



Coveo Enterprise Search 6.0

Open Connector API

Contents

Building a Connector	4
How to Create a Project	4
How to Create a Connector Class	4
How to Implement the Run Method	4
How to Retrieve Starting Addresses	5
How to Crawl Content	5
How to Create a Document.....	6
How to Index a Document.....	7
Advanced Topics	8
How to Choose a Uri Pattern	8
What Are URI Types and Filters	8
How to Set Data on the Document	9
How to Retrieve Fields and Metadata.....	10
How to Log Messages/Warnings/Errors	10
How to Get User Identities	11
Rebuild vs Refresh.....	11
How to Manually Verify Document Changes	11
How to Implement Pause/Resume	12
How to Use Persistent Data.....	13
How to Implement Live Indexing.....	13
How to Get Source Parameters.....	13
How to Implement the PerformInitialize Method	13
How to Implement the Update Method	14
How to Handle Attachments	14
How to Manage Document Types.....	14
How to Set the Security	15
How to Use the CheckPoint Method.....	16
How to Handle Exceptions.....	16
How to Use DocumentMapping	17
How to Implement Refresh Document/Folder.....	18
How to Implement Delete Document/Folder.....	18
What is Automatic Deletion of Removed Documents	18
How to Manually Remove Documents.....	18
Interfaces	19
ISaveRestoreState Interface.....	19
ILiveIndexing Interface.....	20
IDeleteDocument and IDeleteFolder Interfaces.....	20

IRefreshDocument and IRefreshFolder Interfaces	20
IBeautifyURI Interface	21
IHasPersistentData Interface	21
ISaveRestorePersistentData Interface	22
General Knowledge.....	23
What is a Connector	23
How to Add an Additional Connector	23
Why Use a ClassLibrary	24
Why Retrieve Metadata	24
User Identities in CES	24
Samples.....	24
Empty Project.....	24
Gmail Pop3 Connector.....	25

Coveo Enterprise Search (CES) features an API that enables the development of additional connectors. The Open Connector API allows developers to write their own [connectors](#)—simple but efficient ones, as well as more complex ones—in order to crawl specific objects, repositories, etc. that are not supported by CES. Coveo uses the API to develop its own connectors—*Documentum*, *Symantec Enterprise Vault*, *Microsoft Exchange*, *Database*, *Salesforce*, *Quest Archive Manager* and *Lotus Notes*.

Building a Connector

This section of the Open Connector API documentation guides you through the creation of a simple [connector](#). Moreover, it explains how to integrate it to CES, as well as crawl with it. The following lists the different steps required to develop a simple connector:

How to Create a Project

The first step to perform in order to develop a connector is to create a project. You can use the [empty template project](#) and skip directly to [How to Retrieve Starting Addresses](#).

Connectors are written using *Visual Studio 2005* or *2008*. They can be written in any language that is *.Net* compatible; however, in this documentation, examples are written in *C#*.

Create a new *C#* project of *ClassLibrary* type. The output of this type of project is a *dynamically loaded library* (dll). Once the project is created, add the Open Connector API references, which are located in the *Coveo Enterprise Search* bin folder.

- *Coveo.CES.CustomCrawlers.dll*
- *Coveo.CES.Interops.dll*
- *Coveo.CNL.dll*

How to Create a Connector Class

This step is intended to help implement a new connector class to the project as a project is empty when it is created. In order to mark a new class as a connector, it is mandatory to inherit from the *CustomCrawler* class.

Inheriting from CustomCrawler

```
using Coveo.CES.CustomCrawlers;
namespace MyConnector
{
    public class MyConnectorClass : CustomCrawler
    {
    }
}
```

By doing so, the class is marked as a connector; however, note that it cannot be compiled yet. For more information, refer to [How to Implement the Run Method](#).

How to Implement the Run Method

In order for the project to compile, it is important to implement the *Run* method. The *Run* method is the main method of a connector; meaning that on a source *Rebuild* or *Refresh*, this method is automatically called. All the steps required to crawl the target system must occur in the scope of this method.

Empty Run Method

```
protected override void Run()
{
    // Crawl content.
}
```

Gmail Pop3 Connector Run Method


```
protected override void Run()
{
    // For the Gmail pop3 server, adding recent: before the username
    // allows for retrieving all the mails received in the last 30 days.
    string recentModeLogin = string.Format("recent:{0}", Context.UserName);
    StartCrawling(recentModeLogin);
}
```

When this method is overridden, the project can compile and [the newly created connector can be added to CES](#).

