization from both the connector and CFS.

To enable field standardization

1. Stop the connector.
2. Open the connector's configuration file.
3. In the [Connector] section, set the following parameters:

`EnableFieldNameStandardization`	A Boolean value that specifies whether to enable field standardization. Set this parameter to **`true`**.
`FieldNameDictionaryPath`	The path to the dictionary file that contains the rules to use to standardize documents. A standard dictionary is included with the connector and is named `dictionary.xml`.

For example:

```
[Connector]
EnableFieldNameStandardization=true
FieldNameDictionaryPath=dictionary.xml
```

4. Save the configuration file and restart the connector.

Customize Field Standardization

Field standardization modifies documents so that they have a consistent structure and consistent field names. You can use field standardization so that documents indexed into IDOL through different connectors use the same fields to store the same type of information. Field standardization only modifies fields that are specified in a dictionary, which is defined in XML format. A standard dictionary, named `dictionary.xml`, is supplied in the installation folder of every connector.

In most cases you should not need to modify the standard dictionary, but you can modify it to suit your requirements or create dictionaries for different purposes. By modifying the dictionary, you can configure the connector to apply rules that modify documents before they are ingested. For example, you can move fields, delete fields, or change the format of field values.

The following examples demonstrate how to perform some operations with field standardization.

The following rule renames the field `Author` to `DOCUMENT_METADATA_AUTHOR_STRING`. This rule applies to all components that run field standardization and applies to all documents.

```
<FieldStandardization>
    <Field name="Author">
        <Move name="DOCUMENT_METADATA_AUTHOR_STRING"/>
    </Field>
</FieldStandardization>
```

The following rule demonstrates how to use the `Delete` operation. This rule instructs CFS to remove the field `KeyviewVersion` from all documents (the `Product` element with the attribute `key="ConnectorFrameWork"` ensures that this rule is run only by CFS).

```
<FieldStandardization>
    <Product key="ConnectorFrameWork">
        <Field name="KeyviewVersion">
            <Delete/>
        </Field>
    </Product>
</FieldStandardization>
```

There are several ways to select fields to process using the `Field` element.

Field element attribute	Description	Example

`name`	Select a field where the field name matches a fixed value.	Select the field `MyField`:
		`<Field name="MyField">`
		`...`
		`</Field>`
		Select the field `Subfield`, which is a subfield of `MyField`:
		`<Field name="MyField">`
		`<Field name="Subfield">`
		`...`
		`</Field>`
		`</Field>`

`path`	Select a field where its path matches a fixed value.	Select the field `Subfield`, which is a subfield of `MyField`.
		`<Field path="MyField/Subfield">`
		`...`
		`</Field>`

`nameRegex`	Select all fields at the current depth where the field name matches a regular expression.	In this case the field name must begin with the word `File`:
		`<Field nameRegex="File.*">`
		`...`
		`</Field>`

`pathRegex`	Select all fields where the path of the field matches a regular expression.
	This operation can be inefficient because every metadata field must be checked. If possible, select the fields to process another way.
		This example selects all subfields of `MyField`.
		`<Field pathRegex="MyField/[^/]*">`
		`...`
		`</Field>`
		This approach would be more efficient:
		`<Field name="MyField">`
		`<Field nameRegex=".*">`
		`...`
		`</Field>`
		`</Field>`

You can also limit the fields that are processed based on their value, by using one of the following:

Field element attribute	Description	Example
`matches`	Process a field if its value matches a fixed value.	Process a field named `MyField`, if its value matches `abc`.

		`<Field name="MyField" matches="abc">`
		`...`
		`</Field>`

`matchesRegex`	Process a field if its entire value matches a regular expression.	Process a field named `MyField`, if its value matches one or more digits.
		`<Field name="MyField" matchesRegex="\d+">`
		`...`
		`</Field>`

`containsRegex`	Process a field if its value contains a match to a regular expression.	Process a field named `MyField` if its value contains three consecutive digits.
		`<Field name="MyField" containsRegex="\d{3}">`
		`...`
		`</Field>`

The following rule deletes every field or subfield where the name of the field or subfield begins with `temp`.

```
<FieldStandardization>
    <Field pathRegex="(.*/)?temp[^/]*">
        <Delete/>
    </Field>
</FieldStandardization>
```

The following rule instructs CFS to rename the field `Author` to `DOCUMENT_METADATA_AUTHOR_STRING`, but only when the document contains a field named `DocumentType` with the value `230` (the KeyView format code for a PDF file).

```
<FieldStandardization>
    <Product key="ConnectorFrameWork">
        <IfField name="DocumentType" matches="230"> <!-- PDF -->
            <Field name="Author">
                <Move name="DOCUMENT_METADATA_AUTHOR_STRING"/>
            </Field>
        </IfField>
    </Product>
</FieldStandardization>
```

> **TIP:**
> In this example, the `IfField` element is used to check the value of the `DocumentType` field. The `IfField` element does not change the current position in the document. If you used the `Field` element, field standardization would attempt to find an `Author` field that is a subfield of `DocumentType`, instead of finding the `Author` field at the root of the document.

The following rules demonstrate how to use the `ValueFormat` operation to change the format of dates. The only format that you can convert date values into is the IDOL AUTNDATE format. The first rule transforms the value of a field named `CreatedDate`. The second rule transforms the value of an attribute named `Created`, on a field named `Date`.

```
<FieldStandardization>
    <Field name="CreatedDate">
        <ValueFormat type="autndate" format="YYYY-SHORTMONTH-DD HH:NN:SS"/>
    </Field>
    <Field name="Date">
        <Attribute name="Created">
            <ValueFormat type="autndate" format="YYYY-SHORTMONTH-DD HH:NN:SS"/>
        </Attribute>
    </Field>
</FieldStandardization>
```

As demonstrated by this example, you can select field attributes to process in a similar way to selecting fields.

You must select attributes using either a fixed name or a regular expression:

Select a field attribute by name	`<Attribute name="MyAttribute">`
Select attributes that match a regular expression	`<Attribute nameRegex=".*">`

You can then add a restriction to limit the attributes that are processed:

Process an attribute only if its value matches a fixed value	`<Attribute name="MyAttribute" matches="abc">`
Process an attribute only if its value matches a regular expression	`<Attribute name="MyAttribute" matchesRegex=".*">`
Process an attribute only if its value contains a match to a regular expression	`<Attribute name="MyAttribute" containsRegex="\w+">`

The following rule moves all of the attributes of a field to sub fields, if the parent field has no value. The `id` attribute on the first `Field` element provides a name to a matching field so that it can be referred to by later operations. The `GetName` and `GetValue` operations save the name and value of a selected field or attribute (in this case an attribute) into variables (in this case `$'name'` and `$'value'`) which can be used by later operations. The `AddField` operation uses the variables to add a new field at the selected location (the field identified by `id="parent"`).

```
<FieldStandardization>
    <Field pathRegex=".*" matches="" id="parent">
        <Attribute nameRegex=".*">
            <GetName var="name"/>
            <GetValue var="value"/>
            <Field fieldId="parent">
                <AddField name="$'name'" value="$'value'"/>
            </Field>
            <Delete/>
```

```
        </Attribute>
    </Field>
</FieldStandardization>
```

The following rule demonstrates how to move all of the subfields of `UnwantedParentField` to the root of the document, and then delete the field `UnwantedParentField`.

```
<FieldStandardization id="root">
    <Product key="MyConnector">
        <Field name="UnwantedParentField">
            <Field nameRegex=".*">
                <Move destId="root"/>
            </Field>
            <Delete/>
        </Field>
    </Product>
</FieldStandardization>
```

Run Lua Scripts

IDOL Connectors can run custom scripts written in Lua, an embedded scripting language. You can use Lua scripts to process documents that are created by a connector, before they are sent to CFS and indexed into IDOL Server. For example, you can:

- Add or modify document fields.
- Manipulate the information that is indexed into IDOL.
- Call out to an external service, for example to alert a user.

There might be occasions when you do not want to send documents to a CFS. For example, you might use the `Collect` action to retrieve documents from one repository and then insert them into another. You can use a Lua script to transform the documents from the source repository so that they can be accepted by the destination repository.

To run a Lua script from a connector, use one of the following methods:

- Set the `IngestActions` configuration parameter in the connector's configuration file. For information about how to do this, see Run a Lua Script using an Ingest Action, on page 75. The connector runs ingest actions on documents before they are sent for ingestion.
- Set the `IngestActions` action parameter when using the `Synchronize` action.
- Set the `InsertActions` configuration parameter in the connector's configuration file. The connector runs insert actions on documents before they are inserted into a repository.
- Set the `CollectActions` action parameter when using the `Collect` action.

Write a Lua Script

A Lua script that is run from a connector must have the following structure:

```
function handler(config, document, params)
	...
end
```

The `handler` function is called for each document and is passed the following arguments:

Argument	Description
`config`	A `LuaConfig` object that you can use to retrieve the values of configuration parameters from the connector's configuration file.
`document`	A `LuaDocument` object. The document object is an internal representation of the document being processed. Modifying this object changes the document.
`params`	The `params` argument is a table that contains additional information provided by the connector: • **TYPE**. The type of task being performed. The possible values are `ADD`, `UPDATE`, `DELETE`, or `COLLECT`. • **SECTION**. The name of the section in the configuration file that contains configuration parameters for the task. • **FILENAME**. The document filename. The Lua script can modify this file, but must not delete it. • **OWNFILE**. Indicates whether the connector (and CFS) has ownership of the file. A value of `true` means that CFS deletes the file after it has been processed.

The following script demonstrates how you can use the `config` and `params` arguments:

```
function handler(config, document, params)
	-- Write all of the additional information to a log file
	for k,v in pairs(params) do
		log("logfile.txt", k..": "..tostring(v))
	end

	-- The following lines set variables from the params argument
	type = params["TYPE"]
	section = params["SECTION"]
	filename = params["FILENAME"]

	-- Read a configuration parameter from the configuration file
	-- If the parameter is not set, "DefaultValue" is returned
	val = config:getValue(section, "Parameter", "DefaultValue")

	-- If the document is not being deleted, set the field FieldName
	-- to the value of the configuration parameter
	if type ~= "DELETE" then
		document:setFieldValue("FieldName", val)
	end

	-- If the document has a file (that is, not just metadata),
	-- copy the file to a new location and write a stub idx file
```

```
        -- containing the metadata.
        if filename ~= "" then
                copytofilename = "./out/"..create_uuid(filename)
                copy_file(filename, copytofilename)
                document:writeStubIdx(copytofilename..".idx")
        end

        return true
end
```

For the connector to continue processing the document, the `handler` function must return **`true`**. If the function returns **`false`**, the document is discarded.

> **TIP:**
> You can write a library of useful functions to share between multiple scripts. To include a library of functions in a script, add the code **`dofile("library.lua")`** to the top of the lua script, outside of the `handler` function.

Run a Lua Script using an Ingest Action

To run a Lua script on documents that are sent for ingestion, use an Ingest Action.

To run a Lua script using an Ingest Action

1. Open the connector's configuration file.
2. Find one of the following sections in the configuration file:
   - To run a Lua script on all documents retrieved by a specific task, find the `[*TaskName*]` section.
   - To run a Lua script on all documents that are sent for ingestion, find the `[Ingestion]` section.

   > **NOTE:**
   > If you set the `IngestActions` parameter in a `[TaskName]` section, the connector does not run any `IngestActions` set in the `[Ingestion]` section for that task.
3. Use the `IngestActions` parameter to specify the path to your Lua script. For example:

   `IngestActions=LUA:C:\Autonomy\myScript.lua`
4. Save and close the configuration file.

Related Topics

- Write a Lua Script, on page 73

Example Lua Scripts

This section contains example Lua scripts.

- Add a Field to a Document, on the next page
- Merge Document Fields, on the next page

Add a Field to a Document

The following script demonstrates how to add a field named “MyField” to a document, with a value of “MyValue”.

```
function handler(config, document, params)
       document:addField("MyField", "MyValue");
       return true;
end
```

The following script demonstrates how to add the field `AUTN_NEEDS_MEDIA_SERVER_ANALYSIS` to all JPEG, TIFF and BMP documents. This field indicates to CFS that the file should be sent to a Media Server for analysis (you must also define the `MediaServerAnalysis` task in the CFS configuration file).

The script finds the file type using the `DREREFERENCE` document field, so this field must contain the file extension for the script to work correctly.

```
function handler(config, document, params)
       local extensions_for_ocr = { jpg = 1 , tif = 1, bmp = 1 };
       local filename = document:getFieldValue("DREREFERENCE");
       local extension, extension_found = filename:gsub("^.*%.(%w+)$", "%1", 1);

       if extension_found > 0 then
               if extensions_for_ocr[extension:lower()] ~= nil then
                  document:addField("AUTN_NEEDS_MEDIA_SERVER_ANALYSIS", "");
               end
       end

       return true;
end
```

Merge Document Fields

This script demonstrates how to merge the values of document fields.

When you extract data from a repository, the connector can produce documents that have multiple values for a single field, for example:

```
#DREFIELD ATTACHMENT="attachment.txt"
#DREFIELD ATTACHMENT="image.jpg"
#DREFIELD ATTACHMENT="document.pdf"
```

This script shows how to merge the values of these fields, so that the values are contained in a single field, for example:

```
#DREFIELD ATTACHMENTS="attachment.txt, image.jpg, document.pdf"
```

Example Script

```
function handler(config, document, params)
	onefield(document,"ATTACHMENT","ATTACHMENTS")
	return true;
end

function onefield(document,existingfield,newfield)
	if document:hasField(existingfield) then
		local values = { document:getFieldValues(existingfield) }

		local newfieldvalue=""
		for i,v in ipairs(values) do
		    if i>1 then
		        newfieldvalue = newfieldvalue ..", "
		    end

		    newfieldvalue = newfieldvalue..v
		end

		document:addField(newfield,newfieldvalue)
	end

	return true;
end
```

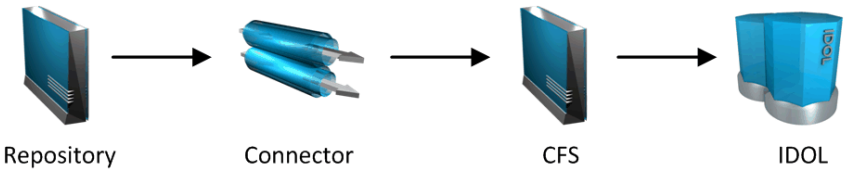
Chapter 9: Ingestion

After a connector finds new documents in a repository, or documents that have been updated or deleted, it sends this information to another component called the *ingestion target*. This section describes where you can send the information retrieved by the HPE File System Connector, and how to configure the ingestion target.

Introduction

A connector can send information to a single ingestion target, which could be:

- **Connector Framework Server**. To process information and then index it into IDOL, Haven OnDemand, or Vertica, send the information to a Connector Framework Server (CFS). Any files retrieved by the connector are *imported* using KeyView, which means the information contained in the files is converted into a form that can be indexed. If the files are containers that contain *subfiles*, these are extracted. You can manipulate and enrich documents using Lua scripts and automated tasks such as field standardization, image analysis, and speech-to-text processing. CFS can index your documents into one or more indexes. For more information about CFS, refer to the *Connector Framework Server Administration Guide*.



- **Haven OnDemand**. You can index documents directly into a Haven OnDemand text index. Haven OnDemand can extract text, metadata, and subfiles from over 1000 different file formats, so you might not need to send documents to CFS.
- **Another Connector.** Use another connector to keep another repository up-to-date. When a connector receives documents, it inserts, updates, or deletes the information in the repository. For example, you could use an Exchange Connector to extract information from Microsoft Exchange, and send the documents to a Notes Connector so that the information is inserted, updated, or deleted in the Notes repository.

> **NOTE:**
> The destination connector can only insert, update, and delete documents if it supports the `insert`,

> `update`, and `delete` fetch actions.

In most cases HPE recommends ingesting documents through CFS, so that KeyView can extract content from any files retrieved by the connector and add this information to your documents. You can also use CFS to manipulate and enrich documents before they are indexed. However, if required you can configure the connector to index documents directly into:

- **IDOL Server**. You might index documents directly into IDOL Server when your connector produces metadata-only documents (documents that do not have associated files). In this case there is no need for the documents to be imported. Connectors that can produce metadata-only documents include ODBC Connector and Oracle Connector.
- **Vertica.** The metadata extracted by connectors is structured information held in structured fields, so you might use Vertica to analyze this information.
- **MetaStore**. You can index document metadata into a MetaStore for records management.

Send Data to Connector Framework Server

This section describes how to configure ingestion into Connector Framework Server (CFS).

To send data to a CFS

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. In the `[Ingestion]` section, set the following parameters:

`EnableIngestion`	To enable ingestion, set this parameter to **`true`**.
`IngesterType`	To send data to CFS, set this parameter to **`CFS`**.
`IngestHost`	The host name or IP address of the CFS.
`IngestPort`	The ACI port of the CFS.

   For example:

   ```
   [Ingestion]
   EnableIngestion=True
   IngesterType=CFS
   IngestHost=localhost
   IngestPort=7000
   ```

4. (Optional) If you are sending documents to CFS for indexing into IDOL Server, set the `IndexDatabase` parameter. When documents are indexed, IDOL adds each document to the database specified in the document's `DREDBNAME` field. The connector sets this field for each document, using the value of `IndexDatabase`.

`IndexDatabase`	The name of the IDOL database into which documents are indexed. Ensure that this database exists in the IDOL Server configuration file.

   - To index all documents retrieved by the connector into the same IDOL database, set this parameter in the `[Ingestion]` section.

- To use a different database for documents retrieved by each task, set this parameter in the *TaskName* section.

5. Save and close the configuration file.

Send Data to Haven OnDemand

This section describes how to configure ingestion into Haven OnDemand. HPE File System Connector can index documents into a Haven OnDemand text index, or send the documents to a Haven OnDemand combination which can perform additional processing and then index the documents into a text index.

> **NOTE:**
> Haven OnDemand combinations do not accept binary files, so any documents that have associated binary files are indexed directly into a text index and cannot be sent to a combination.

Prepare Haven OnDemand

Before you can send documents to Haven OnDemand, you must create a text index. For information about how to create text indexes, refer to the Haven OnDemand documentation.

Before you can send documents to a Haven OnDemand combination endpoint, the combination must exist. HPE File System Connector requires your combination to accept the following input parameters, and produce the following output.

Input Parameters

Name	Type	Description
`json`	any	A JSON object that contains a single attribute 'documents' that is an array of document objects.
`index`	string	The name of the text index that you want the combination to add documents to. HPE File System Connector uses the value of the parameter `HavenOnDemandIndexName` to set this value.
`duplicate_mode`	string	Specifies how to handle duplicates when adding documents to the text index. HPE File System Connector uses the value of the parameter `HavenOnDemandDuplicateMode` to set this value.

Output

Name	Type	Description
`result`	any	The result of the call to `AddToTextIndex` made by the combination.

Send Data to Haven OnDemand

This section describes how to send documents to Haven OnDemand.

To send data to Haven OnDemand

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. In the `[Ingestion]` section, set the following parameters:

`EnableIngestion`	To enable ingestion, set this parameter to **`true`**.
`IngesterType`	To send data to Haven OnDemand, set this parameter to **`HavenOnDemand`**.
`HavenOnDemandApiKey`	Your Haven OnDemand API key. You can obtain the key from your Haven OnDemand account.
`HavenOnDemandIndexName`	The name of the Haven OnDemand text index to index documents into.
`IngestSSLConfig`	The name of a section in the connector's configuration file that contains SSL settings. The connection to Haven OnDemand must be made over TLS. For more information about sending documents to the ingestion server over TLS, see Configure Outgoing SSL Connections, on page 34.
`HavenOnDemandCombinationName`	(Optional) The name of the Haven OnDemand combination to send documents to. If you set this parameter, HPE File System Connector sends documents to the combination endpoint instead of indexing them directly into the text index. **NOTE:** Haven OnDemand combinations do not accept binary files. Therefore any document that has an associated binary file is indexed directly into the text index. If you don't set this parameter, HPE File System Connector indexes all documents directly into the text index specified by `HavenOnDemandIndexName`.

For example:

```
[Ingestion]
EnableIngestion=True
IngesterType=HavenOnDemand
HavenOnDemandApiKey=[Your API Key]
HavenOnDemandIndexName=MyTextIndex
IngestSSLConfig=SSLOptions
HavenOnDemandCombinationName=MyCombination
```

```
[SSLOptions]
SSLMethod=TLSV1
```

4. Save and close the configuration file.

Send Data to Another Repository

You can configure a connector to send the information it retrieves to another connector. When the destination connector receives the documents, it inserts them into another repository. When documents are updated or deleted in the source repository, the source connector sends this information to the destination connector so that the documents can be updated or deleted in the other repository.

> **NOTE:**
> The destination connector can only insert, update, and delete documents if it supports the `insert`, `update`, and `delete` fetch actions.

To send data to another connector for ingestion into another repository

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. In the `[Ingestion]` section, set the following parameters:

`EnableIngestion`	To enable ingestion, set this parameter to **`true`**.
`IngesterType`	To send data to another repository, set this parameter to **`Connector`**.
`IngestHost`	The host name or IP address of the machine hosting the destination connector.
`IngestPort`	The ACI port of the destination connector.
`IngestActions`	Set this parameter so that the source connector runs a Lua script to convert documents into form that can be used with the destination connector's insert action. For information about the required format, refer to the Administration Guide for the destination connector.

For example:

```
[Ingestion]
EnableIngestion=True
IngesterType=Connector
IngestHost=AnotherConnector
IngestPort=7010
IngestActions=Lua:transformation.lua
```

4. Save and close the configuration file.

Index Documents Directly into IDOL Server

This section describes how to index documents from a connector directly into IDOL Server.

> **TIP:**
> In most cases, HPE recommends sending documents to a Connector Framework Server (CFS). CFS extracts metadata and content from any files that the connector has retrieved, and can manipulate and enrich documents before they are indexed. CFS also has the capability to insert documents into more than one index, for example IDOL Server and a Vertica database. For information about sending documents to CFS, see Send Data to Connector Framework Server, on page 79

To index documents directly into IDOL Server

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. In the `[Ingestion]` section, set the following parameters:

`EnableIngestion`	To enable ingestion, set this parameter to **`true`**.
`IngesterType`	To send data to IDOL Server, set this parameter to **`Indexer`**.
`IndexDatabase`	The name of the IDOL database to index documents into.

   For example:

   ```
   [Ingestion]
   EnableIngestion=True
   IngesterType=Indexer
   IndexDatabase=News
   ```

4. In the `[Indexing]` section of the configuration file, set the following parameters:

`IndexerType`	To send data to IDOL Server, set this parameter to **`IDOL`**.
`Host`	The host name or IP address of the IDOL Server.
`Port`	The IDOL Server ACI port.
`SSLConfig`	(Optional) The name of a section in the connector's configuration file that contains SSL settings for connecting to IDOL.

   For example:

   ```
   [Indexing]
   IndexerType=IDOL
   Host=10.1.20.3
   Port=9000
   SSLConfig=SSLOptions
   ```

```
[SSLOptions]
SSLMethod=SSLV23
```

5. Save and close the configuration file.

Index Documents into Vertica

HPE File System Connector can index documents into Vertica, so that you can run queries on structured fields (document metadata).

Depending on the metadata contained in your documents, you could investigate the average age of documents in a repository. You might want to answer questions such as: How much time has passed since the documents were last updated? How many files are regularly updated? Does this represent a small proportion of the total number of documents? Who are the most active users?

> **TIP:**
> In most cases, HPE recommends sending documents to a Connector Framework Server (CFS). CFS extracts metadata and content from any files that the connector has retrieved, and can manipulate and enrich documents before they are indexed. CFS also has the capability to insert documents into more than one index, for example IDOL Server and a Vertica database. For information about sending documents to CFS, see Send Data to Connector Framework Server, on page 79

Prerequisites

- HPE File System Connector supports indexing into Vertica 7.1 and later.
- You must install the appropriate Vertica ODBC drivers (version 7.1 or later) on the machine that hosts HPE File System Connector. If you want to use an ODBC Data Source Name (DSN) in your connection string, you will also need to create the DSN. For more information about installing Vertica ODBC drivers and creating the DSN, refer to the HPE Vertica documentation.

New, Updated and Deleted Documents

When documents are indexed into Vertica, HPE File System Connector adds a timestamp that contains the time when the document was indexed. The field is named `VERTICA_INDEXER_TIMESTAMP` and the timestamp is in the format `YYYY-MM-DD HH:NN:SS`.

When a document in a data repository is modified, HPE File System Connector adds a new record to the database with a new timestamp. All of the fields are populated with the latest data. The record describing the older version of the document is not deleted. You can create a projection to make sure your queries only return the latest record for a document.

When HPE File System Connector detects that a document has been deleted from a repository, the connector inserts a new record into the database. The record contains only the `DREREFERENCE` and the field `VERTICA_INDEXER_DELETED` set to `TRUE`.

Fields, Sub-Fields, and Field Attributes

Documents that are created by connectors can have multiple levels of fields, and field attributes. A database table has a flat structure, so this information is indexed into Vertica as follows:

- Document fields become columns in the flex table. An IDOL document field and the corresponding database column have the same name.
- Sub-fields become columns in the flex table. A document field named `my_field` with a sub-field named `subfield` results in two columns, `my_field` and `my_field.subfield`.
- Field attributes become columns in the flex table. A document field named `my_field`, with an attribute named `my_attribute` results in two columns, `my_field` holding the field value and `my_field.my_attribute` holding the attribute value.

Prepare the Vertica Database

Indexing documents into a standard database is problematic, because documents do not have a fixed schema. A document that represents an image has different metadata fields to a document that represents an e-mail message. Vertica databases solve this problem with *flex tables*. You can create a flex table without any column definitions, and you can insert a record regardless of whether a referenced column exists.

You must create a flex table before you index data into Vertica.

When creating the table, consider the following:

- Flex tables store entire records in a single column named `__raw__`. The default maximum size of the `__raw__` column is 128K. You might need to increase the maximum size if you are indexing documents with large amounts of metadata.
- Documents are identified by their `DREREFERENCE`. HPE recommends that you do not restrict the size of any column that holds this value, because this could result in values being truncated. As a result, rows that represent different documents might appear to represent the same document. If you do restrict the size of the `DREREFERENCE` column, ensure that the length is sufficient to hold the longest `DREREFERENCE` that might be indexed.

To create a flex table without any column definitions, run the following query:

```
create flex table my_table();
```

To improve query performance, create real columns for the fields that you query frequently. For documents indexed by a connector, this is likely to include the `DREREFERENCE`:

```
create flex table my_table(DREREFERENCE varchar NOT NULL);
```

You can add new column definitions to a flex table at any time. Vertica automatically populates new columns with values for existing records. The values for existing records are extracted from the `__raw__` column.

For more information about creating and using flex tables, refer to the HPE Vertica Documentation or contact HPE Vertica technical support.

Send Data to Vertica

To send documents to a Vertica database, follow these steps.

To send data to Vertica

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. In the `[Ingestion]` section, set the following parameters:

`EnableIngestion`	To enable ingestion, set this parameter to **`true`**.
`IngesterType`	To send data to a Vertica database, set this parameter to **`Indexer`**.

   For example:

   ```
   [Ingestion]
   EnableIngestion=TRUE
   IngesterType=Indexer
   ```

4. In the `[Indexing]` section, set the following parameters:

`IndexerType`	To send data to a Vertica database, set this parameter to **`Library`**.
`LibraryDirectory`	The directory that contains the library to use to index data.
`LibraryName`	The name of the library to use to index data. You can omit the `.dll` or `.so` file extension. Set this parameter to **`verticaIndexer`**.
`ConnectionString`	The connection string to use to connect to the Vertica database.
`TableName`	The name of the table in the Vertica database to index the documents into. The table must be a flex table and must exist before you start indexing documents. For more information, see Prepare the Vertica Database, on the previous page.

   For example:

   ```
   [Indexing]
   IndexerType=Library
   LibraryDirectory=indexerdlls
   LibraryName=verticaIndexer
   ConnectionString=DSN=VERTICA
   TableName=my_flex_table
   ```

5. Save and close the configuration file.

Send Data to a MetaStore

You can configure a connector to send documents to a MetaStore. When you send data to a Metastore, any files associated with documents are ignored.

TIP:
In most cases, HPE recommends sending documents to a Connector Framework Server (CFS). CFS extracts metadata and content from any files that the connector has retrieved, and can manipulate and enrich documents before they are indexed. CFS also has the capability to insert documents into more than one index, for example IDOL Server and a MetaStore. For information about sending documents to CFS, see Send Data to Connector Framework Server, on page 79

To send data to a MetaStore

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. In the `[Ingestion]` section, set the following parameters:

   `EnableIngestion` To enable ingestion, set this parameter to **`true`**.

   `IngesterType` To send data to a MetaStore, set this parameter to **`Indexer`**.

   For example:

   ```
   [Ingestion]
   EnableIngestion=True
   IngesterType=Indexer
   ```

4. In the `[Indexing]` section, set the following parameters:

   `IndexerType` To send data to a MetaStore, set this parameter to **`MetaStore`**.

   `Host` The host name of the machine hosting the MetaStore.

   `Port` The port of the MetaStore.

   For example:

   ```
   [Indexing]
   IndexerType=Metastore
   Host=MyMetaStore
   Port=8000
   ```

5. Save and close the configuration file.

Run a Lua Script after Ingestion

You can configure the connector to run a Lua script after batches of documents are successfully sent to the ingestion server. This can be useful if you need to log information about documents that were processed, for monitoring and reporting purposes.

To configure the file name of the Lua script to run, set the `IngestBatchActions` configuration parameter in the connector's configuration file.

- To run the script for all batches of documents that are ingested, set the parameter in the `[Ingestion]` section.
- To run the script for batches of documents retrieved by a specific task, set the parameter in the

`[TaskName]` section.

> **NOTE:**
> If you set the parameter in a `[TaskName]` section, the connector does not run any scripts specified in the `[Ingestion]` section for that task.

For example:

```
[Ingestion]
IngestBatchActions0=LUA:./scripts/myScript.lua
```

For more information about this parameter, refer to the *HPE File System Connector Reference*.

The Lua script must have the following structure:

```
function batchhandler(documents, ingesttype)
   ...
end
```

The `batchhandler` function is called after each batch of documents is sent to the ingestion server. The function is passed the following arguments:

Argument	Description
`documents`	A table of document objects, where each object represents a document that was sent to the ingestion server. A document object is an internal representation of a document. You can modify the document object and this changes the document. However, as the script runs after the documents are sent to the ingestion server, any changes you make are not sent to CFS or IDOL.
`ingesttype`	A string that contains the ingest type for the documents. The `batchhandler` function is called multiple times if different document types are sent.

For example, the following script prints the ingest type (`ADD`, `DELETE`, or `UPDATE`) and the reference for all successfully processed documents to `stdout`:

```
function batchhandler(documents, ingesttype)
     for i,document in ipairs(documents) do
         local ref = document:getReference()
         print(ingesttype..": "..ref)
     end
 end
```

Chapter 10: Monitor the Connector

This section describes how to monitor the connector.

IDOL Admin

IDOL Admin is an administration interface for performing ACI server administration tasks, such as gathering status information, monitoring performance, and controlling the service. IDOL Admin provides an alternative to constructing actions and sending them from your web browser.

Prerequisites

By default, the latest version of HPE File System Connector should include the `admin.dat` file that is required to run IDOL Admin. If you do not have this file, you must download it separately.

Supported Browsers

IDOL Admin supports the following browsers:

- Internet Explorer 11 and later
- Edge
- Chrome (latest version)
- Firefox (latest version)

Install IDOL Admin

You must install IDOL Admin on the same host that the ACI server or component is installed on. To set up a component to use IDOL Admin, you must configure the location of the `admin.dat` file and enable Cross Origin Resource Sharing.

To install IDOL Admin

1. Stop the ACI server.
2. Save the `admin.dat` file to any directory on the host.

3. Using a text editor, open the ACI server or component configuration file. For the location of the configuration file, see the ACI server documentation.
4. In the `[Paths]` section of the configuration file, set the `AdminFile` parameter to the location of the `admin.dat` file. If you do not set this parameter, the ACI server attempts to find the `admin.dat` file in its working directory when you call the IDOL Admin interface.
5. Enable Cross Origin Resource Sharing.
6. In the `[Service]` section, add the `Access-Control-Allow-Origin` parameter and set its value to the URLs that you want to use to access the interface.

   Each URL must include:

   - the `http://` or `https://` prefix

   > **NOTE:**
   > URLs can contain the `https://` prefix if the ACI server or component has SSL enabled.

   - The host that IDOL Admin is installed on
   - The ACI port of the component that you are using IDOL Admin for

   Separate multiple URLs with spaces.

   For example, you could specify different URLs for the local host and remote hosts:

   ```
   Access-Control-Allow-Origin=http://localhost:9010
   http://Computer1.Company.com:9010
   ```

   Alternatively, you can set `Access-Control-Allow-Origin=*`, which allows you to access IDOL Admin using any valid URL (for example, `localhost`, direct IP address, or the host name). The wildcard character (*) is supported only if no other entries are specified.

   If you do not set the `Access-Control-Allow-Origin` parameter, IDOL Admin can communicate only with the server's ACI port, and not the index or service ports.
7. Start the ACI server.

You can now access IDOL Admin (see Access IDOL Admin, below).

Access IDOL Admin

You access IDOL Admin from a web browser. You can access the interface only through URLs that are set in the `Access-Control-Allow-Origin` parameter in the ACI server or component configuration file. For more information about configuring URL access, see Install IDOL Admin, on the previous page.

To access IDOL Admin from the host that it is installed on

- Type the following URL into the address bar of your web browser:

  `http://localhost:`*`port`*`/action=admin`

  where *`port`* is the ACI server or component ACI port.

To access IDOL Admin from a different host

- Type the following URL into the address bar of your web browser:

  `http://`*`host`*`:`*`port`*`/action=admin`

where:

host is the name or IP address of the host that IDOL Admin is installed on.

port is the ACI server or component ACI port of the IDOL Admin host.

Use the Connector Logs

As the HPE File System Connector runs, it outputs messages to its logs. Most log messages occur due to normal operation, for example when the connector starts, receives actions, or sends documents for ingestion. If the connector encounters an error, the logs are the first place to look for information to help troubleshoot the problem.

The connector separates messages into the following message types, each of which relates to specific features:

Log Message Type	Description
`Action`	Logs actions that are received by the connector, and related messages.
`Application`	Logs application-related occurrences, such as when the connector starts.
`Collect`	Messages related to the `Collect` fetch action.
`Delete`	Messages related to the `Delete` fetch action.
`Identifiers`	Messages related to the `Identifiers` fetch action.
`Insert`	Messages related to the `Insert` fetch action.
`Synchronize`	Messages related to the `Synchronize` fetch action.
`Update`	Messages related to the `Update` fetch action.
`View`	Messages related to the `View` action.

Customize Logging

You can customize logging by setting up your own *log streams*. Each log stream creates a separate log file in which specific log message types (for example, action, index, application, or import) are logged.

To set up log streams

1. Open the HPE File System Connector configuration file in a text editor.
2. Find the `[Logging]` section. If the configuration file does not contain a `[Logging]` section, add one.
3. In the `[Logging]` section, create a list of the log streams that you want to set up, in the format *N=LogStreamName*. List the log streams in consecutive order, starting from 0 (zero). For example:

```
[Logging]
LogLevel=FULL
LogDirectory=logs
0=ApplicationLogStream
1=ActionLogStream
```

You can also use the `[Logging]` section to configure any default values for logging configuration parameters, such as `LogLevel`. For more information, see the *HPE File System Connector Reference*.

4. Create a new section for each of the log streams. Each section must have the same name as the log stream. For example:

```
[ApplicationLogStream]
[ActionLogStream]
```

5. Specify the settings for each log stream in the appropriate section. You can specify the type of logging to perform (for example, full logging), whether to display log messages on the console, the maximum size of log files, and so on. For example:

```
[ApplicationLogStream]
LogTypeCSVs=application
LogFile=application.log
LogHistorySize=50
LogTime=True
LogEcho=False
LogMaxSizeKBs=1024

[ActionLogStream]
LogTypeCSVs=action
LogFile=logs/action.log
LogHistorySize=50
LogTime=True
LogEcho=False
LogMaxSizeKBs=1024
```

6. Save and close the configuration file. Restart the service for your changes to take effect.

Monitor the Progress of a Task

This section describes how to monitor the progress of a task.

> **NOTE:**
> You can only monitor the progress of a synchronize task. Progress reports are not available for other actions.

To monitor the progress of a task

- Send the following action to the connector:

```
action=QueueInfo&QueueName=fetch&QueueAction=progress&Token=...
```

where,

`Token` The token of the task that you want to monitor. If you started the task by sending an action to the connector, the token was returned in the response. If the connector started the task according to the schedule in its configuration file, you can use the `QueueInfo` action to find the token (use `/action=QueueInfo&QueueName=fetch&QueueAction=getstatus`).

The connector returns the progress report, inside the `<progress>` element of the response. The following example is from a File System Connector.

```
<autnresponse>
  <action>QUEUEINFO</action>
  <response>SUCCESS</response>
  <responsedata>
    <action>
       <token>MTAuMi4xMDUuMTAzOjEyMzQ6RkVUQ0g6MTAxNzM0MzgzOQ==</token>
       <status>Processing</status>
       <progress>
          <building_mode>false</building_mode>
          <percent>7.5595</percent>
          <time_processing>18</time_processing>
          <estimated_time_remaining>194</estimated_time_remaining>
          <stage title="MYTASK" status="Processing" weight="1" percent="7.5595">
             <stage title="Ingestion" status="Processing" weight="999"
percent="7.567">
                <stage title="C:\Test Files\" status="Processing" weight="6601"
percent="7.567" progress="0" maximum="6601">
                   <stage title="Folder01" status="Processing" weight="2317"
percent="43.116" progress="999" maximum="2317"/>
                   <stage title="Folder02" status="Pending" weight="2567"/>
                   <stage title="Folder03" status="Pending" weight="1715"/>
                   <stage title="." status="Pending" weight="2"/>
                </stage>
             </stage>
             <stage title="Deletion" status="Pending" weight="1"/>
          </stage>
       </progress>
    </action>
  </responsedata>
</autnresponse>
```

To read the progress report

The information provided in the progress report is unique to each connector. For example, the File System Connector reports the progress of a synchronize task by listing the folders that require processing.

The progress report supplied by the connector can include several *stages*:

- A *stage* represents part of a task.
- A stage can have sub-stages. In the previous example, the stage "C:\Test Files\" has three stages that represent sub-folders ("Folder01", "Folder02", and "Folder03") and one stage that represents the

contents of the folder itself ("."). You can limit the depth of the sub-stages in the progress report by setting the `MaxDepth` parameter in the `QueueInfo` action.

- The `weight` attribute indicates the amount of work included in a stage, relative to other stages at the same level.
- The `status` attribute shows the status of a stage. The status can be "Pending", "Processing", or "Finished".
- The `progress` attribute shows the number of items that have been processed for the stage.
- The `maximum` attribute shows the total number of items that must be processed to complete the stage.
- The `percent` attribute shows the progress of a stage (percentage complete). In the previous example, the progress report shows that MYTASK is 7.5595% complete.
- Finished stages are grouped, and pending stages are not expanded into sub-stages, unless you set the action parameter `AllStages=true` in the `QueueInfo` action.

Set Up Event Handlers

The fetch actions sent to a connector are asynchronous. Asynchronous actions do not run immediately, but are added to a queue. This means that the person or application that sends the action does not receive an immediate response. However, you can configure the connector to call an event handler when an asynchronous action starts, finishes, or encounters an error.

You can use an event handler to:

- return data about an event back to the application that sent the action.
- write event data to a text file, to log any errors that occur.

The connector can call an event handler for the following events:

OnStart	The `OnStart` event is called when the connector starts processing an asynchronous action.
OnFinish	The `OnFinish` event is called when the connector successfully finishes processing an asynchronous action.
OnError	The `OnError` event is called when an asynchronous action fails and cannot continue.
OnErrorReport	The `OnErrorReport` event is called when an asynchronous action encounters an error, but the action continues. This event is not available for every connector.

Event Handlers

You can configure the connector to call an internal event handler, or write your own event handler. Connectors include the following internal event handlers.

TextFileHandler

The TextFileHandler writes event data to a text file.

HttpHandler

The HttpHandler sends event data to a URL.

LuaHandler

The LuaHandler runs a Lua script. The event data is passed into the script. The script must have the following form:

```
function handler(request, xml)
       ...
end
```

- `request` is a table holding the request parameters.
- `xml` is a string holding the response to the request.

Configure an Event Handler

To configure an event handler, follow these steps.