The method is empty; therefore, nothing happens. To crawl specific content, refer to [How to Retrieve Starting Addresses](#).

How to Retrieve Starting Addresses

Once the Run method is implemented, it is important to retrieve the starting addresses specified in the

Add Source page, which can be accessed by clicking  in the **Sources** section of the **Sources and Collections** page (Administration Tool > Index > Sources and Collections); therefore, the connector knows which addresses to crawl.

To retrieve information from the source configuration, the **CrawlingContext** class must be used. The **CustomCrawler** class already has the Context property, which allows the user to retrieve the necessary contextual information.

The URI property is an array of strings containing all the starting addresses specified in the source crawled by the connector:

Getting the starting addresses

```
protected override void Run()
{
    foreach (string address in Context.Uris ) {
        // Crawl the specified address.
        Crawl(address);
    }
}
```

The format of the URIs is defined by the person writing the connector and is intended to match the targeted system. For more information on how to choose a good pattern, refer to [How to Choose a URI Pattern](#).

The Run method now has the necessary information to target specific documents within the crawled system. The next step is to crawl content using these URIs.

How to Crawl Content

Once the URIs are retrieved from the source, it is only logical to crawl their content. This section of the connector is not really related to the Open Connector API itself; therefore, it is important, as a developer, to have the appropriate knowledge of the targeted system as well as its APIs.

The main idea behind crawling consists of connecting to a remote system, looping in order to retrieve all the documents and indexing them until all of them have been crawled. The following details the steps to follow:

1. Establish a connection to the remote system by using specific credentials. Credentials in the Open Connector API come from the [CES user identities](#) and can be [retrieved from the context](#).
2. Within the loop, retrieve the data requested using the URIs and [create a document](#). In order to increase the quality of the searches, it is important to retrieve as much [metadata](#) as possible.

Communication Example

```
private void Crawl(string p Uri)
{
    Connection connection = EstablishConnection(p Uri, user, password, domain);
    while (connection.HasData) {
        // Retrieve the data.
        Data data = connection.GetNextItemToIndex();
        // Index this data.
        IndexData(data);
    }
}
```

Getting the next item

```
#GmailPop3Connector.GettingTheNextItem
```

How to Create a Document

An instance of the Document class is an item to index, for example a *Microsoft Word* document or database entry. These documents are displayed when a search is performed in CES.

To index a document, CES requires these two properties to have values:

- **URI:** The URI is a unique identifier for a document. It is the first property to set after creating a document. For more information concerning the way to select a good URI for your documents, refer to [How to Choose a Uri Pattern](#). For more information concerning the different types of URIs as well as their interactions with filters in CES, refer to [What Are URI Types and Filters](#).
- **Security:** At least one entity must have access to the document; otherwise, there is no reason to put it in the index. For more information concerning the way to set securities on a document, refer to [How to Set the Security](#).

The following lists other important properties in terms of usability:

- **Data:** Represents the body of the document. There are several ways of putting the data in the document. For more information, refer to [How to Set Data on the Document](#).
- **Metadata and Fields:** Adds searchable information to the documents. For more information, refer to [Why Retrieve Metadata](#), [fields can also be directly set](#) with the metadata.
- **FileName:** Automatically assigns a title to the document. The filename is also useful to identify the [kind of indexed documents](#).
- **ModifiedDate:** Quickly determines if a document has been modified in a subsequent crawling run. It is necessary to [manually determine if a document must be indexed](#).

Creating a Document

```
private void IndexData(Data p_Data)
{
    // Documents should always be created like this.
    using (Document doc = new Document(Context)) {
        // Set the unique Uri.
        doc.Uri = p_Data.Uri;
        // Allow everyone to see the document.
        doc.AddACLEntry(new ACLEntry());
        // Set the Data.
        doc.Data = p_Data.Data;
        // Set the ModifiedDate.
        doc.ModifiedDate = p_Data.ModifiedDate;
        // Set the filename.
        doc.FileName = p_Data.FileName;
        // Set the Metadata.
        doc.Metadata = p_Data.Metadata;
    }
}
```

Creating a Gmail Pop3 Document

```
#GmailPop3Connector.CreateADocument
```

Once the document is created, the next step is to index the document.

How to Index a Document

Once the instance of the document has been completed, it is ready to be indexed. The `IndexDocument` method of the `CustomCrawler` class is responsible of the indexing. It takes a document in parameter and performs several verifications. If nothing unusual is found, then the document is indexed. Documents that need to be re-indexed are automatically flagged and indexed, otherwise they are marked as unchanged in the log.

Indexing a Document

```
// Documents should always be created like this.
using (Document doc = new Document(Context)) {
    // ...Filling the document.
    // Index the document.
    IndexDocument(doc);
}
```

The last step required to index all the documents is to loop around the `IndexDocument` function in order to index the remaining documents targeted by the starting addresses. Afterwards, the connector will work properly. It can now be compiled and [added to CES](#). In order for the connector to be more efficient and robust, several features can be added to it. These features are covered in the [Advanced Topics](#) section. The following is a brief overview of the two most important ones:

- [Pause/Resume](#): This feature is used to interrupt a crawling run. Pause operations are called (if available) when a Refresh/Rebuild is paused, the index becomes read-only or CES shuts down. When crawling resumes, the connector starts where it had left of instead of restarting from scratch.
- [Live Indexing](#): This feature is the way used by CES to keep the index up to date with the content of its sources. Live indexing consists of retrieving the modifications since the last Refresh/Rebuild or the last live indexing.

Several other advanced features can help improve the connector.

Advanced Topics

This section provides important information concerning more advanced features related to the development of connectors. Note that these features are not mandatory for the connector to work; however they are very useful and can help improve its robustness and efficiency.

How to Choose a Uri Pattern

Documents need to have a valid URI (Uniform Resource Identifier) in order to be indexed. The URI, which must be unique, is the property that differentiates documents. It is assigned by the connector to every [newly created document](#). For example, it is possible to identify an *Exchange* email by using this URI: <https://owa.acme.com/exchange/joSmith@coveo.com/Inbox/b5aee0f1efec674b8397cf4bfef8b5b800001c099fe>.

In this identifier, different sections are required. The following details the URI:

- **https://**: Scheme of the URI. Other typical schemes are *file://*, *http://*, etc. CES supports custom schemes; therefore *acme://* is a valid scheme.
- **owa.acme.com/exchange/joSmith@coveo.com/Inbox/**: Each segment, separated by the / character, corresponds to a specific folder. It is not required to keep the folder architecture in the URI as long as it is unique; however in the index browser, the folder structure will be recreated, making it easier to browse content from the Administration Tool. Also, the folder structure can be used to fill a [metadata field](#).
- **b5aee0f1efec674b8397cf4bfef8b5b800001c099fe**: As it is the last segment, this section corresponds to the document itself. It can be the filename; however, note that in the case of *Exchange* documents, emails can have the same name, which prevents its use as a unique identifier.

Different difficulties can arise when choosing a URI pattern as each document must be assigned a unique URI. The following presents thoughts that can guide you in finding a unique URI for each document:

- *Can two files have the same name?* If two files, or a file and a folder, have the same value in the attribute used, it is not a good attribute for the URI pattern.
- *Can folder or file names contain the / character?* Because the / character is automatically counted as a delimiter between each segment, it is suggested to replace the / character.
- *Can items have attachments?* There is a specific pattern to follow for attachments, it is described in [How to Handle Attachments](#).

What Are URI Types and Filters

Filtering is an important operation in CES as it allows users to ignore or include specific documents. Filters can be customized in the source configuration. This operation is not applied by default, the connector has to test the files it finds against these filters; meaning the strings entered in the filter must be compared with the specific URI set on the document. Two different methods allow the connector to test a document against a filter; however they must be called:

- **GetActionForDocument**: This method also validates other attributes and returns an Action object, which corresponds to what is done with the document. It is described in detail in [How to Manually Verify Document Changes](#).
- **IsUriFiltered**: This method validates if a specific string is filtered by the filters set on the current source. It is described in details in [How to Manually Verify Document Changes](#).

A document has 3 different URI attributes that can be filtered:

- **URI:** This is the unique identifier of a document, as explained in [How to Choose a URI Pattern](#).
- **PrintableUri:** This URI is optional and does not need to be unique. For example, a URI can contain ID numbers or guids that are not meaningful for a user. If so, set the PrintableUri as the name equivalent in order for the URI to be meaningful.
- **ClickableUri:** Normally, the link of a document is its URI. If that URI is customized, it might not be pertaining; therefore it has no impact when clicked. The ClickableUri should be set to a URI that allows the document to be opened. It is optional and does not need to be unique.

When using the filtering method, either of these three URIs can be used. By default, the unique URI prevails. To select a different one, use the *FilterBy* property on the document, which allows to select which URI type will filter the document. A fourth choice to *FilterBy* exists: *IgnoreFilter*. When this value is selected, the *GetActionForDocument* method does not test the filters; however it is important to validate them by using *IsUriFiltered*.