To configure an event handler

1. Stop the connector.
2. Open the connector's configuration file in a text editor.
3. Use the `OnStart`, `OnFinish`, `OnErrorReport`, or `OnError` parameter to specify the name of a section in the configuration file that contains event handler settings for the corresponding event.
   - To run an event handler for all actions, set these parameters in the `[Actions]` section. For example:

     ```
     [Actions]
     OnStart=NormalEvents
     OnFinish=NormalEvents
     OnErrorReport=ErrorEvents
     OnError=ErrorEvents
     ```

   - To run an event handler for specific actions, use the action name as a section in the configuration file. The following example runs an event handler when the *Fetch* action starts and finishes successfully:

     ```
     [Fetch]
     OnStart=NormalEvents
     OnFinish=NormalEvents
     ```

4. Create a new section in the configuration file to contain the settings for your event handler. You must name the section using the name you specified with the `OnStart`, `OnFinish`, `OnErrorReport`, or `OnError` parameter.
5. In the new section, set the following parameters.

   `LibraryName` (Required) The name of the library to use as the event handler. You can write your own event handler, or use one of the internal event handlers:
   - To write event data to a text file, set this parameter to **`TextFileHandler`**, and then set the `FilePath` parameter to specify the path of the file.
   - To send event data to a URL, set this parameter to **`HttpHandler`**, and then use the HTTP event handler parameters to specify the URL, proxy server settings, credentials and so on.
   - To run a Lua script, set this parameter to **`LuaHandler`**, and then set the `LuaScript` parameter to specify the path to the Lua script.

   For example:

   ```
   [NormalEvents]
   LibraryName=TextFileHandler
   FilePath=./events.txt

   [ErrorEvents]
   LibraryName=LuaHandler
   LuaScript=./error.lua
   ```

6. Save and close the configuration file.

Set Up Performance Monitoring

You can configure a connector to pause tasks temporarily if performance indicators on the local machine or a remote machine breach certain limits. For example, if there is a high load on the CPU or memory of the repository from which you are retrieving information, you might want the connector to pause until the machine recovers.

> **NOTE:**
> Performance monitoring is available on Windows platforms only. To monitor a remote machine, both the connector machine and remote machine must be running Windows.

Configure the Connector to Pause

To configure the connector to pause

1. Open the configuration file in a text editor.
2. Find the `[FetchTasks]` section, or a `[`*`TaskName`*`]` section.
   - To pause all tasks, use the `[FetchTasks]` section.
   - To specify settings for a single task, find the `[`*`TaskName`*`]` section for the task.

3. Set the following configuration parameters:

`PerfMonCounterNameN`	The names of the performance counters that you want the connector to monitor. You can use any counter that is available in the Windows `perfmon` utility.
`PerfMonCounterMinN`	The minimum value permitted for the specified performance counter. If the counter falls below this value, the connector pauses until the counter meets the limits again.
`PerfMonCounterMaxN`	The maximum value permitted for the specified performance counter. If the counter exceeds this value, the connector pauses until the counter meets the limits again.
`PerfMonAvgOverReadings`	(Optional) The number of readings that the connector averages before checking a performance counter against the specified limits. For example, if you set this parameter to `5`, the connector averages the last five readings and pauses only if the average breaches the limits. Increasing this value makes the connector less likely to pause if the limits are breached for a short time. Decreasing this value allows the connector to continue working faster following a pause.
`PerfMonQueryFrequency`	(Optional) The amount of time, in seconds, that the connector waits between taking readings from a performance counter.

For example:

```
[FetchTasks]
PerfMonCounterName0=\\machine-hostname\Memory\Available MBytes
PerfMonCounterMin0=1024
PerfMonCounterMax0=1024000
PerfMonCounterName1=\\machine-hostname\Processor(_Total)\% Processor Time
PerfMonCounterMin1=0
PerfMonCounterMax1=70
PerfMonAvgOverReadings=5
PerfMonQueryFrequency=10
```

> **NOTE:**
> You must set both a minimum and maximum value for each performance counter. You can not set only a minimum or only a maximum.

4. Save and close the configuration file.

Determine if an Action is Paused

To determine whether an action has been paused for performance reasons, use the `QueueInfo` action:

```
/action=queueInfo&queueAction=getStatus&queueName=fetch
```

You can also include the optional `token` parameter to return information about a single action:

```
/action=queueInfo&queueAction=getStatus&queueName=fetch&token=...
```

The connector returns the status, for example:

```
<autnresponse>
  <action>QUEUEINFO</action>
  <response>SUCCESS</response>
  <responsedata>
    <actions>
      <action owner="2266112570">
        <status>Processing</status>
        <queued_time>2016-Jul-27 14:49:40</queued_time>
        <time_in_queue>1</time_in_queue>
        <process_start_time>2016-Jul-27 14:49:41</process_start_time>
        <time_processing>219</time_processing>
        <documentcounts>
          <documentcount errors="0" task="MYTASK"/>
        </documentcounts>
        <fetchaction>SYNCHRONIZE</fetchaction>
        <pausedforperformance>true</pausedforperformance>
        <token>...</token>
      </action>
    </actions>
  </responsedata>
</autnresponse>
```

When the element `pausedforperformance` has a value of `true`, the connector has paused the task for performance reasons. If the `pausedforperformance` element is not present in the response, the connector has not paused the task.

Set Up Document Tracking

Document tracking reports metadata about documents when they pass through various stages in the indexing process. For example, when a connector finds a new document and sends it for ingestion, a document tracking event is created that shows the document has been added. Document tracking can help you detect problems with the indexing process.

You can write document tracking events to a database, log file, or IDOL Server. For information about how to set up a database to store document tracking events, refer to the *IDOL Server Administration Guide*.

To enable Document Tracking

1. Open the connector's configuration file.
2. Create a new section in the configuration file, named `[DocumentTracking]`.
3. In the new section, specify where the document tracking events are sent.

- To send document tracking events to a database through ODBC, set the following parameters:

`Backend`	To send document tracking events to a database, set this parameter to **`Library`**.
`LibraryPath`	Specify the location of the ODBC document tracking library. This is included with IDOL Server.
`ConnectionString`	The ODBC connection string for the database.

  For example:

  ```
  [DocumentTracking]
  Backend=Library
  LibraryPath=C:\Autonomy\IDOLServer\IDOL\modules\dt_odbc.dll
  ConnectionString=DSN=MyDatabase
  ```

- To send document tracking events to the connector's synchronize log, set the following parameters:

`Backend`	To send document tracking events to the connector's logs, set this parameter to **`Log`**.
`DatabaseName`	The name of the log stream to send the document tracking events to. Set this parameter to **`synchronize`**.

  For example:

  ```
  [DocumentTracking]
  Backend=Log
  DatabaseName=synchronize
  ```

- To send document tracking events to an IDOL Server, set the following parameters:

`Backend`	To send document tracking events to an IDOL Server, set this parameter to **`IDOL`**.
`TargetHost`	The host name or IP address of the IDOL Server.
`TargetPort`	The index port of the IDOL Server.

  For example:

  ```
  [DocumentTracking]
  Backend=IDOL
  TargetHost=idol
  TargetPort=9001
  ```

  For more information about the parameters you can use to configure document tracking, refer to the *HPE File System Connector Reference*.

4. Save and close the configuration file.

Chapter 11: Lua Functions and Methods Reference

This section describes the functions and methods that you can use in your Lua scripts.

General Functions

Function	Description
abs_path	Returns the supplied path as an absolute path.
base64_decode	Decodes a base64-encoded string.
base64_encode	Base64-encodes a string.
convert_date_time	Converts date and time formats using standard IDOL date formats.
convert_encoding	Converts the encoding of a string from one character encoding to another.
copy_file	Copies a file.
create_path	Creates the specified directory tree.

Function	Description
create_uuid	Creates a universally unique identifier.
delete_file	Deletes a file.
delete_path	Deletes a specified directory, but only if it is empty.
doc_tracking	Raises a document tracking event for a document.
encrypt	Encrypts a string.
encrypt_ security_field	Encrypts the ACL.
extract_date	Searches a string for a date and returns the date.
file_setdates	Modifies the properties of a file (for example created date, last modified date).
get_config	Loads a configuration file.
get_log	(Deprecated) Returns a LuaLog object that provides the capability to write log messages.
get_log	Returns a LuaLog object that provides the capability to write log messages.
get_log_service	Obtains a LuaLogService object, from which you can obtain a LuaLog object that you can use to write messages to a log file.
get_task_config	Returns a LuaConfig object that contains the configuration of the fetch task that called the script.
get_task_name	Returns the name of the fetch task that called the script.
getcwd	Returns the current working directory of the application.
gobble_ whitespace	Reduces multiple adjacent while spaces.
hash_file	Hashes a file using the SHA1 or MD5 algorithm.
hash_string	Hashes a string.
is_dir	Checks if the supplied path is a directory.
log	Appends log messages to a file.
move_file	Moves a file.
parse_csv	Parse comma-separated values into individual strings.
parse_ document_csv	Parse a CSV file into documents and call a function on each document.
parse_ document_idx	Parse an IDX file into documents and call a function on each document.

Function	Description
parse_document_idx	Parse an IDX string or file and return a LuaDocument.
parse_document_xml	Parse an XML file into documents and call a function on each document.
parse_document_xml	Parse an XML string or file and return a LuaDocument.
parse_json	Parse a string of JSON data and return a LuaJsonValue.
parse_json_array	Parse a string of JSON data (that is a JSON array) and a return a LuaJsonArray.
parse_json_object	Parse a string of JSON data (that is a JSON object) and return a LuaJsonObject.
parse_xml	Parse XML string to a LuaXmlDocument.
regex_match	Performs a regular expression match on a string.
regex_replace_all	Searches a string for matches to a regular expression, and replaces the matches.
regex_search	Performs a regular expression search on a string.
script_path	Returns the path and file name of the script that is running.
send_aci_action	Sends a query to an ACI server.
send_aci_command	Sends a query to an ACI server.
send_and_wait_for_async_aci_action	Sends a query to an ACI server and then waits for the action to finish. Use this method for sending asynchronous actions so that the action response is returned instead of a token.
send_http_request	Sends an HTTP request.
sleep	Pauses the running thread.
unzip_file	Extracts the contents of a zip file.
url_escape	Percent-encode a string.
url_unescape	Replaces URL escaped characters and returns a standard string.
xml_encode	Takes a string and encodes it using XML escaping.
zip_file	Zips the supplied path (file or directory).

abs_path

The `abs_path` method returns the supplied path as an absolute path.

Syntax

```
abs_path( path )
```

Arguments

Argument	Description
`path`	(string) A relative path.

Returns

(String). A string containing the supplied path as an absolute path.

base64_decode

The `base64_decode` method decodes a base64-encoded string.

Syntax

```
base64_decode( input )
```

Arguments

Argument	Description
`input`	(string) The string to decode.

Returns

(String). The decoded string.

If the input is not a valid base64-encoded string, the function returns `nil`.

base64_encode

The `base64_encode` method base64-encodes a string.

Syntax

```
base64_encode( input )
```

Arguments

Argument	Description
input	(string) The string to base64-encode.

Returns

(String). A base64-encoded string.

convert_date_time

The convert_date_time method converts date and time formats using standard IDOL formats. All date and time input is treated as local time unless it contains explicit time zone information.

The InputFormatCSV and OutputFormat arguments specify date and time formats, and accept the following values:

- AUTNDATE. The HPE date format (1 to a maximum of 10 digits). This format covers the epoch range (1 January 1970 to 19 January 2038) to a resolution of one second, and dates between 30 October 1093 BC and 26 October 3058 to a resolution of one minute.
- date formats that you specify using one or more of the following:

YY	Year (2 digits). For example, 99, 00, 01 and so on.
YYYY	Year (4 digits). For example, 1999, 2000, 2001 and so on.
#YY+	Year (2 or 4 digits). If you provide 2 digits, then it uses the YY format. If you provide 4 digits, it uses the YYYY format. For example, it interprets 07 as 2007 AD and 1007 as 1007 AD.
#Y	Year (1 to a maximum of 16 digits) and can be followed by AD or BC. An apostrophe (') immediately before the year denotes a truncated year. For example, 2008, '97 (interpreted as 1997), 97 (interpreted as 97 AD), '08 (interpreted as 2008), 2008 AD and 200 BC. A truncated year with a BC identifier is invalid ('08 BC).
#FULLYEAR	Year (1 to a maximum of 16 digits). For example 8, 98, 108, 2008, each of which is taken literally. The year is taken relative to the common EPOCH (0AD).
#ADBC	Time Period. For example, AD, CE, BC, BCE or any predefined list of EPOCH indicators. Typically, the year specified using the above Year formats is interpreted as un-truncated and relative to the EPOCH. For example, 84 AD is interpreted as 1984 AD and 84 BC is interpreted as 84 BC.

	The only exception to this is when you use both #YY+ and #ADBC. In this case, the format is interpreted as un-truncated even if the year was set to truncated by #YY+. For example, 99 AD is interpreted as the year 99 AD. HPE recommends you use only YY, YYYY or #FULLYEAR with #ADBC.
LONGMONTH	A long month, for example, January, February and so on.
SHORTMONTH	A short month, for example, Jan, Feb and so on.
MM	Month (2 digits). For example, 01, 10, 12 and so on.
M+	Month (1 or 2 digits). For example, 1,2,3,10 and so on.
DD	Day (2 digits). For example, 01, 02, 03, 12, 23 and so on.
D+	Day (1 or 2 digits). For example, 1, 2, 12, 13, 31 and so on.
LONGDAY	2 digits with a postfix. For example, 1st, 2nd and so on.
HH	Hour (2 digits). For example, 01, 12, 13 and so on.
H+	Hour (1 or 2 digits).
NN	Minute (2 digits).
N+	Minute (1 or 2 digits).
SS	Second (2 digits).
S+	Second (1 or 2 digits).
ZZZ	Time Zone, for example, GMT, EST, PST, and so on.
ZZZZZ	Time Difference (1 to 9 digits). For example, +04 denotes 4 hours ahead of UTC. Other examples include +4, +04, +0400, +0400 MSD (the string MSD is ignored). A further example is +030, in this case the time differences is interpreted as 30 minutes.
#PM	AM or PM indicator (2 characters). For example, 2001/09/09 02:46:40 pm
#S	A space

The following table shows some example date and time formats:

Date and time format string	Example date
DD/MM/YYYY	09/05/2013
D+ SHORTMONTH YYYY	2 Jan 2001
D+ LONGMONTH YYYY HH:NN:SS ZZZZZ	17 August 2003 10:41:07 -0400

Syntax

```
convert_date_time( Input, InputFormatCSV, OutputFormat )
```

Arguments

Argument	Description
`Input`	(string) The date and time to convert.
`InputFormatCSV`	(string) A comma-separated list of the possible date and time formats of the input.
`OutputFormat`	(string) The format of the date and time to output.

Returns

(String). A string containing the date and time in the desired format.

convert_encoding

The `convert_encoding` method converts the encoding of a string from one character encoding to another.

Syntax

```
convert_encoding( input, encodingTo, encodingTables [, encodingFrom])
```

Arguments

Argument	Description
`input`	(string) The string to convert.
`encodingTo`	(string) The character encoding to convert *to* (same as IDOL encoding names).
`encodingTables`	(string) The path to the conversion tables.
`encodingFrom`	(string) The character encoding to convert *from*. The default is “UTF8”.

Returns

(String). A string, using the specified character encoding.

copy_file

The `copy_file` method copies a file.

Syntax

```
copy_file( src, dest [, overwrite] )
```

Arguments

Argument	Description
`src`	(string) The source file.
`dest`	(string) The destination path and file name.
`overwrite`	(boolean) A boolean that specifies whether to copy the file if the destination file already exists. If this argument is `false` and the file already exists, the copy operation fails. The default is `true`, which means that the existing file is overwritten.

Returns

(Boolean). A Boolean, `true` to indicate success or `false` for failure.

create_path

The `create_path` method creates the specified directory tree.

Syntax

```
create_path( path )
```

Arguments

Argument	Description
`path`	(string) The path to create.

create_uuid

The `create_uuid` method creates a universally unique identifier.

Syntax

```
create_uuid()
```

Returns

(String). A string containing the universally unique identifier.

delete_file

The `delete_file` method deletes a file.

Syntax

```
delete_file( path )
```

Arguments

Argument	Description
`path`	(string) The path and filename of the file to delete.

Returns

(Boolean). A boolean, `true` to indicate success or `false` for failure.

delete_path

The `delete_path` function deletes the specified directory, but only if it is empty.

Syntax

```
delete_path( path )
```

Arguments

Argument	Description
`path`	(string) The empty directory to delete.

Returns

Nothing.

Example

```
delete_path( "C:\MyFolder\AnotherFolder\" )
```

doc_tracking

The `doc_tracking` function raises a document tracking event for a document.

Syntax

```
doc_tracking( document , eventName [, eventMetadata] [, reference] )
```

Arguments

Argument	Description
`document`	(LuaDocument) The document to track.
`eventName`	(string) The event name. You can type a description of the event.
`eventMetadata`	(table) A table of key-value pairs that contain metadata for the document tracking event.
`reference`	(string) The document reference. You can set this parameter to override the document reference used.

Returns

(Boolean). A Boolean that indicates whether the event was raised successfully.

Example

```
local ref=document:getReference()

doc_tracking(document, "The document has been processed",
      {myfield="myvalue", anotherfield="anothervalue"}, ref )
```

encrypt

The `encrypt` method encrypts a string and returns the encrypted string. It uses the same encryption method as ACL encryption.

Syntax

```
encrypt( content )
```

Arguments

Argument	Description
`content`	(string) The string to encrypt.

Returns

(String). The encrypted string.

encrypt_security_field

The `encrypt_security_field` method returns the encrypted form of the supplied field.

Syntax

```
encrypt_security_field( field )
```

Arguments

Argument	Description
`field`	(string) An Access Control List string.

Returns

(String). An encrypted string.

extract_date

The `extract_date` function searches a string for a date and returns the date. This function uses standard IDOL date formats. All date and time input is treated as local time unless it contains explicit

time zone information.

The following table describes the standard IDOL date formats:

YY	Year (2 digits). For example, 99, 00, 01 and so on.
YYYY	Year (4 digits). For example, 1999, 2000, 2001 and so on.
#YY+	Year (2 or 4 digits). If you provide 2 digits, then it uses the YY format. If you provide 4 digits, it uses the YYYY format. For example, it interprets 07 as 2007 AD and 1007 as 1007 AD.
#Y	Year (1 to a maximum of 16 digits) and can be followed by AD or BC. An apostrophe (') immediately before the year denotes a truncated year. For example, 2008, '97 (interpreted as 1997), 97 (interpreted as 97 AD), '08 (interpreted as 2008), 2008 AD and 200 BC. A truncated year with a BC identifier is invalid ('08 BC).
#FULLYEAR	Year (1 to a maximum of 16 digits). For example 8, 98, 108, 2008, each of which is taken literally. The year is taken relative to the common EPOCH (0AD).
#ADBC	Time Period. For example, AD, CE, BC, BCE or any predefined list of EPOCH indicators. Typically, the year specified using the above Year formats is interpreted as un-truncated and relative to the EPOCH. For example, 84 AD is interpreted as 1984 AD and 84 BC is interpreted as 84 BC. The only exception to this is when you use both #YY+ and #ADBC. In this case, the format is interpreted as un-truncated even if the year was set to truncated by #YY+. For example, 99 AD is interpreted as the year 99 AD. HPE recommends you use only YY, YYYY or #FULLYEAR with #ADBC.
LONGMONTH	A long month, for example, January, February and so on.
SHORTMONTH	A short month, for example, Jan, Feb and so on.
MM	Month (2 digits). For example, 01, 10, 12 and so on.
M+	Month (1 or 2 digits). For example, 1,2,3,10 and so on.
DD	Day (2 digits). For example, 01, 02, 03, 12, 23 and so on.
D+	Day (1 or 2 digits). For example, 1, 2, 12, 13, 31 and so on.
LONGDAY	2 digits with a postfix. For example, 1st, 2nd and so on.
HH	Hour (2 digits). For example, 01, 12, 13 and so on.
H+	Hour (1 or 2 digits).
NN	Minute (2 digits).
N+	Minute (1 or 2 digits).
SS	Second (2 digits).

S+	Second (1 or 2 digits).
ZZZ	Time Zone, for example, GMT, EST, PST, and so on.
ZZZZZ	Time Difference (1 to 9 digits). For example, +04 denotes 4 hours ahead of UTC. Other examples include +4, +04, +0400, +0400 MSD (the string MSD is ignored). A further example is +030, in this case the time differences is interpreted as 30 minutes.
#PM	AM or PM indicator (2 characters). For example, 2001/09/09 02:46:40 pm
#S	A space

The following table shows some example date and time formats:

Date and time format string	Example date
DD/MM/YYYY	09/05/2013
D+ SHORTMONTH YYYY	2 Jan 2001
D+ LONGMONTH YYYY HH:NN:SS ZZZZZ	17 August 2003 10:41:07 -0400

Syntax

```
extract_date( input, formatCSV, outputFormat )
```

Arguments

Argument	Description
input	(string) The string that you want to search for a date.
formatCSV	(string) A comma-separated list of the possible date and time formats for dates contained in the input.
outputFormat	(string) The format for the output.