FilterBy

```
private void NewDocument(Data p_Data)
{
    using (Document doc = new Document(Context)) {
        doc.Uri = p_Data.Uri;
        doc.PrintableUri = p_Data.PrintableUri;
        doc.FilterBy = FilterByUri.PrintableUri;
    }
}
```

How to Set Data on the Document

The first way to set the data on a document is by directly setting it into the Data attribute. Basically, assign an array of bytes which corresponds to the data. This is pretty simple, but should be used sparingly. If the files have a chance of being big in the target system, do not set the data this way as it will use an important amount of memory.

Data

```
private void SetData(Data p_Data, Byte[] p_Array)
{
    using (Document doc = new Document(Context)) {
        doc.Uri = p_Data.Uri;
        doc.Data = p_Array;
    }
}
```

Another method used to set data is to use streams or a string:

- The first version of the *StreamData* method receives a string in parameter and directly sets it as the data. Note that it displays the same problems as the table of bytes; however it saves some manipulation.
- The second version receives a stream in parameter. It then writes this stream into the data as the data is made available. This has the advantage of not using much memory, no matter how big the file is.
- The third version also takes a stream in parameter; however it takes an integer as well, in case only a specific number of bytes must be written from the stream.

StreamData

```
private void SetStream(Data p_Data, Stream p_Stream)
{
    using (Document doc = new Document(Context)) {
        doc.Uri = p_Data.Uri;
        doc.StreamData(p_Stream);
    }
}
```

How to Retrieve Fields and Metadata

In order to power the search, CES uses system fields and metadata. Document conversion automatically fills certain fields; however a connector can specifically set values on specific system fields. The more fields are filled, the more options are available in the search in order to refine a search.

The `MetaData` property of the `Document` class can be set with a `Hashtable` containing the metadata name as well as its value. If the **Include metadata in document** checkbox is selected in the source configuration, all the metadata is added to the document; therefore when performing a search, if the query matches the metadata, the appropriate document is displayed. This is the basic and easiest way to use metadata.

Setting MetaData

```
private void SetMetaData(Document p_Doc)
{
    Hashtable metaData = m_Connection.GetMetaData();
    p_Doc.Metadata = metaData;
}
```

To get the most out of the available metadata, system fields and custom fields can be filled. To fill the system fields, use the `SetField` method on the document. The names of the already available fields are in `SystemFields` enum. Any value can be entered as the field name, but some work with the custom fields will be necessary to make them appear in the search.

SetField

```
private void SetField(Document p_Doc, Hashtable p_Meta)
{
    if (p_Meta.Contains("Author")) {
        doc.SetField(SystemFields.SysAuthor, p_Meta["Author"].ToString());
    }
}
```

In order for the custom fields to be automatically filled, the metaname of the custom field must match the metadata name of the hashtable.

How to Log Messages/Warnings/Errors

It can be useful for a connector to write feedback in the CES logs. This information can be used to troubleshoot issues or simply monitor the logs. There are multiple methods available in the `CrawlingContext` class to do so:

- **LogMessage:** This is the most generic logging method with which all possible results can be achieved. It requires the most parameters, but gives the required results.
- **LogCrawlerMessage:** This shortcut method uses `LogMessage`, but with the *Severity* of the entry set to *Normal*.
- **LogCawlerWarning:** This shortcut method uses `LogMessage`, but with the *Severity* of the entry set to *Warning*.
- **LogCrawlerErrorMessage:** This shortcut method uses `LogMessage`, but with the *Severity* of the entry set to *Error*.

Log a warning

```
private void LogSomeWarning(string p_Message, Document p_Doc)
{
    Context.LogMessage(p_Message, p_Doc.Uri, Severity.Warning,
        Operation.Unspecified);
}
```

How to Get User Identities

User identity is an important part of security in CES. It is frequently required by connectors in order to authenticate remote systems. To retrieve the values of a user identity, use the properties of the **CrawlingContext class** (use the Context property of the **CustomCrawler class**):

- **HasUserIdentity:** This property returns a Boolean value that indicates whether or not there is a specific user identity set on the source. If it is *True*, the Username and UserPassword will have values; else they will be empty and the identity of the user running CES should be used.
- **UserName:** This property contains the name of the user to crawl with. This is typically in the form *domain\username*.
- **UserPassword:** This is the password of the user to crawl with.

Example Title

```
private void Authenticate()
{
    if (Context.HasUserIdentity) {
        m_Connection.LogIn(Context.UserName, Context.UserPassword);
    }
}
```

Rebuild vs Refresh

Rebuild and Refresh are operations that both call the Run method of the connector; however, even though they go through the same method, there is a significant difference between them.

- **Rebuild:** Re-indexes everything, even if nothing has changed (it still check filters).
- **Refresh:** Only gets or indexes documents that have been modified or whose securities have changed.

To verify if your connector is currently processing a Rebuild or Refresh, use the ForceRefresh property, inherited from the **CustomCrawler class**. If it returns *True*, the operation is a Rebuild, otherwise it is a Refresh.

How to Manually Verify Document Changes

In order to develop the most efficient connector possible, it is important to avoid performing useless operations. For example, if it is possible to determine early on if a document needs to be indexed or not, or if it is possible to only fetch a specific portion of the information it stores. Usually, getting specific metadata allows to determine if the entire data needs to be downloaded; this can therefore save a lot of time.

The IndexDocument method verifies the modification date, ETag and filters; however it is the last possible moment in the crawling process to do so. To manually validate these values earlier, several methods are available:

- **IsOutOfDate:** This method is a member of the **Document class**. It requires to set the URI as well as the modified date on the document, otherwise the method simply returns *True*.

IsOutOfDate

```
private void CheckDate(Data p_Data)
{
    using (Document doc = new Document(Context)) {
        doc.Uri = p_Data.Uri;
        doc.ModifiedDate = p_Data.ModifiedDate;
        if (doc.IsOutOfDate()) {
            IndexDocument(doc);
        } else {
            ReportUnchangedDocument(p_Data.Uri);
        }
    }
}
```

- **GetActionForDocument:** This is the suggested method to determine what to do with a document. It requires to set the URI, ModifiedDate and optionally the [Document Type](#) or filename as its extension. It returns an action which can be one of three things following options:
 - **RetrieveAndIndex:** Download the data and get the metadata
 - **IndexByReference:** Get only the URIs and modified date
 - **Ignore:** Skip the document

GetActionForDocument

```
private void GetActionForDocument(Data p_PartialData)
{
    using (Document doc = new Document(Context)) {
        doc.Uri = p_PartialData.Uri;
        doc.ModifiedDate = p_PartialData.ModifiedDate;
        doc.FileName = p_PartialData.Filename;
        switch (GetActionForDocument(doc)) {
            case Action.RetrieveAndIndex:
                {
                    // Fetch all Data and index
                }
            case Action.IndexByReference:
                {
                    // Index without anything more.
                }
            case Action.Ignore:
                {
                    // Skip
                }
        }
    }
}
```

- **IsUriFiltered:** This method takes a string and returns True if it must be filtered and False if it must be indexed, as far as the filters go.

By using these methods when appropriate, the flow of the connector can be accelerated by keeping to a minimum the amount of data downloaded.

How to Implement Pause/Resume

This feature is used to interrupt a crawling run. Pause operations are called (if available) when a Refresh/Rebuild is paused, the index becomes read-only or CES is shutdown. When the crawling is resumed, the connector resumes from where it left off instead of restarting from scratch.

This feature is strongly recommended, as it can save a lot of time. Whenever the index goes into read-only mode or CES shutdowns, any running source without Pause/Resume has to restart from the beginning, which can be particularly frustrating if the source was big and had been running for a long time. Pause/Resume can prevent this problem from occurring.

To implement Pause/Resume, implement the [ISaveRestoreState Interface](#) on your connector.

How to Use Persistent Data

Persistent data is data kept on disk even after CES shuts down. Hence, it is available the next time the connector is used. This can be useful if the connector requires a cache or some other information in order to work properly. To keep persistent data, implement the [ISaveRestorePersistentData Interface](#) or [IHasPersistentData Interface](#).

How to Implement Live Indexing

CES uses live indexing in order to keep the index up to date. Live Indexing must be implemented by the connectors and it is strongly recommended to do so, as it gets the modifications that have occurred since the last Rebuild/Refresh or Live Indexing. Some target systems offer APIs to efficiently perform these operations; therefore allowing the connector to have a highly efficient Live Indexing.

To implement Live Indexing, implement the [ILiveIndexing Interface](#).