Returns

(String). A string containing the date and time in the desired format.

Example

The following example would return the value "1989/01/14":

```
extract_date("This string contains a date 14/01/1989 somewhere",
"DD/YYYY/MM,DD/MM/YYYY", "YYYY/MM/DD")
```

file_setdates

The `file_setdates` method sets the metadata for the file specified by `path`. If the `format` argument is not specified, the dates must be specified in seconds since the epoch (1st January 1970).

Syntax

```
file_setdates( path, created, modified, accessed [, format] )
```

Arguments

Argument	Description
`path`	(string) The path or filename of the file.
`created`	(string) The date created (Windows only).
`modified`	(string) The date modified.
`accessed`	(string) The date last accessed.
`format`	(string) The format of the dates supplied. The format parameter uses the same values as other IDOL components. The default is "EPOCHSECONDS"

Returns

(Boolean). A Boolean indicating whether the operation was successful.

get_config

The `get_config` function loads a configuration file.

Configuration files are cached after the first call to `get_config`, to avoid unnecessary disk I/O in the likely event that the same configuration is accessed frequently by subsequent invocations of the Lua script. One cache is maintained per Lua state, so the maximum number of reads for a configuration file is equal to the number of threads that run Lua scripts.

If you do not specify a path, the function returns the configuration file with the same name as the ACI server executable file.

Syntax

```
get_config( [path] )
```

Arguments

Argument	Description
`path`	(string) The path of the configuration file to load.

Returns

(LuaConfig). A LuaConfig object.

get_log

> **DEPRECATED:**
> This version of the `get_log` function is deprecated in HPE File System Connector 11.4.0 and later.
>
> This version of the function is still available for existing implementations, but it might be incompatible with new functionality. This version of the function might be removed in future.
>
> HPE recommends that you use the function `get_log( log_type )` instead.

The `get_log` function reads a configuration file and returns a LuaLog object that provides the capability to use the specified log stream.

Syntax

```
get_log( config, logstream )
```

Arguments

Argument	Description
`config`	(LuaConfig) A LuaConfig object that represents the configuration file which contains the log stream. You can obtain a LuaConfig object using the function get_config.
`logstream`	(string) The name of the section in the configuration file that contains the settings for the log stream.

Returns

(LuaLog). A LuaLog object that provides the capability to use the log stream.

Example

```
local config = get_config("connector.cfg")
local log = get_log(config, "SynchronizeLogStream")
```

get_log

The `get_log` method returns a LuaLog object that provides the capability to write messages to a specified log type.

Syntax

```
get_log( log_type )
```

Arguments

Argument	Description
`log_type`	(string) The log type name.

Returns

(LuaLog). A LuaLog object.

Example

```
local log = get_log("application")
log:write_line(log_level_normal(), "doing something...")
```

get_log_service

The `get_log_service` function obtains a LuaLogService object, from which you can obtain a LuaLog object that you can use to write messages to a log file.

> **IMPORTANT:**
> To obtain a `LuaLog` object for writing to a standard log stream, call the function without any arguments or use the function get_log instead. Set the `config` argument only when you want to write to a custom log file that is not controlled by the ACI Server.

Syntax

```
get_log_service( [config] )
```

Arguments

Argument	Description
`config`	(LuaConfig) A LuaConfig object that represents the configuration file which contains the logging settings. You can obtain a LuaConfig object using the function get_config.

Returns

(LuaLogService). A LuaLogService object.

Example

The following example shows how to use the `get_log_service` function to obtain a `LuaLogService` object. From this you can obtain a `LuaLog` object. The `LuaLog` object has a method, `write_line`, that writes messages to the log file.

```
local logService = get_log_service()
-- import is a standard log stream for CFS
local log = logService:get_log("import")
log:write_line( log_level_error() , "The log message")
```

The get_log function provides an easier way to accomplish the same task:

```
local log = get_log("import")
-- import is a standard log stream for CFS
log:write_line( log_level_error() , "The log message")
```

The following example demonstrates how to write messages to a custom log file. Declare the log service globally to avoid a `LuaLogService` object being created every time the script runs. For example, if you are writing a Lua script to use with a connector or CFS, call the `get_log_service` function outside the `handler` function.

```
local luaConfigString = [===[
[Logging]
LogLevel=FULL
0=LuaLogStream

[LuaLogStream]
LogTypeCSVs=lua
LogFile=lua.log
]===]

local config = LuaConfig:new(luaConfigString)

-- global log service instance for Lua log stream
luaLogService = get_log_service(config)
```

```
local luaLog = luaLogService:get_log("lua")
luaLog:write_line(log_level_normal(), "running Lua script")
```

get_task_config

The `get_task_config` function returns a LuaConfig object that contains the configuration of the fetch task that called the script.

For information about the methods you can use to read information from the LuaConfig object, see LuaConfig Methods, on page 140.

Syntax

```
get_task_config()
```

Returns

(LuaConfig). A LuaConfig object.

get_task_name

The `get_task_name` function returns a string that contains the name of the fetch task that called the script.

Syntax

```
get_task_name()
```

Returns

(String). A string that contains the task name.

getcwd

The `getcwd` method returns the current working directory of the application.

Syntax

```
getcwd()
```

Returns

(String). Returns a string containing the absolute path of the current working directory.

gobble_whitespace

The `gobble_whitespace` method reduces multiple adjacent white spaces (tabs, carriage returns, spaces, and so on) in the specified string to a single space.

Syntax

```
gobble_whitespace( input )
```

Arguments

Argument	Description
`input`	(string) An input string.

Returns

(String). A string without adjacent white spaces.

hash_file

The `hash_file` method hashes the contents of the specified file using the SHA1 or MD5 algorithm.

Syntax

```
hash_file( FileName, Algorithm )
```

Arguments

Argument	Description
`FileName`	(string) The name of the file.
`Algorithm`	(string) The type of algorithm to use. Must be either SHA1 or MD5.

Returns

(String). A hash of the file contents.

hash_string

The `hash_string` method hashes the specified string using the SHA1 or MD5 algorithm.

Syntax

```
hash_string( StringToHash, Algorithm )
```

Arguments

Argument	Description
`StringToHash`	(string) The string to hash.
`Algorithm`	(string) The algorithm to use. Must be either SHA1 or MD5.

Returns

(String). The hashed input string.

is_dir

The `is_dir` method checks if the supplied path is a directory.

Syntax

```
is_dir( path )
```

Arguments

Argument	Description
`path`	(string) The path to check.

Returns

(Boolean). Returns `true` if the supplied path is a directory, `false` otherwise.

log

The `log` method appends log messages to the specified file.

Syntax

```
log( file, message )
```

Arguments

Argument	Description
`file`	(string) The file to append log messages to.
`message`	(string) The message to print to the file.

move_file

The `move_file` method moves a file.

Syntax

```
move_file( src, dest [, overwrite] )
```

Arguments

Argument	Description
`src`	(string) The source file.
`dest`	(string) The destination file.
`overwrite`	(boolean) A boolean that specifies whether to move the file if the destination file already exists. If this argument is `false`, and the destination file already exists, the move operation fails. The default is `true`, which means that the destination file is overwritten.

Returns

(Boolean). Returns `true` to indicate success, `false` otherwise.

parse_csv

The `parse_csv` method parses a string of comma-separated values into individual strings. The method understands quoted values (such that parsing 'foot, "leg, torso", elbow' produces three values) and ignores white space around delimiters.

Syntax

```
parse_csv( input [, delimiter ] )
```

Arguments

Argument	Description
`input`	(string) The string to parse.
`delimiter`	(string) The delimiter to use (the default delimiter is ",").

Returns

(Strings). You can put them in a table like this:

```
local results = { parse_csv("cat,tree,house", ",") };
```

parse_document_csv

The `parse_document_csv` function parses a CSV file into documents and calls a function on each document.

This function can handle CSV files with or without a header row, but if a header row is not present you must:

- set the named parameter `use_header_row` to `false`.
- specify the document field names to use by setting the named parameter `csv_field_names`.

Syntax

```
parse_document_csv( filename, handler [, params ] )
```

Arguments

Argument	Description
`filename`	(string) The path and file name of the CSV file to parse into documents.
`handler`	(document_handler_function) The function to call on each document that is parsed from the CSV file.
`params`	(table) A table of named parameters to configure parsing. The table maps parameter names (String) to parameter values. For information about the parameters that you can set, see the following table.

Named Parameters

Named Parameter	Description
`content_field`	(string, default `DRECONTENT`) The name of the field, in the CSV file, to use as the document content.
`csv_field_names`	(string list) A list of names for the fields that exist in the CSV file. This overrides any header row, if one is present.
`reference_field`	(string, default `DREREFERENCE`) The name of the field, in the CSV file, to use as the document reference.
`use_header_row`	(boolean, default `TRUE`) Specify whether the CSV file includes a header row (whether the first row is a list of field names and not values). If this parameter is `True` and you do not set `csv_field_names`, the field names in the header row are used as the names of the document fields.

Example

The following example parses a CSV file named `data.csv`, and calls the function `documentHandler` on each document. The values in the field `item_id` become document references and the values in the field `body` become document content.

```
function documentHandler(document)
  -- do something, for example
  print(document:getReference())
end

...

parse_document_csv("./data.csv", documentHandler, {
        reference_field="item_id",
        content_field="body"
    })
```

The following example shows how to provide field names when there is no header row in the CSV file:

```
parse_document_csv("./data_no_header.csv", documentHandler, {
        use_header_row=false,
        csv_field_names={"DREREFERENCE", "title", "modified", "DRECONTENT"}
    })
```

Returns

Nil.

parse_document_idx

The `parse_document_idx` function parses an IDX file into documents and calls a function on each document.

Syntax

```
parse_document_idx( filename, handler )
```

Arguments

Argument	Description
`filename`	(string) The path and file name of the IDX file to parse into documents.
`handler`	(document_handler_function) The function to call on each document that is parsed from the IDX file. The function must accept a LuaDocument as the only argument.

Example

The following example uses the `parse_document_idx` function to parse an IDX file, and calls the function `appendDocumentReference` on each document.

```
local references = {}

function appendDocumentReference(document)
    table.insert(references, document:getReference())
end

function read_idx_references(filename)
    parse_document_idx(filename, appendDocumentReference)
    return references
end
```

Returns

Nil.

parse_document_idx

The `parse_document_idx` function parses an IDX string or file, and returns a LuaDocument.

> **TIP:**
> You can use this function if the string or file contains a single document. If you have an IDX file

> that contains multiple documents you can use the function `parse_document_idx(filename, handler)` instead.

Syntax

```
parse_document_idx( input, file )
```

Arguments

Argument	Description
`input`	(string) The string or path to the file that contains the document in IDX format.
`file`	(boolean, default `false`) Specifies whether the `input` argument is a path to a file.

Example

If you have a string named `myIdxString` that contains a document in IDX format, you can obtain a LuaDocument object as follows:

```
local myDocument = parse_document_idx(myIdxString)
```

Returns

(LuaDocument). A LuaDocument object that represents the document.

parse_document_xml

The `parse_document_xml` function parses an XML file into documents and calls a function on each document.

Syntax

```
parse_document_xml( filename, handler [, params ] )
```

Arguments

Argument	Description
`filename`	(string) The path and file name of the XML file to parse into documents.
`handler`	(document_handler_function) The function to call on each document that is parsed from the XML file. The function must accept a LuaDocument as the only argument.
`params`	(table) A table of named parameters to configure parsing. The table maps parameter

Argument	Description
	names (String) to parameter values. For information about the parameters that you can set, see the following table.

Named Parameters

Named Parameter	Description
`content_paths`	(string list, default `DRECONTENT`) The paths in the XML to the elements that contain document content. You can specify a list of paths.
`document_root_paths`	(string list, default `DOCUMENT`) The paths in the XML to the elements that represent the root of a document. You can specify a list of paths.
`include_root_path`	(boolean, default `false`) Specifies whether to include the `document_root_paths` node in the document metadata. The default value includes only children of the root node.
`reference_paths`	(string list, default `DREREFERENCE`) The paths in the XML to elements that contain document references. Though you can specify a list of paths, there must be exactly one reference per document.

The default values of the `content_paths`, `document_root_path`, and `reference_paths` arguments are suitable for processing XML documents written out using the CFS `XmlWriter`.

Example

The following example parses an XML file named `data.xml`, and calls the function `printReference` on each document. Two values have been set for the named parameter `content_paths`. You might want to do this if there are multiple fields that contain content or you want to use the same script with XML files that have different schema.

```
local function printReference(document)
    print(document:getReference())
end

local xmlParams = {
        document_root_paths={"DOC"},
        reference_paths={"REF"},
        content_paths={"CONTENT","MORE_CONTENT"}
    }

parse_document_xml("./data.xml", printReference, xmlParams)
```

Returns

Nothing.

parse_document_xml

The `parse_document_xml` function parses an XML string or file into a document and returns a LuaDocument.

> **TIP:**
> You can use this function if the string or file contains a single document. If you have an XML file that contains multiple documents you can use the function `parse_document_xml(filename, handler [, params] )` instead.

Syntax

```
parse_document_xml( input, file [, params ] )
```

Arguments

Argument	Description
`input`	(string) The string or path to the file that contains the document in XML format.
`file`	(boolean) Specifies whether the `input` argument is a path to a file.
`params`	(table) A table of named parameters to configure parsing. The table maps parameter names (String) to parameter values. For information about the parameters that you can set, see the following table.

Named Parameters

Named Parameter	Description
`content_paths`	(string list, default `DRECONTENT`) The paths in the XML to the elements that contain document content. You can specify a list of paths.
`document_root_paths`	(string list, default `DOCUMENT`) The paths in the XML to the elements that represent the root of a document. You can specify a list of paths.
`include_root_path`	(boolean, default `false`) Specifies whether to include the `document_root_paths` node in the document metadata. The default value includes only children of the root node.
`reference_paths`	(string list, default `DREREFERENCE`) The paths in the XML to elements that contain document references. Though you can specify a list of paths, there must be exactly one reference per document.

The default values of the `content_paths`, `document_root_path`, and `reference_paths` arguments are suitable for processing XML documents written out using the CFS `XmlWriter`.

Example

If you have a string named `myXmlString` that contains a document in XML format, you can obtain a LuaDocument object as follows:

```
local xmlParams = {
        document_root_paths={"DOC"},
        reference_paths={"REF"},
        content_paths={"CONTENT","MORE_CONTENT"}
    }

local myDocument = parse_document_xml(myXmlString, false, xmlParams)
```

Returns

(LuaDocument). A LuaDocument object that represents the document.

parse_json

The `parse_json` function parses a string of JSON and returns a JSON value.

Syntax

```
parse_json( json )
```

Arguments

Argument	Description
`json`	(string) The input string to parse.

Returns

(LuaJsonValue). A LuaJsonValue containing the JSON data.

Example

```
local fh = io.open("example_json.json", "r")
local file_content = fh:read("*all")
fh:close()

local myJsonValue = parse_json(file_content)

if myJsonValue:is_object() then
```

```
  document:insertJson( myJsonValue:object() , "MyJsonField" )
elseif myJsonValue:is_array() then
  document:insertJson( myJsonValue:array() , "MyJsonField" )
end
```

If the file `example_json.json` contained the following JSON object:

```
{
  "product": "IDOL",
  "component": "CFS",
  "version": { "major": 11, "minor": 3 },
  "ports": [
    { "type": "ACI", "number": 7000 },
    { "type": "Service", "number": 17000 }
  ]
}
```

The resulting document might look like this:

```
<document>
   <reference>ref1</reference>
   <xmlmetadata>
      <MyJsonField>
        <component>CFS</component>
        <ports>
          <number>7000</number>
          <type>ACI</type>
        </ports>
        <ports>
          <number>17000</number>
          <type>Service</type>
        </ports>
        <product>IDOL</product>
        <version>
          <major>11</major>
          <minor>3</minor>
        </version>
      </MyJsonField>
   </xmlmetadata>
</document>
```

See Also

If you know that the input data is a JSON object, you can also use the function parse_json_object. If you know that the input data is a JSON array, you can also use the function parse_json_array.

parse_json_array

The `parse_json_array` function parses a string and returns a JSON array.

Syntax

```
parse_json_array( json )
```

Arguments

Argument	Description
json	(string) The input string to parse.

Returns

(LuaJsonArray). A LuaJsonArray containing the JSON data.

See Also

If you can't be sure whether the input data is a JSON array, you can use the function parse_json to parse the JSON. The parse_json function returns a LuaJsonValue, which can represent a JSON array or a JSON object.

parse_json_object

The `parse_json_object` function parses a string and returns a JSON object.

Syntax

```
parse_json_object( json )
```

Arguments

Argument	Description
json	(string) The input string to parse.

Returns

(LuaJsonObject). A LuaJsonObject containing the JSON data.

See Also

If you can't be sure whether the input data is a JSON object, you can use the function parse_json to parse the JSON. The parse_json function returns a LuaJsonValue, which can represent a JSON array

or a JSON object.

parse_xml

The `parse_xml` method parses an XML string to a LuaXmlDocument.

Syntax

```
parse_xml( xml )
```

Arguments

Argument	Description
`xml`	(string) XML data as a string.

Returns

(LuaXmlDocument). A LuaXmlDocument containing the parsed data, or nil if the string could not be parsed.

regex_match

The `regex_match` method performs a regular expression match on a string.

Syntax

```
regex_match( input, regex [, case] )
```

Arguments

Argument	Description
`input`	(string) The string to match.
`regex`	(string) The regular expression to match against.
`case`	(boolean) A boolean that specifies whether the match is case-sensitive. The match is case sensitive by default (`true`).

Returns

One or more strings, or `nil`.

If the string matches the regular expression, and the regular expression has no sub-matches, the full string is returned.

If the string matches the regular expression, and the regular expression has sub-matches, then only the sub-matches are returned.

If the string does not match the regular expression, there are no return values (any results are `nil`).

You can assign multiple strings to a table. To assign the return values to a table, surround the function call with braces. For example:

```
matches = { regex_match( input, regex ) }
```

Examples

```
local r1, r2, r3 = regex_match( "abracadabra", "(a.r)((?:a.)*ra)" )
```

Results: `r1="abr"`, `r2="acadabra"`, `r3=nil`

```
local r1, r2, r3 = regex_match( "abracadabra", "a.r(?:a.)*ra" )
```

Results: `r1="abracadabra"`, `r2=nil`, `r3=nil`

regex_replace_all

The `regex_replace_all` method searches a string for matches to a regular expression, and replaces the matches according to the value specified by the `replacement` argument.

Syntax

```
regex_replace_all( input, regex, replacement )
```

Arguments

Argument	Description
`input`	(string) The string in which you want to replace values.
`regex`	(string) The regular expression to use to find values to be replaced.
`replacement`	(string) A string that specifies how to replace the matches of the regular expression.

Returns

(String). The modified string.

Examples

```
regex_replace_all("ABC ABC ABC", "AB", "A")
-- returns "AC AC AC"

regex_replace_all("One Two Three", "\\w{3}", "_")
-- returns "_ _ _ee"


regex_replace_all("One Two Three", "(\\w+) (\\w+)", "\\2 \\1")
-- returns "Two One Three"
```

regex_search

The `regex_search` method performs a regular expression search on a string. This method returns a LuaRegexMatch object, rather than strings.

Syntax

```
regex_search ( input, regex [, case])
```

Arguments

Argument	Description
`input`	(string) The string in which to search.
`regex`	(string) The regular expression with which to search.
`case`	(boolean) A boolean that specifies whether the match is case-sensitive. The match is case sensitive by default (`true`).

Returns

(LuaRegexMatch).

script_path

The `script_path` function returns the path and file name of the script that is running.

Syntax

```
script_path()
```

Returns

(String, String) Returns the path of the folder that contains the script and the file name of the script, as separate strings.

Example

```
local script_directory, script_filename = script_path()
```

You can use this function to load scripts using their location relative to the current script. In the following example only the first return value from `script_path()` - the directory - is concatenated with `"more_scripts/another_script.lua"`.

```
dofile(script_path().."more_scripts/another_script.lua")
```

send_aci_action

The `send_aci_action` method sends a query to an ACI server. This method takes the action parameters as a table instead of the full action as a string, as with `send_aci_command`. This avoids issues with parameter values containing an ampersand (&).

Syntax

```
send_aci_action( host, port, action [, parameters] [, timeout] [, retries] [,
sslParameters] )
```

Arguments

Argument	Description
`host`	(string) The ACI host to send the query to.
`port`	(number) The port to send the query to.
`action`	(string) The action to perform (for example, `query`).
`parameters`	(table) A Lua table containing the action parameters, for example, { param1="value1", param2="value2" }
`timeout`	(number) The number of milliseconds to wait before timing out. The default is 3000.
`retries`	(number) The number of times to retry if the request fails. The default is 3.
`sslParameters`	(table) A Lua table containing the SSL settings.