How to Get Source Parameters

It is possible to add custom source parameters that will be used by the connector. These parameters can be strings, integers or Booleans. For example, in order to be able to crawl, the connector can require the name of a database as well as the starting address. Getting the values of these source parameters is possible by using the **CrawlingContext class**, which is available to a connector when using the inherited Context property of the **CustomCrawler class**. The GetConfigValue method then allows the connector to retrieve these parameters:

GetConfigValue

```
private void GetAParameter()
{
    string acmeString = Context.GetConfigValue("acmeString");
    string acmeStringDefault = Context.GetConfigValue("acmeString", "defaultValue");
}
```

In the above example, the first GetConfigValue returns *null* if the "acmeString" parameter is not found. On the other hand, the second call returns "defaultValue" if the "acmeString" parameter is not found.

How to Implement the PerformInitialize Method

The PerformInitialize method is a virtual method of the **CustomCrawler class**. It is possible for a connector to override it in order to perform initialization operations. This method is called once when the connector is first started, loaded for the first time or when CES restarts. **This method must not throw exceptions.**

PerformInitialize

```
protected override void PerformInitialize()
{
    // Initialize some values
    m_Connection = new Connection(GetConnectionInfo());
}
```

How to Implement the Update Method

The Update method is a virtual method of the **CustomCrawler class**. It is possible for a connector to override it in order to receive updates when the source configuration is modified. It might not be called immediately, but it will be before another operation is performed. This is particularly useful to keep user identities and source parameters, as well as other structures that might depend on these values up to date in the connector. **This method must not throw exceptions.**

Update

```
protected override void Update()
{
    // Create a new connection based on the newly available parameters.
    m_Connection = new Connection(GetConnectionInfo());
}
```

How to Handle Attachments

Multiple systems can add attachments to their items in order to add information, files, etc. This parent/child link is often valuable and should be preserved if possible. CES supports this parent/child link; therefore it is important for the connector to preserve it when it crawls.

To add an attachment to a document, use the AddAttachment method on the parent document. This method takes another document in parameter; the attachment document. Once that is done, call IndexDocument on the parent.

Note: The URI attachments must be different from normal documents. Their unique URIs must correspond to the URI of the parent document, followed by the ? character and the attachment name.

Adding attachments

```
private void AddAttachment(Data p_AttachmentData, Document p_Owner)
{
    // Create the Attachment Document.
    Document attachment = null;
    try {
        attachment = new Document(Context);
        attachment.Uri = p_Owner.Uri + "?" + p_AttachmentData.FileName;
        // Add the Attachment
        p_Owner.AddAttachment(attachment);
    } catch (Exception) {
        if (attachment != null) {
            attachment.Dispose();
        }
        throw;
    }
    // Index the documents
    IndexDocument(p_Owner);
}
```

Note: Attachment documents are not scoped in a *using* block, as doing so destroys them before they are indexed. The parent document correctly disposes of all of its attachments. However, it is a user's responsibility to dispose of the current document in case of exceptions.

How to Manage Document Types

Document types are used to quickly classify a document in a category. It is possible to set actions based on document types directly in the CES Administration Tool. It is then up to the connector to make sure these settings are respected.

The `GetActionForDocument` method uses the Document Type and file extension to test a document against these settings. It is important to set these values in order for the settings and data in the index to be consistent.

How to Set the Security

Setting securities is often required by connectors in order for users to safely search documents without accessing restricted ones. Security is extremely important when crawling documents. Indeed, the connector has access to credentials that can access all the documents; however some of the documents have restricted permissions which are essential in order to keep the search secure.

It is important to have a good knowledge of the target system used in order to easily apply it to the securities of CES. Because CES is for *Windows* only, the securities kept are associated with *Windows* users. If the securities kept by the targeted system are also *Windows* securities, it is easy to retrieve and set them. The *.Net* [System.Security](#) namespace contains all the tools necessary to handle *Windows Aces* and *Security Descriptors*. If the securities of the system do not map directly to *Windows* securities, it is possible to create an external security provider in order to retrieve these permissions.

At least one instance of the `ACLEntry` class must be set on a document in order to index securities:

ACLEntry

```
private void AddACLEntry(CommonAce p_Ace, string p_ServerName, bool p_Allowed)
{
    using (Document doc = new Document(Context)) {
        doc.Uri = p_Data.Uri;
        ACLEntry security = new ACLEntry(SecurityType.Windows,
            p_Ace.SecurityIdentifier.Value, p_ServerName, p_Allowed);
        doc.AddACLEntry(security);
    }
}
```

In the above example, one way to create an `ACLEntry` is displayed:

- **SecurityType.Windows:** The `SecurityType` is an enum that specifies which type of security is entered in the `ACLEntry`:
 - **Windows:** A standard *Windows* security
 - **WindowsSID:** A *Windows* security in SID form.
 - **CustomGroup:** A custom security group that is resolved thanks to a mapping set in the Administration Tool.
 - **CustomUser:** A custom user that is mapped to a *Windows NT* security in the Administration Tool.
 - **ExternalGroup:** An external security group that is resolved by using an external security provider.
 - **External User:** An external user that is resolved by using an external security provider.
- **p_Ace.SecurityIdentifier.Value:** In this case, this ace value gives the *Windows* identity.
- **p_ServerName:** The name of the machine on which this security is found. This is useful for local users.
- **p_Allowed:** This is a Boolean value determining if it is an allowed or denied security. *True* allows the identity to access the document, while *False* prevents it from finding/accessing the document.