Returns

(String). Returns the XML response as a string. If required, you can call `parse_xml` on the string to return a `LuaXmlDocument`. If the request fails, it returns nil.

Example

```
send_aci_action( "localhost", 9000, "query" ,
                      {text = "*", print = "all"} );
```

See Also

- send_aci_command, below

send_aci_command

The `send_aci_command` method sends a query to an ACI server.

Syntax

```
send_aci_command( host, port, query [, timeout] [, retries] [, sslParameters] )
```

Arguments

Argument	Description
`host`	(string) The ACI host to send the query to.
`port`	(number) The port to send the query to.
`query`	(string) The query to send (for example, `action=getstatus`)
`timeout`	(number) The number of milliseconds to wait before timing out. The default is 3000.
`retries`	(number) The number of times to retry if the request fails. The default is 3.
`sslParameters`	(table) A Lua table containing the SSL settings.

Returns

(String). Returns the XML response as a string. If required, you can call `parse_xml` on the string to return a `LuaXmlDocument`. If the request fails, it returns nil.

See Also

- send_aci_action, on page 133

send_and_wait_for_async_aci_action

The `send_and_wait_for_async_aci_action` method sends a query to an ACI server. The method does not return until the action has completed.

You might use this method when you want to use an asynchronous action. The `send_aci_action` method returns as soon as it receives a response, which for an asynchronous action means that it returns a token. The method `send_and_wait_for_async_aci_action` sends an action and then waits. It polls the server until the action is complete and then returns the response.

Argument	Description
`host`	(string) The ACI host to send the query to.
`port`	(number) The ACI port to send the query to.
`action`	(string) The name of the action to perform.
`parameters`	(table) A Lua table containing the action parameters, for example, { param1="value1", param2="value2" }
`timeout`	(number) The number of milliseconds to wait before timing out. The default is 60000 (1 minute).
`retries`	(number) The number of times to retry if the connection fails. The default is 3.
`sslParameters`	(table) A Lua table containing the SSL settings.

Returns

(String). Returns the XML response as a string. If required, you can call `parse_xml` on the string to return a `LuaXmlDocument`. If the request fails, it returns nil.

See Also

- send_aci_action, on page 133

send_http_request

The `send_http_request` function sends a HTTP request.

Syntax

```
send_http_request( params )
```

Arguments

Argument	Description
`params`	(table) Named parameters that configure the HTTP request. The table maps parameter names (string) to parameter values. For information about the parameters that you can set, see the following table.

Named Parameters

Named Parameter	Description
`url`	(string) The full URL for the HTTP request. If you provide a URL, the other parameters used to build the URL are not used.
`method`	(string) The HTTP method to use (`GET`, `POST`, or `DELETE`). The default is `"GET"`.
`headers`	(table) A table of HTTP headers to send with the request. The table must map header names to values.
`content`	(string) For HTTP POST, the data to be sent with the post.
`site`	(string) The site to which the HTTP request is sent.
`port`	(integer) The port to which the HTTP request is sent. By default, the request is sent to port 80.
`uri`	(string) The URI to request.
`params`	(table) Additional parameters for the request. The table must map parameter names to parameter values.
`section`	(string) The name of a section in the configuration file that contains transport related parameters such as SSL or proxy settings.

Returns

String. The HTTP response.

Examples

```
local wikipedia_page = send_http_request(
   {url = "http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol"} )

local google_search = send_http_request(
   {site = "www.google.co.uk", port = 80, uri = "/search",
   params = {q = "http protocol", safe = "active"}} )

local search = send_http_request(
   {site = "www.site.com", port = 80, uri = "/query",
   params = {text = "http headers", maxresults = "10"}
   headers = { Cache-Control = "no-cache" }} )
```

sleep

The `sleep` method pauses the thread.

Syntax

```
sleep( milliseconds )
```

Arguments

Argument	Description
`milliseconds`	(number) The number of milliseconds for which to pause the current thread.

unzip_file

The `unzip_file` method extracts the contents of a zip file.

Syntax

```
unzip_file( path, dest )
```

Arguments

Argument	Description
`path`	(string) The path and filename of the file to unzip.
`dest`	(string) The destination path where files are extracted.

Returns

(Boolean). Returns a Boolean indicating success or failure.

url_escape

The `url_escape` method percent-encodes a string.

Syntax

```
url_escape( input )
```

Arguments

Argument	Description
`input`	(string) The string to percent-encode.

Returns

(String). The percent-encoded string.

url_unescape

The `url_unescape` method replaces URL escaped characters and returns a standard string.

Syntax

```
url_unescape( input )
```

Arguments

Argument	Description
`input`	(string) The string to process.

Returns

(String). The modified string.

xml_encode

The `xml_encode` method takes a string and encodes it using XML escaping.

Syntax

```
xml_encode ( content )
```

Arguments

Argument	Description
`content`	(string) The string to encode.

Returns

(String).

zip_file

The `zip_file` method zips the supplied path (file or directory). It overwrites the output file only if you set the optional `overwrite` argument to `true`.

Syntax

```
zip_file( path [, overwrite] )
```

Arguments

Argument	Description
`path`	(string) The path or filename of the file or folder to zip.
`overwrite`	(boolean) A boolean that specifies whether to force the creation of the zip file if an output file already exists. The default is `false`.

Returns

(Boolean). Returns a Boolean indicating success or failure. On success writes a file called `path.zip`.

LuaConfig Methods

A `LuaConfig` object provides access to configuration information. You can retrieve a `LuaConfig` for a given configuration file using the `get_config` function.

If you have a `LuaConfig` object called `config` you can call its methods using the ':' operator. For example:

```
config:getValue(sectionName, parameterName)
```

Constructor	Description
LuaConfig:new	The constructor for a `LuaConfig` object (creates a new `LuaConfig` object).

Method	Description
getEncryptedValue	Returns the unencrypted value from the config of an encrypted value.
getValue	Returns the value of the configuration parameter key in a given section.
getValues	Returns all the values of a configuration parameter if you have multiple values for a key (for example, a comma-separated list or numbered list like keyN).

getEncryptedValue

The `getEncryptedValue` method returns the unencrypted value from the configuration file of an encrypted value.

Syntax

```
getEncryptedValue( section, parameter )
```

Arguments

Argument	Description
`section`	(string) The section in the configuration file.
`parameter`	(string) The parameter in the configuration file to get the value for.

Returns

(String). The unencrypted value.

getValue

The `getValue` method returns the value of the configuration parameter key in a given section. If the key does not exist in the section, then it returns the default value.

Syntax

```
getValue( section, key [, default] )
```

Arguments

Argument	Description
`section`	(string) The section name in the configuration file.
`key`	(string) The name of the key from which to read.
`default`	(string/boolean/number) The default value to use if no key is found.

Returns

A string, boolean, or integer containing the value read from the configuration file.

getValues

The `getValues` method returns multiple values for a parameter (for example, a comma-separated list or numbered list like keyN).

Syntax

```
getValues( section, parameter )
```

Arguments

Argument	Description
`section`	(string) The section in the configuration file.
`parameter`	(string) The parameter to find in the configuration file.

Returns

(Strings). The strings can be assigned to a table. To map the return values to a table, surround the function call with braces. For example:

```
values = { config:getValues( section, parameter ) }
```

LuaConfig:new

The constructor for a `LuaConfig` object (creates a new `LuaConfig` object).

Syntax

```
LuaConfig:new( config_buffer )
```

Arguments

Argument	Description
`config_buffer`	(string) The configuration to use to create the `LuaConfig` object.

Returns

(LuaConfig). The new `LuaConfig` object.

Example

```
local config_buffer = "[default]\nparameter=value"
local config = LuaConfig:new(config_buffer)
```

LuaDocument Methods

This section describes the methods provided by the `LuaDocument` object. A `LuaDocument` allows you to access and modify the reference, metadata and content of a document.

If you have a `LuaDocument` object called `document` you can call its methods using the `':'` operator. For example:

```
document:addField(name, value)
```

Constructor	Description
LuaDocument:new	The constructor for a `LuaDocument` object (creates a new `LuaDocument` object that only contains a reference).

Method	Description
addField	Creates a new field.
addSection	Add an empty section to the end of the document.
appendContent	Appends content to the existing content of the document.
copyField	Creates a new named field with the same value as an existing named field.
copyFieldNoOverwrite	Copies a field to a certain name but does not overwrite the existing value.
countField	Returns the number of fields with the name specified.
deleteField	Removes a field from the document.
getContent	Returns the document content.
getField	Returns the first field with a specified name.
getFieldNames	Returns all the field names for the document.
getFields	Returns all fields with the specified name.
getFieldValue	Gets a field value.
getFieldValues	Gets all values of a multi-valued field.
getNextSection	Gets the next section in a document, allowing you to perform find or add operations on every section.
getReference	Returns the document reference.
getSection	Returns a LuaDocument object with the specified section as the active section.
getSectionCount	Returns the number of sections in the document.
getValueByPath	Gets the value of the document field or sub field with the specified path.
getValuesByPath	Gets all values of a multi-value document field or sub field, with the specified path.
hasField	Checks whether the document has a named field.
insertJson	Inserts metadata, from a JSON string, into a document.
insertXml	Inserts XML metadata into a document.
insertXmlWithoutRoot	Inserts XML metadata into a document.
removeSection	Removes a section from a document.
renameField	Renames a field.
setContent	Sets the content for a document.

Method	Description
setFieldValue	Sets a field value.
setReference	Sets the document reference.
to_idx	Returns a string containing the document in IDX format.
to_json	Returns a string containing the document in JSON format.
to_xml	Returns a string containing the document in XML format.
writeStubIdx	Writes out a stub IDX document.
writeStubXml	Writes out a stub XML document.

addField

The `addField` method adds a new field to the document.

Syntax

```
addField ( fieldname, fieldvalue )
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field to add.
`fieldvalue`	(string) The value to set for the field.

addSection

The `addSection` method adds an empty section to the end of the document.

Syntax

```
addSection()
```

Returns

(LuaDocument). Returns a LuaDocument object representing the document, with the new section as the active section.

Example

```
local newSection = document:addSection()   -- Add a new section to the document
newSection:setContent("content")           -- Set content for the new section
```

appendContent

The `appendContent` method appends `content` to the existing content (the `DRECONTENT` field) of a document or document section.

Syntax

```
appendContent ( content [, number])
```

Arguments

Argument	Description
`content`	(string) The content to append.
`number`	(number) The document section to modify. If you do not specify this argument, `content` is appended to the last section. If you specify a number greater than the number of existing sections, additional empty sections are created.

Examples

```
-- Append content to the last section
document:appendContent("content")

-- Append content to section 7, empty sections are created before this section if
necessary
document:appendContent("content", 7)
```

copyField

The `copyField` method copies a field value to a new field. If the target field already exists it is overwritten.

Syntax

```
copyField ( sourcename, targetname [, case] )
```

Arguments

Argument	Description
`sourcename`	(string) The name of the field to copy.
`targetname`	(string) The destination field name.
`case`	(boolean) A boolean that specifies whether `sourcename` is case-sensitive. The field name is case sensitive by default (`true`).

copyFieldNoOverwrite

The `copyFieldNoOverwrite` method copies a field value to a new field but does not overwrite the existing value. After calling this function the target field will contain all values of the source field in addition to any values it already had.

Syntax

```
copyFieldNoOverwrite( sourcename, targetname [, case])
```

Arguments

Argument	Description
`sourcename`	(string) The name of the field to copy.
`targetname`	(string) The destination field name.
`case`	(boolean) A boolean that specifies whether `sourcename` is case-sensitive. The name is case sensitive by default (`true`).

countField

The `countField` method returns the number of fields with the specified name.

Syntax

```
countField( fieldname [, case] )
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field to count.
`case`	(boolean) A boolean that specifies whether `fieldname` is case sensitive. The field name is case sensitive by default (`true`).

Returns

(Number) The number of fields with the specified name.

deleteField

The `deleteField` method deletes a field from a document. If you specify the optional `value` argument, the field is deleted only if has the specified value.

Syntax

```
deleteField( fieldName [, case] )
```

```
deleteField( fieldName, value [, case] )
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field to delete.
`value`	(string) The value of the field. If this is specified only fields with matching names and values are deleted. If this is not specified, all fields that match `fieldname` are deleted.
`case`	(boolean) A boolean that specifies whether `fieldname` is case sensitive. The field name is case sensitive by default (`true`).

getContent

The `getContent` method gets the content (the value of the `DRECONTENT` field) for a document or document section.

Syntax

```
getContent([number])
```

Arguments

Argument	Description
`number`	(number) The document section for which you want to return the content. If you do not specify this argument, the method returns the content of the active section. For the document object passed to the script's `handler` function, the active section is the first section (section 0).

Returns

(String). The document content as a string.

Examples

```
local content7 = document:getContent(7)     -- Get content for section 7
local section = document:getSection(3)      -- Get document for section 3
local content3 = section:getContent()       -- Get content for section 3
local content0 = document:getContent()      -- Get content for section 0
```

getField

The `getField` method returns a LuaField object representing the field with the specified name.

Syntax

```
getField( name [, case])
```

Arguments

Argument	Description
`name`	(string) The name of the field.
`case`	(boolean) A boolean that specifies whether the `name` argument is case-sensitive. The name is case sensitive by default (`true`).

Returns

(LuaField). A LuaField object.

getFieldNames

The `getFieldNames` method returns all of the field names for the document.

Syntax

```
getFieldNames()
```

Returns

(Strings) The names of the fields. To map the return values to a table, surround the function call with braces. For example:

```
names = { document:getFieldNames() }
```

getFields

The `getFields` method returns LuaField objects where each object represents a field that matches the specified name.

Syntax

```
getFields( name [, case])
```

Arguments

Argument	Description
`name`	(string) The name of the field.
`case`	(boolean) A boolean that specifies whether the `name` argument is case-sensitive. The name is case sensitive by default (`true`).

Returns

(LuaFields) One LuaField for each matching field. To map the return values to a table, surround the function call with braces. For example:

```
fields = { document:getFields( name ) }
```

getFieldValue

The `getFieldValue` method gets the value of a field in a document. To return the values of a multi-value field, see getFieldValues, below.

Syntax

```
getFieldValue( fieldname [, case])
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field to be retrieved.
`case`	(boolean) A boolean that specifies whether `fieldname` is case-sensitive. The argument is case sensitive by default (`true`).

Returns

(String). A string containing the value.

getFieldValues

The `getFieldValues` method gets all values from all fields that have the same name.

Syntax

```
getFieldValues( fieldname [, case])
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field.
`case`	(boolean) A boolean that specifies whether `fieldname` is case-sensitive. The argument is case sensitive by default (`true`).

Returns

(Strings). Strings that contain the values. To map the return values to a table, surround the function call with braces. For example:

```
fieldvalues = { document:getFieldValues( fieldname ) }
```

getNextSection

The `getNextSection` method returns the next section of a document (if the document has been divided into sections).

The document object passed to the script's `handler` function represents the first section of the document. This means that the methods described in this section read and modify only the first section.

Calling `getNextSection` on the LuaDocument passed to the `handler` function will always return the second section. To perform operations on every section, see the example below.

When a document is divided into sections, each section has the same fields. The only difference between each section is the document content (the value of the `DRECONTENT` field).

Syntax

```
getNextSection()
```

Returns

(LuaDocument) A LuaDocument object that contains the next DRE section.

Example

To perform operations on every section, use the following script.

```
local section = document
while section do
       -- Manipulate section
       section = section:getNextSection()
end
```

getReference

The `getReference` method returns a string containing the reference (the value of the `DREREFERENCE` document field).

Syntax

```
getReference()
```

Returns

(String). A string containing the reference.

getSection

The `getSection` method returns a LuaDocument object with the specified section as the active section.

Syntax

```
getSection(number)
```

Arguments

Argument	Description
`number`	(number) The document section for which you want to return a LuaDocument object.

Returns

(LuaDocument). A LuaDocument object with the specified section as the active section.

Example

```
-- Get object for section 7 of document
local section = document:getSection(7)

-- Get the content from the section
local content = section:getContent()
```

getSectionCount

The `getSectionCount` method returns the number of sections in a document.

Syntax

```
getSectionCount()
```

Returns

(Number). The number of sections.

Example

```
local sectionCount = document:getSectionCount()
```

getValueByPath

The `getValueByPath` method gets the value of a document field. The field is specified by its path, which means that you can get the value of a sub field. If you pass this method the path of a multi-value field, only the first value is returned. To return all of the values from a multi-value field, see getValuesByPath, below.

Syntax

```
getValueByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path of the field.

Returns

(String). A string containing the value.

Example

```
local value = document:getValueByPath("myfield")
local subfieldvalue = document:getValueByPath("myfield/subfield")
```

getValuesByPath

The `getValuesByPath` method gets all values of a document field. The field is specified by its path, which means that you can get values from a sub field.

Syntax

```
getValuesByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path of the field.

Returns

(Strings). Strings that contain the values. To map the return values to a table, surround the function call with braces. For example:

```
fieldvalues = { document:getValuesByPath("myfield/subfield") }
```

hasField

The `hasField` method checks to see if a field exists in the current document.

Syntax

```
hasField ( fieldname [, case])
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field.
`case`	(boolean) A boolean that specifies whether the `fieldname` is case-sensitive. The field name is case-sensitive by default.

Returns

(Boolean). `True` if the field exists, `false` otherwise.

insertJson

The `insertJson` method inserts metadata from a JSON string, LuaJsonObject, or LuaJsonArray, into a document.

Syntax

```
insertJson ( json [, fieldName] )
```

Arguments

Argument	Description
`json`	(string or LuaJsonArray or LuaJsonObject) The JSON to insert into the document.
`fieldName`	(string) The name of the metadata field to add the JSON to. This field is created if it does not exist. If you do not specify a field the JSON is added at the root of the document.

Example

The following example Lua script uses the `insertJson` method to add metadata to a document:

```
function handler(document)
  local jsonString = [[
    {"email":"example@domain.com",
     "name":"A N Example",
     "address":
       {"address1":"New street",
        "city":"Cambridge",
        "country":"Great Britain"},
     "id":15}
  ]]

  document:insertJson( jsonString , "MyField")
  return true
end
```

When written to an IDX file, the resulting document looks like this:

```
#DREREFERENCE c:\test.html
#DREFIELD DREFILENAME="c:\test.html"
...
#DREFIELD MyField/address/address1="New street"
#DREFIELD MyField/address/city="Cambridge"
#DREFIELD MyField/address/country="Great Britain"
#DREFIELD MyField/email="example@domain.com"
#DREFIELD MyField/id="15"
#DREFIELD MyField/name="A N Example"
#DRECONTENT
Some content

#DREENDDOC
```

insertXml

The `insertXml` method inserts XML metadata into the document.

Syntax

```
insertXml ( node )
```

Arguments

Argument	Description
node	(LuaXmlNode) The node to insert.

Returns

(LuaField). A LuaField object of the inserted data.

insertXmlWithoutRoot

The `insertXmlWithoutRoot` method inserts XML metadata into the document.

This method does not insert the top level node. All of the child nodes are inserted into the document. `insertXmlWithoutRoot(node)` is therefore equivalent to calling `insertXml()` for each child node.

Syntax

```
insertXmlWithoutRoot ( node )
```

Arguments

Argument	Description
node	(LuaXmlNode) The node to insert.

LuaDocument:new

The constructor for a `LuaDocument` object (creates a new `LuaDocument` object that only contains a reference).

Syntax

```
LuaDocument:new( reference )
```

Arguments

Argument	Description
`reference`	(string) The reference to assign to the new document.

Returns

(LuaDocument). The new `LuaDocument` object.

Example

```
local reference = "my_reference"
local document = LuaDocument:new(reference)
```

removeSection

The `removeSection` method removes a section from a document.

Syntax

```
removeSection( sectionNumber )
```

Arguments

Argument	Description
`sectionNumber`	(number) A zero-based index that specifies the section to remove. For example, to remove the second section, set this argument to `1`.

Returns

Nothing.

Example

```
-- Example that removes the last section of a document
if document:getSectionCount() > 0 then
   local lastSection = document:getSectionCount() - 1
   document:removeSection( lastSection )
end
```

renameField

The `renameField` method changes the name of a field.

Syntax

```
renameField( currentname, newname [, case])
```

Arguments

Argument	Description
`currentname`	(string) The name of the field to rename.
`newname`	(string) The new name of the field.
`case`	(boolean) A boolean that specifies whether the `currentname` argument is case-sensitive. The argument is case sensitive by default (`true`).

setContent

The `setContent` method sets the content (the value of the `DRECONTENT` field) for a document or document section.

Syntax

```
setContent( content [, number] )
```

Arguments

Argument	Description
`content`	(string) The content to set for the document or document section.
`number`	(number) The document section to modify. If you do not specify a number, the method modifies the active section. For the document object passed to the script's `handler` function, the active section is the first section (section 0). If you specify a number greater than the number of existing sections, additional empty sections are created.

Examples

```
-- Set content for section 0
document:setContent("content0")
```

```
-- Get document for section 1
local section = document:getNextSection()

-- Set content for section 1
section:setContent("content1")

-- Set content for section 7, and assign sections 2-6 to
-- empty string if non-existent
document:setContent("content7", 7)
```

setFieldValue

The `setFieldValue` method sets the value of a field in a document. If the field does not exist, it is created. If the field already exists, the existing value is overwritten.

Syntax

```
setFieldValue( fieldname, newvalue )
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field to set.
`newvalue`	(string) The value to set for the field.

setReference

The `setReference` method sets the reference (the value of the `DREREFERENCE` document field) to the string passed in.

Syntax

```
setReference( reference )
```

Arguments

Argument	Description
`reference`	(string) The reference to set.

to_idx

The `to_idx` method returns a string containing the document in IDX format.

Syntax

```
to_idx()
```

Returns

(String). Returns the document as a string.

to_json

The `to_json` method returns a string containing the document in JSON format.

Syntax

```
to_json()
```

Returns

(String). Returns the document as a string.

to_xml

The `to_xml` method returns a string containing the document in XML format.

Syntax

```
to_xml()
```

Returns

(String). Returns the document as a string.

writeStubIdx

The `writeStubIdx` method writes out a stub IDX document (a metadata file used by IDOL applications). The file is created in the current folder, but you can specify a full path and file name if you want to create the file in another folder.

Syntax

```
writeStubIdx( filename )
```

Arguments

Argument	Description
`filename`	(string) The name of the file to create.

Returns

(Boolean). `True` if written, `false` otherwise.

writeStubXml

The `writeStubXml` method writes out an XML file containing the metadata for the document. The file is created in the current folder but you can specify a full path and file name if you want to create the file in another folder.

Syntax

```
writeStubXml( filename )
```

Arguments

Argument	Description
`filename`	(String) The name of the file to create.

Returns

(Boolean) `True` if successful, `false` otherwise.

LuaField Methods

This section describes the methods provided by `LuaField` objects. A `LuaField` represents a single field in a document. You can retrieve `LuaField` objects for a document using the `LuaDocument` `getField` and `getFields` methods. In its simplest form a field has just a name and a value, but it can also contain sub-fields.

If you have a `LuaField` object called `field` you can call its methods using the ':' operator. For example:

```
field:addField(name, value)
```

Method	Description
addField	Adds a sub field with the specified name and value.
copyField	Copies the sub field to another sub field.
copyFieldNoOverwrite	Copies the sub field to another sub field but does not overwrite the destination.
countField	Returns the number of sub fields that exist with the specified name.
deleteAttribute	Deletes the attribute with the specified name.
deleteField	Deletes the sub field with the specified name.
getAttributeValue	Gets the value of an attribute.
getField	Gets the sub field specified by the name.
getFieldNames	Returns the names of all sub fields of this field.
getFields	Gets all the sub fields specified by the name.
getFieldValues	Returns all the values of the sub field with the specified name.
getValueByPath	Returns the value of a sub field with the specified path.
getValuesByPath	Returns all the values of the sub field with the specified path.
hasAttribute	Returns a Boolean specifying if the field has the specified attribute passed in by name.
hasField	Returns a Boolean specifying if the sub field exists or not.
insertJson	Inserts metadata from a JSON string, LuaJsonObject, or LuaJsonArray into the field.
insertXml	Inserts XML metadata into a document.
insertXmlWithoutRoot	Inserts XML metadata into a document.
name	Returns the name of the field object in a string.
renameField	Renames a sub field.
setAttributeValue	Sets the value for the specified attribute of the field.
setValue	Sets the value of the field.
value	Returns the value of the field object.

addField

The `addField` method adds a sub field with the specified name and value.

Syntax

```
addField( fieldname, fieldvalue )
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field.
`fieldvalue`	(string) The value of the field.

copyField

The `copyField` method copies a sub field to another sub field. If the target sub field exists it is overwritten.

Syntax

```
copyField ( from, to [, case])
```

Arguments

Argument	Description
`from`	(string) The name of the field to copy.
`to`	(string) The name of the field to copy to.
`case`	(boolean) A boolean that specifies whether the `from` argument is case sensitive. The argument is case sensitive by default (`true`).

copyFieldNoOverwrite

The `copyFieldNoOverwrite` method copies the sub field to another sub field but does not overwrite the destination. After this operation the destination field contains all the values of the source field as well as any values it already had.

Syntax

```
copyFieldNoOverwrite( from, to [, case])
```

Arguments

Argument	Description
`from`	(string) The name of the field to copy.
`to`	(string) The name of the field to copy to.
`case`	(boolean) A boolean that specifies whether the `from` argument is case sensitive. The argument is case sensitive by default (`true`).

countField

The `countField` method returns the number of sub fields that exist with the specified name.

Syntax

```
countField ( fieldname [, case])
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field.
`case`	(boolean) A boolean that specifies whether the `fieldname` argument is case sensitive. The argument is case sensitive by default (`true`).

Returns

(Number). The number of sub fields that exist with the specified name.

deleteAttribute

The `deleteAttribute` method deletes the specified field attribute.

Syntax

```
deleteAttribute( name )
```

Arguments

Argument	Description
`name`	(string) The name of the attribute to delete.

deleteField

The `deleteField` method deletes the sub field with the specified name.

Syntax

```
deleteField( name [, case] )
deleteField( name , value [, case] )
```

Arguments

Argument	Description
`name`	(string) The name of the sub field to delete.
`value`	(string) The value of the sub field. If this is specified a field is deleted only if it has the specified name and value. If this is not specified, all fields with the specified name are deleted.
`case`	(boolean) A boolean that specifies whether the `name` argument is case sensitive. The argument is case sensitive by default (`true`).

getAttributeValue

The `getAttributeValue` method gets the value of the specified attribute.

Syntax

```
getAttributeValue( name )
```

Arguments

Argument	Description
`name`	(string) The name of the attribute.

Returns

(String). The attribute value.

getField

The `getField` method returns the specified sub field.

Syntax

```
getField ( name [, case])
```

Arguments

Argument	Description
`name`	(string) The name of the field to return.
`case`	(boolean) A boolean that specifies whether the `name` argument is case sensitive. The argument is case sensitive by default (`true`).

Returns

(LuaField) A LuaField object.

getFieldNames

The `getFieldNames` method returns the names of the sub fields in the LuaField object.

Syntax

```
getFieldNames()
```

Returns

(Strings). The names of the sub fields. The strings can be assigned to a table. To map the return values to a table, surround the function call with braces. For example:

```
fieldnames = { field:getFieldNames() }
```

getFields

The `getFields` method returns all of the sub fields specified by the `name` argument.

Syntax

```
getFields( name [, case] )
```

Arguments

Argument	Description
`name`	(string) The name of the fields.
`case`	(boolean) A boolean that specifies whether the `name` argument is case sensitive. The argument is case sensitive by default (`true`).

Returns

(LuaFields) One LuaField per matching field. The objects can be assigned to a table. To map the return values to a table, surround the function call with braces. For example:

```
fields = { field:getFields( name [, case]) }
```

getFieldValues

The `getFieldValues` method returns the values of all of the sub fields with the specified name.

Syntax

```
getFieldValues( fieldname [, case] )
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field.
`case`	(boolean) A boolean that specifies whether the `fieldname` argument is case sensitive. The argument is case sensitive by default (`true`).

Returns

(Strings) One string for each value. The strings can be assigned to a table. To map the return values to a table, surround the function call with braces. For example:

```
fieldvalues = { field:getFieldValues( fieldname ) }
```

getValueByPath

The `getValueByPath` method gets the value of a sub-field, specified by path. If you pass this method the path of a sub-field that has multiple values, only the first value is returned. To return all of the values from a multi-value sub-field, see getValuesByPath, below.

Syntax

```
getValueByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path of the sub-field.

Returns

(String). A string containing the value.

Example

Consider the following document:

```
<DOCUMENT>
  ...
  <A_FIELD>
    <subfield>
      <anothersubfield>the value to return</anothersubfield>
    </subfield>
  </A_FIELD>
  ...
</DOCUMENT>
```

The following example demonstrates how to retrieve the value "the value to return" from the sub-field `anothersubfield`, using a LuaField object representing `A_FIELD`:

```
local field = document:getField("A_FIELD")
local value = field:getValueByPath("subfield/anothersubfield")
```

getValuesByPath

The `getValuesByPath` method gets the values of a sub-field, specified by path.

Syntax

```
getValuesByPath( path )
```

Arguments

Argument	Description
path	(string) The path of the sub-field.

Returns

(Strings). Strings that contain the values. To map the return values to a table, surround the function call with braces. For example:

```
fieldvalues = { myfield:getValuesByPath("subfield/anothersubfield") }
```

Example

Consider the following document:

```
<DOCUMENT>
  ...
  <A_FIELD>
    <subfield>
      <anothersubfield>one</anothersubfield>
      <anothersubfield>two</anothersubfield>
      <anothersubfield>three</anothersubfield>
    </subfield>
  </A_FIELD>
  ...
</DOCUMENT>
```

The following example demonstrates how to retrieve the values "one", "two", and "three" from the sub-field `anothersubfield`, using a LuaField object representing `A_FIELD`:

```
local field = document:getField("A_FIELD")
local values = { field:getValuesByPath("subfield/anothersubfield") }
```

hasAttribute

The `hasAttribute` method returns a Boolean indicating whether the field has the specified attribute.

Syntax

```
hasAttribute( name )
```

Arguments

Argument	Description
`name`	(string) The name of the attribute.

Returns

(Boolean). A Boolean specifying if the field has the specified attribute.

hasField

The `hasField` method returns a Boolean specifying if the sub field exists.

Syntax

```
hasField( fieldname [, case])
```

Arguments

Argument	Description
`fieldname`	(string) The name of the field.
`case`	(boolean) A boolean that specifies whether the `fieldname` argument is case sensitive. The argument is case sensitive by default (`true`).

Returns

(Boolean). A Boolean specifying if the sub field exists or not.

insertJson

The `insertJson` method inserts metadata from a JSON string, LuaJsonObject, or LuaJsonArray, into the field.

Syntax

```
insertJson ( json [, fieldName] )
```

Arguments

Argument	Description
`json`	(string or LuaJsonArray or LuaJsonObject) The JSON to insert into the field.
`fieldName`	(string) The name of a sub-field to add the JSON to. If you set this argument, a sub-field is created. If you do not set this argument, the JSON is added to the field that the method was called on.

See Also

- LuaDocument:insertJson

insertXml

The `insertXml` method inserts XML metadata into the document. When called on a LuaField, the `insertXml` method inserts the fields as children of the LuaField.

Syntax

```
insertXml ( node )
```

Arguments

Argument	Description
`node`	(LuaXmlNode) The node to insert.

Returns

(LuaField). A LuaField object of the inserted data.

insertXmlWithoutRoot

The `insertXmlWithoutRoot` method inserts XML metadata into the document.

This method does not insert the top level node. All of the child nodes are inserted into the document. `insertXmlWithoutRoot(node)` is therefore equivalent to calling `insertXml()` for each child node.

Syntax

```
insertXmlWithoutRoot ( node )
```

Arguments

Argument	Description
node	(LuaXmlNode) The node to insert.

name

The name method returns the name of the field object.

Syntax

```
name()
```

Returns

(String). The name of the field object.

renameField

The renameField method renames a sub field.

Syntax

```
renameField( oldname, newname [, case] )
```

Arguments

Argument	Description
oldname	(string) The previous name of the field.
newname	(string) The new name of the field.
case	(boolean) A boolean that specifies whether the oldname argument is case sensitive. The argument is case sensitive by default (true).

setAttributeValue

The setAttributeValue method sets the value for the specified attribute of the field.

Syntax

```
setAttributeValue( attribute, value )
```

Arguments

Argument	Description
`attribute`	(string) The name of the attribute to set.
`value`	(string) The value to set.

setValue

The `setValue` method sets the value of the field.

Syntax

```
setValue( value )
```

Arguments

Argument	Description
`value`	(string) The value to set.

value

The `value` method returns the value of the field object.

Syntax

```
value()
```

Returns

(String). The value of the field object.

LuaHttpRequest Methods

This section describes the methods provided by `LuaHttpRequest` objects.

If you have a `LuaHttpRequest` object called `request` you can call its methods using the ':' operator. For example:

```
request:send()
```

Constructor	Description
LuaHttpRequest:new	The constructor for a `LuaHttpRequest` object (creates a new `LuaHttpRequest`).

Method	Description
send	Sends the HTTP request.
set_body	Sets the body of the HTTP request.
set_config	Specifies settings to use for sending the HTTP request.
set_header	Adds a header to the HTTP request.
set_headers	Replaces all headers in the HTTP request with new headers.
set_method	Specifies the HTTP request method to use when sending the request, for example `GET` or `POST`.
set_url	Sets the URL that you want to send the HTTP request to.

LuaHttpRequest:new

The constructor for a `LuaHttpRequest` object (creates a new `LuaHttpRequest`).

Syntax

```
LuaHttpRequest:new( config, section )
```

Arguments

Argument	Description
`config`	(LuaConfig) The LuaConfig object that contains the settings for sending the HTTP request. To obtain a LuaConfig object, see the function get_config, on page 113 or LuaConfig:new, on page 142.
`section`	(string) The name of the configuration section from which to read the settings.

Returns

(LuaHttpRequest). The new `LuaHttpRequest` object.

Example

The following example creates a new `LuaHttpRequest` object by reading configuration settings from a string that is defined in the script:

```
local config_string = [===[
[HTTP]
ProxyHost=proxy.domain.com
ProxyPort=8080
SSLMethod=NEGOTIATE
]===]

local config = LuaConfig:new(config_string)
local request = LuaHttpRequest:new(config, "HTTP")
```

send

The `send` method sends the HTTP request.

Syntax

```
send()
```

Returns

(LuaHttpResponse). A LuaHttpResponse object.

set_body

The `set_body` method sets the body of the HTTP request.

Syntax

```
set_body( body )
```

Arguments

Argument	Description
body	(string or nil) The body of the HTTP request.

set_config

The `set_config` method specifies settings to use for sending the HTTP request (in the form of a section from a configuration file). For example, you can specify the proxy server to use by setting the parameters `ProxyHost` and `ProxyPort`, or specify the SSL version to use by setting `SSLMethod`.

Syntax

```
set_config( config, section )
```

Arguments

Argument	Description
`config`	(LuaConfig) The LuaConfig object that contains the settings for sending the HTTP request. To obtain a LuaConfig object, see the function get_config, on page 113 or LuaConfig:new, on page 142.
`section`	(string) The name of the configuration section from which to read the settings.

Example

The following example creates a new `LuaConfig` object from a string. Usually you would read the configuration settings from the configuration file using the function get_config, on page 113. It then uses the `LuaConfig` object to provide settings for a `LuaHttpRequest` named `request`.

```
local config_string = [===[
[HTTP]
ProxyHost=proxy.domain.com
ProxyPort=8080
SSLMethod=NEGOTIATE
]===]

local config = LuaConfig:new(config_string)
request:set_config(config, "HTTP")
```

set_header

The `set_header` method adds a header to the HTTP request.

Syntax

```
set_header( name, value )
```

Arguments

Argument	Description
`name`	(string) The name of the HTTP header to add to the request.
`value`	(string or nil) The value of the HTTP header to add to the request.

Example

```
request:set_header("X-EXAMPLE", "Example")
```

set_headers

The `set_headers` method replaces all headers in the HTTP request with new headers.

To add a header without changing existing headers, use the method set_header.

Syntax

```
set_headers( headers )
```

Arguments

Argument	Description
`headers`	(table) The new HTTP headers to add to the request. The table must map header names to values.

Example

```
request:set_headers( {X-EXAMPLE="Example", X-ANOTHER="ANOTHER"} )
```

set_method

Specifies the HTTP request method to use when sending the request, for example `GET` or `POST`.

Syntax

```
set_method( method )
```

Arguments

Argument	Description
`method`	(string) The HTTP request method to use.

Example

```
request:set_method("POST")
```

set_url

The `set_url` method sets the URL that you want to send the HTTP request to.

Syntax

```
set_url( url )
```

Arguments

Argument	Description
`url`	(string) The URL to send the request to.

Example

```
request:set_url("https://www.example.com/")
```

LuaHttpResponse Methods

This section describes the methods provided by `LuaHttpResponse` objects. A `LuaHttpResponse` object is provided when you send an HTTP request using the send method on a LuaHttpRequest object.

If you have a `LuaHttpResponse` object called `response` you can call its methods using the ':' operator. For example:

```
response:get_http_code()
```

Method	Description
get_body	Returns the body of the HTTP response.

Method	Description
get_header	Returns the value of the specified HTTP header.
get_headers	Returns the value of all HTTP headers in the response.
get_http_code	Returns the HTTP response code.

get_body

The `get_body` method returns the body of the HTTP response.

Syntax

```
get_body()
```

Returns

(String). The body of the HTTP response.

get_header

The `get_header` method returns the value of a specified HTTP header.

Syntax

```
get_header( name )
```

Arguments

Argument	Description
`name`	(string) The name of the HTTP header for which you want to get the value.

Returns

(String). The value of the header.

get_headers

The `get_headers` method returns the names and values of all of the headers in the HTTP response.

Syntax

```
get_headers()
```

Returns

(table) A table that maps header names to values.

Example

The following example sends an HTTP request and then prints the headers that are returned to the console:

```
local response = request:send()
for k,v in pairs(response:get_headers()) do
   print("HEADER:",k,v)
end
```

get_http_code

The `get_http_code` method returns the HTTP response code for the request.

Syntax

```
get_http_code()
```

Returns

(number). The response code, for example `200` for success, or `404` for not found.

LuaJsonArray Methods

A `LuaJsonArray` object represents a JSON array.

If you have a `LuaJsonArray` object called `myJsonArray` you can call its methods using the `':'` operator. For example:

```
myJsonArray:size()
```

Constructor	Description
LuaJsonArray:new	The constructor for a `LuaJsonArray` object (creates a new `LuaJsonArray`).

Method	Description
append	Appends a value to the array.
clear	Clears the array, so that it contains no values.
copy	Copies the array.
empty	Checks whether the array is empty.
exists	Checks whether a specified path exists in the array.
existsByPath	Checks whether a specified path exists in the array.
ipairs	Returns an iterator function that you can use to iterate over all of the values in the array.
lookup	Returns the value at a specified path in the array.
lookupByPath	Returns the value at a specified path in the array.
size	Returns the size of the array.
string	Returns the array as a string.

LuaJsonArray:new

The constructor for a `LuaJsonArray` object (creates a new `LuaJsonArray`).

Syntax

```
LuaJsonArray:new( values )
```

Arguments

Argument	Description
`values`	(lua_json_type) The values to add to the array. Each value can be a Boolean, float, integer, string, LuaJsonArray, LuaJsonObject, LuaJsonValue, or nil.

Returns

(LuaJsonArray). The new `LuaJsonArray` object.

Example

The following is a simple JSON array:

```
[4, "a string", true]
```

To create a `LuaJsonArray` object to represent this array:

```
local myJsonArray = LuaJsonArray:new(4,"a string",true)
```

append

The `append` method appends a value to the JSON array.

Syntax

```
append( value )
```

Arguments

Argument	Description
`value`	(lua_json_type) The value to append to the array. This can be a Boolean, float, integer, string, LuaJsonArray, LuaJsonObject, LuaJsonValue, or nil.

Example

```
local myJsonArray = LuaJsonArray:new(4,"a string",true)
myJsonArray:append(6)
-- the array is now [4, "a string", true, 6]
```

clear

The `clear` method clears the array, so that it contains no values.

Syntax

```
clear()
```

Example

```
local myJsonArray = LuaJsonArray:new(4,"a string",true)
myJsonArray:clear()
print (myJsonArray:string())
-- []
```

copy

The `copy` method returns a copy of the array. Modifying the copy does not modify the original array.

Syntax

```
copy()
```

Returns

(LuaJsonArray). A copy of the array.

Example

```
local myJsonArray = LuaJsonArray:new(4,"a string",true)
local anotherJsonArray = myJsonArray:copy()
print (anotherJsonArray:string())
-- [4, "a string", true]
```

empty

The `empty` method checks whether the JSON array is empty.

Syntax

```
empty()
```

Returns

(Boolean). Returns `true` if the array is empty.

Example

```
local myJsonArray = LuaJsonArray:new(4,"a string",true)

print ("Is the array empty?  " , myJsonArray:empty() )
-- Is the array empty?   false

myJsonArray:clear()

print ("Is the array empty?  " , myJsonArray:empty() )
-- Is the array empty?   true
```

exists

The `exists` method checks whether a specified path exists in the JSON array.

Syntax

```
exists( pathElements )
```

Arguments

Argument	Description
`pathElements`	(json_path_string_or_integer) The path to check. Specify one or more path elements, which might be object attribute names (strings) or array indexes (integers).

Returns

(Boolean) Returns `true` if the path exists, `false` otherwise.

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
local myJsonArray = LuaJsonArray:new("zero", 1, 2, myJsonObject)

print (myJsonArray:exists(0))
-- true (LuaJsonArrays are zero-indexed)

print (myJsonArray:exists(4))
-- false (LuaJsonArrays are zero-indexed, and there is no fifth value)

print (myJsonArray:exists(3, "product"))
-- true
```

See Also

- existsByPath, below

existsByPath

The `existsByPath` method checks whether a specified path exists in the JSON array.