How to Use the CheckPoint Method

The CheckPoint method of the CustomCrawler class is an important tool that ensures the connector is responsive to the Administration Tool commands. It is during this method that the connector validates whether it needs to stop or not. This method is automatically called by IndexDocument; however depending on the operations performed by the connector, there can be a certain amount of time between IndexDocument calls. If someone issues a stop command while the connector is busy, it will not be effective until the connector calls CheckPoint.

If the ISaveRestoreState interface has been implemented, the CheckPoint method can also possibly call the SaveState method. The SaveState method is only called once every 50 CheckPoint calls.

In order to keep the connector responsive, it is important to call the CheckPoint method whenever the connector does not index documents for a certain period of time. It is a good practice to call CheckPoint() at the start of every loop found in the connector.

How to Handle Exceptions

Exceptions can arise during a crawling run for a variety of reasons. It is important to handle the exceptions correctly in order for the connector to be robust and efficient. It is also important to keep in mind that you do not want the crawling to stop in the middle of the process (mainly if you do not have [Pause/Resume](#)) because of an exception. Generally, all connector exceptions are split into two groups:

- **Fatal exceptions:** These exceptions are usually connection errors, not controlled by the connector. When something like that occurs, it is normal for the crawling process to log a message concerning it and stop. It is also possible to implement retries and delays in case it is a temporary issue.
- **Ignorable exceptions:** These exceptions represent all the little details that should not stop the crawling process. For example, an access denied to a specific document or an ignored folder. Skip the forbidden item and go to the next one. Corrupted items or random errors in the API of the target can be ignored.

The CustomCrawler class offers a few exception classes that can be useful when developing a connector:

- **CrawlerException:** This is the base class from which the other exception classes derive.
- **CrawlerFatalException:** This is the class used to signal something is not recoverable. In order to stop the crawling without performing [automatically deletion of documents](#), throw it out of the connector and in the CustomCrawler class.
- **CrawlerIgnorableException:** This is the class used to signal something that should not stop the crawling. Catch them, log a warning and resume the crawling.
- **CrawlerStopRefreshException:** When this type of exception is thrown at the CustomCrawler class, class, it aborts the crawling, but does not log anything in the logs.

Exceptions

```
private void HandleExceptions(string p_Uri)
{
    try {
        Crawl(p_Uri);
    } catch (CrawlerIgnorableException ex) {
        // We can continue crawling
        Context.LogMessage(ex.Message, p_Uri, Severity.Warning, Operation.Unspecified);
    } catch (CrawlerFatalException ex) {
        // Should stop crawling
        throw;
    }
}
```

Of course, these are only suggestions. It is possible to create customized exceptions to suit specific needs. What is important is for exceptions to be handled correctly in order for the connector to be robust and efficient.

How to Use DocumentMapping

The DocumentMapping class is another class of the Open Connector API that is used to set the fields on a document. If setup correctly, the mappings between metadata and fields are done automatically.

Mappings are important, as they can be used to fill the system fields with metadata. It might not be necessary to use a DocumentMapping to achieve this result, but it can greatly ease the task when dealing with database tables. To use DocumentMapping, create a new class that inherits from the DocumentMapping class. By doing so, you inherit the default mappings already available:

- filename
- uri
- uricaption
- clickableuri
- printableuri
- originaluri
- title
- body
- modifieddate
- indexmodifieddate
- contenttype

It is also possible to add any other mapping required to a connector. When mappings are being setup, what is entered as a value for filename, *acmeID* for example, is the value used by the ResolveMapping method. This method must be overridden. For every mapping registered, ResolveMapping is called. A string containing *acmeID* is sent; therefore requiring to get the appropriate value. The DocumentMapping class then automatically maps that value to the filename.