Syntax

```
existsByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path to check. Construct the path from array indexes and object attribute names, and use a slash (`/`) as the separator. If an object attribute name includes a slash, then escape the slash with a backslash, for example `"one\\/two"`. Notice that you must also escape a backslash with another backslash.

Returns

(Boolean) Returns `true` if the path exists, `false` otherwise.

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
local myJsonArray = LuaJsonArray:new("zero",1,2,myJsonObject)

print (myJsonArray:existsByPath("3/product"))
-- true (LuaJsonArrays are zero-indexed)
```

See Also

- exists, on page 183

ipairs

The `ipairs` method returns a function that you can use with a for loop to iterate over all of the values in the array.

Syntax

```
ipairs()
```

Example

```
local myJsonArray = LuaJsonArray:new(4,"a string",true)

for i, v in myJsonArray:ipairs() do
  print (v:string())
end

-- prints the following...
```

```
-- 4
-- a string
-- true
```

lookup

The `lookup` method returns the value at the specified path in the JSON array.

> **NOTE:**
> This method does not make a copy of the value, so modifying the returned value affects the original array.

Syntax

```
lookup( pathElements )
```

Arguments

Argument	Description
`pathElements`	(json_path_string_or_integer) The path of the value to return. Specify one or more path elements, which might be object attribute names (strings) or array indexes (integers).

Returns

(LuaJsonValue) Returns the value that exists at the specified path, or `nil` if the specified path does not exist.

Example

The following example demonstrates how to obtain a value:

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
local myJsonArray = LuaJsonArray:new(0, 1, 2, myJsonObject)
local myValue = myJsonArray:lookup(3, "product")
print (myValue:value())
-- IDOL
```

The following example demonstrates how modifying a returned value affects the original array:

```
local myJsonObject = LuaJsonObject:new()
myJsonObject:assign("name", LuaJsonArray:new("value1"))

local myJsonArray = myJsonObject:lookup("name"):array()
myJsonArray:append("value2")
```

```
print (myJsonObject:string())
-- {"name":["value1","value2"]}
```

See Also

- lookupByPath, below

lookupByPath

The `lookupByPath` method returns the value at the specified path in the JSON array.

> **NOTE:**
> This method does not make a copy of the value, so modifying the returned value affects the original array.

Syntax

```
lookupByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path of the value to return. Construct the path from array indexes and object attribute names, and use a slash (`/`) as the separator. If an object attribute name includes a slash, then escape the slash with a backslash, for example `"one\\/two"`. Notice that you must also escape a backslash with another backslash.

Returns

(LuaJsonValue) Returns the value that exists at the specified path, or `nil` if the specified path does not exist.

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
local myJsonArray = LuaJsonArray:new("zero",1,2,myJsonObject)

local myValue = myJsonArray:lookupByPath("3/product")
print (myValue:value())
-- IDOL
```

See Also

- lookup, on page 186

size

The `size` method returns the size of the JSON array.

Syntax

```
size()
```

Returns

(integer). The number of values in the array.

Example

```
local myJsonArray = LuaJsonArray:new(4,"a string",true)
print ("Array size is " , myJsonArray:size() )
-- Array size is 3
```

string

The `string` method returns the JSON array as a string.

Syntax

```
string()
```

Returns

(String). The array as a string.

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
local myJsonArray = LuaJsonArray:new("zero",1,2,myJsonObject)

print (myJsonArray:string())
-- ["zero",1,2,{"product":"IDOL","version":11}]
```

LuaJsonObject Methods

A `LuaJsonObject` object represents a JSON object.

If you have a `LuaJsonObject` object called `object` you can call its methods using the ':' operator. For example:

```
object:size()
```

Constructor	Description
LuaJsonObject:new	The constructor for a `LuaJsonObject` object (creates a new `LuaJsonObject`).

Method	Description
assign	Adds an attribute (a name/value pair) to the object.
assign	Adds a table of attributes (name/value pairs) to the object.
clear	Clears the object, so that it contains no attributes.
copy	Copies the object.
empty	Checks whether the object is empty.
erase	Removes a specified attribute (name/value pair) from the object.
exists	Checks whether a specified path exists in the object.
existsByPath	Checks whether a specified path exists in the object.
lookup	Returns the value at a specified path in the object.
lookupByPath	Returns the value at a specified path in the object.
pairs	Returns an iterator function that you can use to iterate over all of the attributes in the object.
size	Returns the number of attributes (name/value pairs) contained in the object.
string	Returns the object as a string.

LuaJsonObject:new

The constructor for a `LuaJsonObject` object (creates a new `LuaJsonObject`).

If you do not specify any attributes to assign to the object (by omitting the `attributes` argument), an empty object is created.

Syntax

```
LuaJsonObject:new( [attributes] )
```

Arguments

Argument	Description
`attributes`	(lua_json_object_type) A table of attributes (name/value pairs) to assign to the object. The keys in the table must be strings which specify the names of the attributes. Each value can be a Boolean, float, integer, string, LuaJsonObject, LuaJsonArray, LuaJsonValue, or nil. **TIP:** In Lua, keys in a table cannot be assigned the value of `nil`. If you want to have a null value (`nil` in Lua), you can do the following: `LuaJsonObject:new( { key=LuaJsonValue:new(nil) } ).`

Returns

(LuaJsonObject). The new `LuaJsonObject` object.

Example

The following is a simple JSON object:

```
{"product":"IDOL","version":11}
```

To create a `LuaJsonObject` object to represent this JSON object:

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
```

assign

The `assign` method adds an attribute (a name/value pair) to the object.

Syntax

```
assign (name, value)
```

Arguments

Argument	Description
`name`	(string) The name of the attribute to add.
`value`	(lua_json_type) The value of the attribute to add. This can be a Boolean, float, integer, string, LuaJsonArray, LuaJsonObject, LuaJsonValue, or nil.

Example

```
local myJsonObject = LuaJsonObject:new( { name1="value1" , name2="value2" } )
myJsonObject:assign("name3", "value3")
myJsonObject:assign("name4", LuaJsonArray:new("value4a","value4b") )
```

assign

The `assign` method adds a table of attributes (name/value pairs) to the object.

Syntax

```
assign (attributes)
```

Arguments

Argument	Description
`attributes`	(lua_json_object_type) The table of attributes to add. The keys in the table must be strings which specify the names of the attributes to add. Each value can be a Boolean, float, integer, string, LuaJsonArray, LuaJsonObject, LuaJsonValue, or nil. **TIP:** In Lua, keys in a table cannot be assigned the value of `nil`. If you want to have a null value (`nil` in Lua), you can do the following: `myJsonObject:assign( { key=LuaJsonValue:new(nil) } ).`

Example

```
local myJsonObject = LuaJsonObject:new()
myJsonObject:assign( {
        attr1=LuaJsonObject:new( { n=42 } ),
        attr2=LuaJsonArray:new( 1, 2, "three" ),
        attr3=true
```

```
    } )
print(myJsonObject:string())
-- {"attr1":{"n":42},"attr2":[1,2,"three"],"attr3":true}
```

clear

The `clear` method clears the object, so that it contains no attributes (name/value pairs).

Syntax

```
clear()
```

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
myJsonObject:clear()
print (myJsonObject:string())
-- {}
```

copy

The `copy` method returns a copy of the JSON object. Modifying the copy does not modify the original object.

Syntax

```
copy()
```

Returns

(LuaJsonObject). A copy of the object.

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
local anotherJsonObject = myJsonObject:copy()

print (anotherJsonObject:string())
-- {"product":"IDOL","version":11}
```

empty

The `empty` method determines whether the JSON object is empty.

Syntax

```
empty()
```

Returns

(Boolean). Returns `true` if the object has no attributes (name/value pairs).

Example

```
local myJsonObject = LuaJsonObject:new()
print(myJsonObject:empty())
-- true
```

erase

The `erase` method removes a specified attribute (name/value pair) from the JSON object.

Syntax

```
erase( name )
```

Arguments

Argument	Description
`name`	(string) The name of the attribute to erase.

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
myJsonObject:erase("version")
print (myJsonObject:string())
-- {"product":"IDOL"}
```

exists

The `exists` method checks whether a specified path exists in the JSON object.

Syntax

```
exists( pathElements )
```

Arguments

Argument	Description
`pathElements`	(json_path_string_or_integer) The path to check. Specify one or more path elements, which might be object attribute names (strings) or array indexes (integers).

Returns

(Boolean) Returns `true` if the path exists, `false` otherwise.

Example

Consider the following JSON:

```
{
  "product": "IDOL",
  "component": "CFS",
  "version": { "major": 11, "minor": 3 },
  "ports": [
    { "type": "ACI", "number": 7000 },
    { "type": "Service", "number": 17000 }
  ]
}
```

If this JSON is represented by an object named `myJsonObject`, you could use the `exists` method as follows:

```
print (myJsonObject:exists("ports",0,"type"))
-- true (LuaJsonArrays are zero-indexed)

print (myJsonObject:exists("ports",2,"type"))
-- false (LuaJsonArrays are zero-indexed, and there is no third value)

print (myJsonObject:exists("version", "major"))
-- true
```

See Also

- existsByPath, below

existsByPath

The `existsByPath` method checks whether a specified path exists in the JSON object.

Syntax

```
existsByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path to check. Construct the path from array indexes and object attribute names, and use a slash (`/`) as the separator. If an object attribute name includes a slash, then escape the slash with a backslash, for example `"one\\/two"`. Notice that you must also escape a backslash with another backslash.

Returns

(Boolean) Returns `true` if the path exists, `false` otherwise.

Example

Consider the following JSON:

```
{
  "product": "IDOL",
  "component": "CFS",
  "version": { "major": 11, "minor": 3 },
  "ports": [
    { "type": "ACI", "number": 7000 },
    { "type": "Service", "number": 17000 }
  ]
}
```

If this JSON is represented by an object named `myJsonObject`, you could use the `existsByPath` method as follows:

```
print (myJsonObject:existsByPath("ports/0/type"))
-- true (LuaJsonArrays are zero-indexed)

print (myJsonObject:existsByPath("ports/2/type") )
-- false (LuaJsonArrays are zero-indexed, and there is no third value)

print (myJsonObject:existsByPath("version/major"))
-- true
```

See Also

- exists, on page 193

lookup

The `lookup` method returns the value at the specified path in the JSON object.

> **NOTE:**
> This method does not make a copy of the value, so modifying the returned value affects the original object.

Syntax

```
lookup( pathElements )
```

Arguments

Argument	Description
`pathElements`	(json_path_string_or_integer) The path of the value to return. Specify one or more path elements, which might be object attribute names (strings) or array indexes (integers).

Returns

(LuaJsonValue) Returns the value that exists at the specified path, or `nil` if the specified path does not exist.

Example

```
myJsonObject = LuaJsonObject:new()
myJsonObject:assign( {
        attr1=LuaJsonObject:new( { n=42, x=5 } ),
        attr2=LuaJsonArray:new( 1, 2, "three" ),
    } )

print ( myJsonObject:lookup("attr2",2):value() )
-- three (LuaJsonArrays are zero-indexed)

-- Modifying the returned value changes the original object
local lookup_attr = myJsonObject:lookup("attr1"):object()
lookup_attr:assign("y",3)
print (myJsonObject:string())
-- {"attr1":{"n":42,"x":5,"y":3},"attr2":[1,2,"three"]}
```

See Also

- lookupByPath, below

lookupByPath

The `lookupByPath` method returns the value at the specified path in the JSON object.

> **NOTE:**
> This method does not make a copy of the value, so modifying the returned value affects the original object.

Syntax

```
lookupByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path of the value to return. Construct the path from array indexes and object attribute names, and use a slash (`/`) as the separator. If an object attribute name includes a slash, then escape the slash with a backslash, for example `"one\\/two"`. Notice that you must also escape a backslash with another backslash.

Returns

(LuaJsonValue) Returns the value that exists at the specified path, or `nil` if the specified path does not exist.

Example

```
myJsonObject = LuaJsonObject:new()
myJsonObject:assign( {
        attr1=LuaJsonObject:new( { n=42, x=5 } ),
        attr2=LuaJsonArray:new( 1, 2, "three" ),
    } )

print ( myJsonObject:lookupByPath("attr2/2"):value() )
-- three (LuaJsonArrays are zero-indexed)

-- Modifying the returned value changes the original object
local lookup_attr = myJsonObject:lookupByPath("attr1"):object()
lookup_attr:assign("y",3)
```

```
print (myJsonObject:string())
-- {"attr1":{"n":42,"x":5,"y":3},"attr2":[1,2,"three"]}
```

See Also

- lookup, on page 196

pairs

The `pairs` method returns an iterator function that you can use with a for loop to iterate over all of the attributes contained in the JSON object.

Syntax

```
pairs()
```

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )

for i, v in myJsonObject:pairs() do
  print (i .. '=' .. v:string())
end

-- prints the following...
-- product=IDOL
-- version=11
```

size

The `size` method returns the number of attributes (name/value pairs) in the JSON object.

Syntax

```
size()
```

Returns

(Number). The number of attributes in the JSON object.

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
print (myJsonObject:size())
-- 2
```

string

The `string` method returns the JSON object as a string.

Syntax

```
string()
```

Returns

(String). The JSON object as a string.

Example

```
local myJsonObject = LuaJsonObject:new( { product="IDOL" , version=11 } )
print (myJsonObject:string())
-- {"product":"IDOL","version":11}
```

LuaJsonValue Methods

A `LuaJsonValue` object represents a JSON value.

If you have a `LuaJsonValue` object called `myJsonValue` you can call its methods using the `':'` operator. For example:

```
myJsonValue:is_object()
```

Constructor	Description
LuaJsonValue:new	The constructor for a `LuaJsonValue` object (creates a new `LuaJsonValue`).

Method	Description
array	Gets a `LuaJsonArray` that represents the JSON array, if the value is an array.
copy	Copies the value.

Method	Description
exists	Checks whether a specified path exists in the value (when the value is a JSON array or object).
existsByPath	Checks whether a specified path exists in the value (when the value is a JSON array or object).
is_array	Returns whether the value is a JSON array.
is_boolean	Returns whether the value is a Boolean value.
is_float	Returns whether the value is a floating point value.
is_integer	Returns whether the value is an integer value.
is_null	Returns whether the value is a null value.
is_number	Returns whether the value is a number.
is_object	Returns whether the value is a JSON object.
is_simple_value	Returns whether the value is a simple value (Boolean, float, integer, or string).
is_string	Returns whether the value is a string value.
lookup	Returns the value at a specified path (when the value is a JSON array or object).
lookupByPath	Returns the value at a specified path (when the value is a JSON array or object).
object	Gets a `LuaJsonObject` that represents the JSON object, if the value is an object.
string	Returns the value as a string.
value	Returns the value, if the JSON value is a simple type. If you specify a default value and the value cannot be converted to the same type as the default, the default value is returned.

LuaJsonValue:new

The constructor for a `LuaJsonValue` object (creates a new `LuaJsonValue`).

Syntax

```
LuaJsonValue:new( value )
```

Arguments

Argument	Description
value	(lua_json_type) The initial value. This can be a Boolean, float, integer, string, LuaJsonArray, LuaJsonObject, LuaJsonValue, or nil.

Returns

(LuaJsonValue). The new LuaJsonValue object.

array

The array method gets the array, if the value is a JSON array.

Syntax

```
array()
```

Returns

(LuaJsonArray). A LuaJsonArray object that represents the array.

Example

```
local fh = io.open("example_array.json", "r")
local file_content = fh:read("*all")
fh:close()

local myJsonValue = parse_json(file_content)

if myJsonValue:is_array() then
   print "The value is a JSON array"
   local myJsonArray=myJsonValue:array()
   print (myJsonArray:string())
end
```

copy

The copy method returns a copy of the value. Modifying the copy does not affect the original value.

Syntax

```
copy()
```

Returns

(LuaJsonValue). A copy of the value.

exists

The `exists` method checks whether a specified path exists in the JSON value.

Syntax

```
exists( pathElements )
```

Arguments

Argument	Description
`pathElements`	(json_path_string_or_integer) The path to check. Specify one or more path elements, which might be object attribute names (strings) or array indexes (integers).

Returns

(Boolean) Returns `true` if the path exists, `false` otherwise.

Example

Consider the following JSON:

```
{
  "product": "IDOL",
  "component": "CFS",
  "version": { "major": 11, "minor": 3 },
  "ports": [
    { "type": "ACI", "number": 7000 },
    { "type": "Service", "number": 17000 }
  ]
}
```

If this JSON is represented by a `LuaJsonValue` object named `myJsonValue`, you could use the `exists` method as follows:

```
print (myJsonValue:exists("ports",0,"type"))
-- true (LuaJsonArrays are zero-indexed)

print (myJsonValue:exists("ports",2,"type"))
-- false (LuaJsonArrays are zero-indexed, and there is no third value)

print (myJsonValue:exists("version", "major"))
-- true
```

See Also

- existsByPath, below

existsByPath

The `existsByPath` method checks whether a specified path exists in the JSON value.

Syntax

```
existsByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path to check. Construct the path from array indexes and object attribute names, and use a slash (`/`) as the separator. If an object attribute name includes a slash, then escape the slash with a backslash, for example `"one\\/two"`. Notice that you must also escape a backslash with another backslash.

Returns

(Boolean) Returns `true` if the path exists, `false` otherwise.

Example

Consider the following JSON:

```
{
  "product": "IDOL",
  "component": "CFS",
  "version": { "major": 11, "minor": 3 },
  "ports": [
    { "type": "ACI", "number": 7000 },
    { "type": "Service", "number": 17000 }
```

```
  ]
}
```

If this JSON is represented by a `LuaJsonValue` object named `myJsonValue`, you could use the `existsByPath` method as follows:

```
print (myJsonValue:existsByPath("ports/0/type"))
-- true (LuaJsonArrays are zero-indexed)

print (myJsonValue:existsByPath("ports/2/type") )
-- false (LuaJsonArrays are zero-indexed, and there is no third value)

print (myJsonValue:existsByPath("version/major"))
-- true
```

See Also

- exists, on page 202

is_array

The `is_array` method whether the value is a JSON array.

Syntax

```
is_array()
```

Returns

(Boolean). Returns `true` if the value is an array value, `false` otherwise.

is_boolean

The `is_boolean` method returns whether the value is a Boolean value.

Syntax

```
is_boolean()
```

Returns

(Boolean). Returns `true` if the value is a Boolean value, `false` otherwise.

is_float

The `is_float` method returns whether the value is a floating point value.

Syntax

```
is_float()
```

Returns

(Boolean). Returns `true` if the value is a floating point value, `false` otherwise.

is_integer

The `is_integer` method returns whether the value is an integral value.

Syntax

```
is_integer()
```

Returns

(Boolean). Returns `true` if the value is an integer value, `false` otherwise.

is_null

The `is_null` method returns whether the value is a null value.

Syntax

```
is_null()
```

Returns

(Boolean). Returns `true` if the value is a null value, `false` otherwise.

is_number

The `is_number` method returns whether the value is a number.

Syntax

```
is_number()
```

Returns

(Boolean). Returns `true` if the value is a number, `false` otherwise.

is_object

The `is_object` method returns whether the value is a JSON object.

Syntax

```
is_object()
```

Returns

(Boolean). Returns `true` if the value is a JSON object, `false` otherwise.

is_simple_value

The `is_simple_value` method returns whether the value is a simple value (Boolean, float, integer, or string).

Syntax

```
is_simple_value()
```

Returns

(Boolean). Returns `true` if the value is a simple value, `false` otherwise.

is_string

The `is_string` method returns whether the value is a string value.

Syntax

```
is_string()
```

Returns

(Boolean). Returns `true` if the value is a string value, `false` otherwise.

lookup

The `lookup` method returns the value at the specified path in the JSON value.

> **NOTE:**
> This method does not make a copy of the value, so modifying the returned value affects the original object.

Syntax

```
lookup( pathElements )
```

Arguments

Argument	Description
`pathElements`	(json_path_string_or_integer) The path of the value to return. Specify one or more path elements, which might be object attribute names (strings) or array indexes (integers).

Returns

(LuaJsonValue) Returns the value that exists at the specified path, or `nil` if the specified path does not exist.

Example

```
local myJsonObject = LuaJsonObject:new()
myJsonObject:assign( {
        attr1=LuaJsonObject:new( { n=42, x=5 } ),
        attr2=LuaJsonArray:new( 1, 2, "three" ),
    } )

local myJsonValue = LuaJsonValue:new( myJsonObject )

print ( myJsonValue:lookup("attr2",2):value() )
-- three (LuaJsonArrays are zero-indexed)

-- Modifying the returned value changes the original object
local lookup = myJsonValue:lookup("attr1"):object()
```

```
lookup:assign("y",3)
print (myJsonValue:string())
-- {"attr1":{"n":42,"x":5,"y":3},"attr2":[1,2,"three"]}
```

See Also

- lookupByPath, below

lookupByPath

The `lookupByPath` method returns the value at the specified path in the JSON array.

> **NOTE:**
> This method does not make a copy of the value, so modifying the returned value affects the original object.

Syntax

```
lookupByPath( path )
```

Arguments

Argument	Description
`path`	(string) The path of the value to return. Construct the path from array indexes and object attribute names, and use a slash (`/`) as the separator. If an object attribute name includes a slash, then escape the slash with a backslash, for example `"one\\/two"`. Notice that you must also escape a backslash with another backslash.

Returns

(LuaJsonValue) Returns the value that exists at the specified path, or `nil` if the specified path does not exist.

Example

```
local myJsonObject = LuaJsonObject:new()
myJsonObject:assign( {
        attr1=LuaJsonObject:new( { n=42, x=5 } ),
        attr2=LuaJsonArray:new( 1, 2, "three" ),
    } )

local myJsonValue = LuaJsonValue:new( myJsonObject )
```

```
print ( myJsonValue:lookupByPath("attr2/2"):value() )
-- three (LuaJsonArrays are zero-indexed)

-- Modifying the returned value changes the original object
local lookup = myJsonValue:lookupByPath("attr1"):object()
lookup:assign("y",3)
print (myJsonValue:string())
-- {"attr1":{"n":42,"x":5,"y":3},"attr2":[1,2,"three"]}
```

See Also

- lookup, on page 207

object

The `object` method gets the object, if the value is a JSON object.

Syntax

```
object()
```

Returns

(LuaJsonObject) A LuaJsonObject object that represents the JSON object.

Example

```
local fh = io.open("example_json.json", "r")
local file_content = fh:read("*all")
fh:close()

local myJsonValue = parse_json(file_content)

if myJsonValue:is_object() then
   print "The value is a JSON object"
   local myJsonObject=myJsonValue:object()
   print (myJsonObject:string())
end
```

string

The `string` method returns the JSON value as a string.

Syntax

```
string()
```

Returns

(String). The JSON value as a string.

Example

```
local fh = io.open("example_json.json", "r")
local file_content = fh:read("*all")
fh:close()

local myJsonValue = parse_json(file_content)

print(myJsonValue:string())
```

value

The `value` method gets the value, if the JSON value is a simple type (a Boolean, float, integer, or string). This method returns a default value if one is specified and the value cannot be converted to the same type as the default.

Syntax

```
value( [default] )
```

Arguments

Argument	Description
`default`	(lua_json_simple_type) The default value to return, if the value cannot be converted to the same type as the default. You can specify a Boolean, float, integer, string, or nil.

Returns

(lua_json_simple_type). A Boolean, float, integer, string, or nil.

LuaLog Methods

A `LuaLog` object provides the capability to use a log stream. You can obtain a `LuaLog` object for a log stream by using the function get_log.

If you have a `LuaLog` object called `log` you can call its methods using the `':'` operator. For example:

```
log:write_line(level, message)
```

Method	Description
write_line	Write a message to the log stream.

write_line

The `write_line` method writes a message to the log stream.

Syntax

```
write_line( level, message )
```

Arguments

Argument	Description
`level`	The log level for the message. The message only appears in the log if the log level specified here is the same as, or higher than, the log level set for the log stream. To obtain the correct value for the log level, use one of the following functions: • `log_level_always()` • `log_level_error()` • `log_level_warning()` • `log_level_normal()` • `log_level_full()`
`message`	(string) The message to write to the log stream.

Example

```
local config = get_config("connector.cfg")
local log = get_log(config, "SynchronizeLogStream")
log:write_line( log_level_error() , "This message is written to the synchronize
log")
```

LuaLogService Methods

A `LuaLogService` object provides the capability to write messages to a standard or custom log file.

If you have a `LuaLogService` object called `myLogService` you can call its methods using the `':'` operator. For example:

```
myLogService:get_log("application")
```

Constructor	Description
LuaLogService:new	The constructor for a `LuaLogService` object (creates a new `LuaLogService`).

Method	Description
get_log	Returns a LuaLog object that you can use to write log messages to a specified log type.

LuaLogService:new

The constructor for a `LuaLogService` object (creates a new `LuaLogService`).

> **IMPORTANT:**
> Do not create a new log service to write messages to standard log types. Obtain a `LuaLog` object through the function get_log instead.

Syntax

```
LuaLogService:new( config )
```

Arguments

Argument	Description
`config`	(LuaConfig) The configuration object to read the logging settings from. You can obtain a LuaConfig object through the function get_config.

Returns

(LuaLogService). The new `LuaLogService` object.

Example

The following example creates a `LuaLogService` object to write log messages to a custom log file named `lua.log`. Declare the log service globally to avoid a `LuaLogService` object being created every time the script runs. For example, if you are writing a Lua script to use with a connector or CFS, create the `LuaLogService` outside the `handler` function. Otherwise, the server creates a `LuaLogService` object for every document that is processed.

```
local luaConfigString = [===[
[Logging]
LogLevel=FULL
0=LuaLogStream

[LuaLogStream]
LogTypeCSVs=lua
LogFile=lua.log
]===]

local config = LuaConfig:new(luaConfigString)

-- global log service instance for Lua log stream
luaLogService = LuaLogService:new(config)

local luaLog = luaLogService:get_log("lua")
luaLog:write_line(log_level_normal(), "running Lua script")
```

get_log

The `get_log` method returns a LuaLog object that you can use to write log messages to a specified log type.

Syntax

```
get_log( type )
```

Arguments

Argument	Description
`type`	(string) The log type.

Returns

(LuaLog). A LuaLog object.

Example

```
local log = myLuaLogService:get_log("import")
-- import is a standard log type for CFS
```

LuaRegexMatch Methods

A `LuaRegexMatch` object provides information about the matches for a regular expression found in a string. For example, the `regex_search` function returns a `LuaRegexMatch` object.

If a match is found for a regular expression at multiple points in the string, you can use the `next()` method to get a `LuaRegexMatch` object for the next match.

If the regular expression contained sub-expressions (surrounded by parentheses) the methods of `LuaRegexMatch` objects can also be used to retrieve information about the sub-expression matches.

If you have a `LuaRegexMatch` object called `match` you can call its methods using the "`:`" operator. For example:

```
match:length()
```

Method	Description
length	Returns the length of the sub match.
next	Returns a LuaRegexMatch for the next match.
position	Returns the position of the sub match as an index from 1.
size	Returns the number of sub matches for the current match.
str	Returns the string for the sub match.

length

The `length` method returns the length of the match. You can also retrieve the length of sub matches by specifying the `submatch` parameter.

Syntax

```
length( [ submatch ] )
```

Arguments

Argument	Description
`submatch`	(number) The sub match to return the length of, starting at 1 for the first sub match. With

Argument	Description
	the default value of 0 the length of the whole match is returned.

Returns

(Number). The length of the sub match.

next

The `next` method returns a LuaRegexMatch object for the next match.

Syntax

```
next()
```

Returns

(LuaRegexMatch). A LuaRegexMatch object for the next match, or `nil` if there are no matches following this one.

position

The `position` method returns the position of the match in the string searched, where 1 refers to the first character in the string. You can also retrieve the position of sub matches by specifying the `submatch` parameter.

Syntax

```
position( [ submatch ] )
```

Arguments

Argument	Description
`submatch`	(number) The sub match to return the position of, starting at 1 for the first sub match. With the default value of 0 the position of the whole match is returned.

Returns

(Number). The position of the submatch as an index from 1.

size

The `size` method returns the total number of sub matches made for the current match, including the whole match (sub match 0).

Syntax

```
size()
```

Returns

(Number). The number of sub matches for the current match.

str

The `str` method returns the value of the substring that matched the regular expression. You can also retrieve the values of sub matches by specifying the `submatch` parameter.

Syntax

```
str( [ submatch ] )
```

Arguments

Argument	Description
`submatch`	(number) The sub match to return the value of, starting at 1 for the first sub match. With the default value of 0 the value of the whole match is returned.

Returns

(String). The value of the sub match.

LuaXmlDocument Methods

This section describes the methods provided by `LuaXmlDocument` objects. A `LuaXmlDocument` object provides methods for accessing information stored in XML format. You can create a `LuaXmlDocument` from a string containing XML using the `parse_xml` function.

If you have a `LuaXmlDocument` object called `xml` you can call its methods using the `':'` operator. For example:

```
xml:root()
```

Method	Description
root	Returns a LuaXmlNode that is the root node of the XML document.
XPathExecute	Returns a LuaXmlNodeSet that is the result of supplied XPath query.
XPathRegisterNs	Register a namespace with the XML parser. Returns an integer detailing the error code.
XPathValue	Returns the first occurrence of the value matching the XPath query.
XPathValues	Returns the values according to the XPath query.

root

The `root` method returns an LuaXmlNode, which is the root node of the XML document.

Syntax

```
root()
```

Returns

(LuaXmlNode). A LuaXmlNode object.

XPathExecute

The `XPathExecute` method returns a LuaXmlNodeSet, which is the result of the supplied XPath query.

Syntax

```
XPathExecute( xpathQuery )
```

Arguments

Argument	Description
`xpathQuery`	(string) The xpath query to run.

Returns

(LuaXmlNodeSet). A LuaXmlNodeSet object.

XPathRegisterNs

The `XPathRegisterNs` method registers a namespace with the XML parser.

Syntax

```
XPathRegisterNs( prefix, location )
```

Arguments

Argument	Description
`prefix`	(string) The namespace prefix.
`location`	(string) The namespace location.

Returns

(Boolean). `True` if successful, `False` in case of error.

XPathValue

The `XPathValue` method returns the first occurrence of the value matching the XPath query.

Syntax

```
XPathValue( query )
```

Arguments

Argument	Description
`query`	(string) The XPath query to use.

Returns

(String). A string of the value.

XPathValues

The `XPathValues` method returns the values according to the XPath query.

Syntax

```
XPathValues( query )
```

Arguments

Argument	Description
query	(string) The XPath query to use.

Returns

(Strings). The strings can be assigned to a table. To map the return values to a table, surround the function call with braces. For example:

```
values = { xml:XPathValues(query) }
```

LuaXmlNodeSet Methods

A `LuaXmlNodeSet` object represents a set of XML nodes.

If you have a `LuaXmlNodeSet` object called `nodes` you can call its methods using the ':' operator. For example:

```
nodes:size()
```

Method	Description
at	Returns the LuaXmlNode at position `pos` in the set.
size	Returns size of node set.

at

The `at` method returns the `LuaXmlNode` at position `position` in the set.

Syntax

```
at( position )
```

Arguments

Argument	Description
`position`	(number) The index of the item in the array to get.

Returns

(LuaXmlNode).

size

The `size` method returns the size of the node set.

Syntax

```
size()
```

Returns

(Number) An integer, the size of the node set.

LuaXmlNode Methods

A `LuaXmlNode` object represents a single node in an XML document.

If you have a `LuaXmlNode` object called `node` you can call its methods using the `':'` operator. For example:

```
node:name()
```

Method	Description
attr	Returns the first LuaXmlAttribute attribute object for this element.
content	Returns the content (text element) of the XML node.
firstChild	Returns a LuaXmlNode that is the first child of this node.
lastChild	Returns a LuaXmlNode that is the last child of this node.
name	Returns the name of the XML node.
next	Returns a LuaXmlNode that is the next sibling of this node.
nodePath	Returns the XML path to the node that can be used in another `XPath` query.

Method	Description
parent	Returns the parent LuaXmlNode of the node.
prev	Returns a LuaXmlNode that is the previous sibling of this node.
type	Returns the type of the node as a string.

attr

The `attr` method returns the first LuaXmlAttribute attribute object for the LuaXmlNode. If the name argument is specified, the method returns the first LuaXmlAttribute object with the specified name.

Syntax

```
attr( [name] )
```

Arguments

Argument	Description
`name`	(string) The name of the LuaXmlAttribute object.

Returns

(LuaXmlAttribute).

content

The `content` method returns the content (text element) of the XML node.

Syntax

```
content()
```

Returns

(String). A string containing the content.

firstChild

The `firstChild` method returns the LuaXmlNode that is the first child of this node.

Syntax

```
firstChild()
```

Returns

(LuaXmlNode).

lastChild

The `lastChild` method returns the LuaXmlNode that is the last child of this node.

Syntax

```
lastChild()
```

Returns

(LuaXmlNode).

name

The `name` method returns the name of the XML node.

Syntax

```
name()
```

Returns

(String). A string containing the name.

next

The `next` method returns the LuaXmlNode that is the next sibling of this node.

Syntax

```
next()
```

Returns

(LuaXmlNode).

nodePath

The `nodePath` method returns the XML path to the node, which can be used in another XPath query.

Syntax

```
nodePath()
```

Returns

(String). A string containing the path.

parent

The `parent` method returns the parent LuaXmlNode of the node.

Syntax

```
parent()
```

Returns

(LuaXmlNode).

prev

The `prev` method returns a LuaXmlNode that is the previous sibling of this node.

Syntax

```
prev()
```

Returns

(LuaXmlNode).

type

The `type` method returns the type of the node as a string.

Syntax

```
type()
```

Returns

(String) A string containing the type. Possible values are:

```
element_node comment_node element_decl
attribute_node document_node attribute_decl
text_node document_type_node entity_decl
cdata_section_node document_frag_node namespace_decl
entity_ref_node notation_node xinclude_start
entity_node html_document_node xinclude_end
pi_node dtd_node docb_document_node
```

LuaXmlAttribute Methods

A `LuaXmlAttribute` object represents an attribute on an XML element.

If you have a `LuaXmlAttribute` object called `attribute` you can call its methods using the `':'` operator. For example:

```
attribute:name()
```

Method	Description
name	Returns the name of this attribute.
next	Returns a LuaXmlAttribute object for the next attribute in the parent element.
prev	Returns a LuaXmlAttribute object for the previous attribute in the parent element.
value	Returns the value of this attribute.

name

The `name` method returns the name of this attribute.

Syntax

```
name()
```

Returns

(String). A string containing the name of the attribute.

next

The `next` method returns a LuaXmlAttribute object for the next attribute in the parent element.

Syntax

```
next()
```

Returns

(LuaXmlAttribute).

prev

The `prev` method returns a LuaXmlAttribute object for the previous attribute in the parent element.

Syntax

```
prev()
```

Returns

(LuaXmlAttribute).

value

The `value` method returns the value of this attribute.

Syntax

```
value()
```

Returns

(String). A string containing the value of the attribute.

Appendix A: Document Fields

The connector adds the following fields to each document that it ingests:

Field Name	Description
AUTN_IDENTIFIER	An identifier that allows a connector to extract the document from the repository again, for example during the collect or view actions.
DocTrackingId	An identifier used for document tracking functionality.
DREREFERENCE	A reference for the document. This is the standard IDOL reference field, which is used for deduplication.
source_connector_run_id	(Added only when IngestSourceConnectorFields=TRUE). The asynchronous action token of the fetch action that ingested the document.
source_connector_server_id	(Added only when IngestSourceConnectorFields=TRUE). A token that identifies the instance of the connector that retrieved the document (different installations of the same connector populate this field with different IDs). You can retrieve the UID of a connector through action=GetVersion.

Glossary

A

ACI (Autonomy Content Infrastructure)
A technology layer that automates operations on unstructured information for cross-enterprise applications. ACI enables an automated and compatible business-to-business, peer-to-peer infrastructure. The ACI allows enterprise applications to understand and process content that exists in unstructured formats, such as email, Web pages, Microsoft Office documents, and IBM Notes.

ACI Server
A server component that runs on the Autonomy Content Infrastructure (ACI).

ACL (access control list)
An ACL is metadata associated with a document that defines which users and groups are permitted to access the document.

action
A request sent to an ACI server.

active directory
A domain controller for the Microsoft Windows operating system, which uses LDAP to authenticate users and computers on a network.

C

Category component
The IDOL Server component that manages categorization and clustering.

Community component
The IDOL Server component that manages users and communities.

connector
An IDOL component (for example File System Connector) that retrieves information from a local or remote repository (for example, a file system, database, or Web site).

Connector Framework Server (CFS)
Connector Framework Server processes the information that is retrieved by connectors. Connector Framework Server uses KeyView to extract document content and metadata from over 1,000 different file types. When the information has been processed, it is sent to an IDOL Server or Distributed Index Handler (DIH).

Content component
The IDOL Server component that manages the data index and performs most of the search and retrieval operations from the index.

D

DAH (Distributed Action Handler)
DAH distributes actions to multiple copies of IDOL Server or a component. It allows you to use failover, load balancing, or distributed content.

DIH (Distributed Index Handler)
DIH allows you to efficiently split and index extremely large quantities of data into multiple copies of IDOL Server or the Content component. DIH allows you to create a scalable solution that delivers high performance and high availability. It provides a flexible way to batch, route, and categorize the indexing of internal and external content into IDOL Server.

I

IDOL
The Intelligent Data Operating Layer (IDOL) Server, which integrates unstructured, semi-structured and structured information from multiple repositories through an understanding of the content. It delivers a real-time environment in which operations across applications and content are automated.

IDOL Proxy component
An IDOL Server component that accepts incoming actions and distributes them to the appropriate subcomponent. IDOL Proxy also performs some maintenance operations to make sure that the subcomponents are running, and to start and stop them when necessary.

Import
Importing is the process where CFS, using KeyView, extracts metadata, content, and sub-files from items retrieved by a connector. CFS adds the information to documents so that it is indexed into IDOL Server. Importing allows IDOL server to use the information in a repository, without needing to process the information in its native format.

Ingest
Ingestion converts information that exists in a repository into documents that can be indexed into IDOL Server. Ingestion starts when a connector finds new documents in a repository, or documents that have been updated or deleted, and sends this information to CFS. Ingestion includes the import process, and processing tasks that can modify and enrich the information in a document.

Intellectual Asset Protection System (IAS)
An integrated security solution to protect your data. At the front end, authentication checks that users are allowed to access the system that contains the result data. At the back end, entitlement checking and authentication combine to ensure that query results contain only documents that the user is allowed to see, from repositories that the user has permission to access. For more information, refer to the IDOL Document Security Administration Guide.

K

KeyView
The IDOL component that extracts data, including text, metadata, and subfiles from over 1,000 different file types. KeyView can also convert documents to HTML format for viewing in a Web browser.

L

LDAP
Lightweight Directory Access Protocol. Applications can use LDAP to retrieve information from a server. LDAP is used for directory services (such as corporate email and telephone directories) and user authentication. See also: active directory, primary domain controller.

License Server
License Server enables you to license and run multiple IDOL solutions. You must have a License Server on a machine with a known, static IP address.

O

OmniGroupServer (OGS)
A server that manages access permissions for your users. It communicates with your repositories and IDOL Server to apply access permissions to documents.

P

primary domain controller
A server computer in a Microsoft Windows domain that controls various computer resources. See also: active directory, LDAP.

V

View
An IDOL component that converts files in a repository to HTML formats for viewing in a Web browser.

W

Wildcard
A character that stands in for any character or group of characters in a query.

X

XML
Extensible Markup Language. XML is a language that defines the different attributes of document content in a format that can be read by humans and machines. In IDOL Server, you can index documents in XML format. IDOL Server also returns action responses in XML format.

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 Administration Guide (HPE File System Connector 11.4)

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 AutonomyTPFeedback@hpe.com.

We appreciate your feedback!