Example Title

```
public class MyMapping : DocumentMapping
{
    protected override string ResolveMapping(string p_Mapping,
                                             Document p_Document,
                                             out byte[] p_Binary)
    {
        return p_Document.GetMetaDataValue(p_Mapping);
    }
}
private void SetupMapping(Document p_Doc)
{
    MyMapping map = new MyMapping();
    // Add a new mapping to match.
    map["filename"] = "acmeID";
    // Match fields and metadata.
    map.ApplyMappings(p_Doc);
}
```

How to Implement Refresh Document/Folder

The Refresh Document/Folder feature is an optional feature that connectors can implement. When this feature is available, it is possible to refresh a single document or folder directly from the Administration Tool, which can be very useful if only one document needs to be updated in a very big source that takes hours to refresh.

To use this feature, implement the [IRefreshDocument and IRefreshFolder Interfaces](#).

How to Implement Delete Document/Folder

The Delete Document/Folder feature is an optional feature that connectors can implement. When this feature is available, it is possible to delete a single document or folder directly from the Administration Tool, which can be very useful if a document has been indexed and needs to be removed without having to remove the entire source.

To use this feature, implement the [IDeleteDocument and IDeleteFolder Interfaces](#).

What is Automatic Deletion of Removed Documents

The Automatic Deletion of Removed Documents is a feature of the CustomCrawler class that is implemented to keep the index as small and clean as possible. This feature automatically removes documents belonging to a source in the index that was not crawled by the latest [Refresh or Rebuild](#) operation. For example, the first crawling run of the connector finds the documents A, B and C: these documents are added in the index. The second crawling run, for some reason, only finds the documents A and C; hence, B is automatically removed from the index.

Usually, this feature is helpful; however, there are situations where, for example, document B was only unavailable temporarily and should not be deleted. If document B is actually a folder containing a million items; this means all these items have been removed.

This is why the connector must take into account this behavior and work accordingly. [Handling exceptions](#) correctly is the first step to making sure the crawling does not end prematurely. There is also the ReportUriToIgnoreForDeletion method that tags URIs to ignore. The URI that is given to this method and its children will not be removed.

How to Manually Remove Documents

It is possible to manually remove documents from the index by using certain methods available in the Open Connector API. If the remote system gives information concerning the removed documents, it is more efficient and logical to remove them right away than wait for the [automatic deletion of removed documents](#). Removing documents manually is particularly useful during Live Indexing.

Three methods are inherited from the CustomCrawler class in order to remove documents:

- **RemoveDocuments:** This method takes the unique URI of a document and removes it from the index if it is from the current source and it is in the index. A message is logged concerning the document being deleted.
- **RemoveDocumentAndChildren:** This method takes the unique URI of a document or folder and removes it as well as everything it has under it if it is in the current source. A message is logged concerning the documents being deleted.
- **RemoveOlderThan:** This method removes all the documents in the current source that are older than the specified DateTime. A message is logged concerning the documents being deleted.

Interfaces

This section describes in details the different interfaces available for the connectors as well as the way to implement them:

ISaveRestoreState Interface

The ISaveRestoreState interface allows administrators to pause and resume sources while they are crawling through the source interface. Implement this programming interface in order to perform these operations on sources from your connector.

When you implement the ISaveRestoreState interface, implement the SaveState, RestoreState and ResetState methods as well:

- **SaveState:** This method is called before the source is manually paused or stopped by another process of CES; therefore, the connector must be able to save its state at any time. This method must return an object that represents the current state of the connector. The object returned must be serializable and will be returned by the RestoreState method.
- **RestoreState:** This method is called before the source is resumed either manually or by another process of CES. This method takes as an input parameter an object that represents the state of the source when stopped. It is the same object returned by the SaveState method last time it was called.
- **ResetState:** This method is called if a problem occurs while restoring the state in order to ensure the connector is not in an unusable state. It is also called when a Rebuild or Refresh operation completes successfully. It is important to clear the state of the connector when this method is called.

ISaveRestoreState

```
[Serializable]
public class StateInfo
{
    // The current starting address.
    public string m_StartingAddress;
    // Any other information necessary to reposition should be kept here.
}

private StateInfo m_SaveInfo = null;
object ISaveRestoreState.SaveState()
{
    return m_SaveInfo;
}
//*****
/// <inheritDoc/>
//*****
void ISaveRestoreState.RestoreState(object p_State)
{
    m_SaveInfo = (StateInfo)p_State;
}
//*****
/// <inheritDoc/>
//*****
void ISaveRestoreState.ResetState()
{
    m_SaveInfo = null;
}
```

ILiveIndexing Interface

The ILiveIndexing interface allows administrators to enable a feature called *Live Indexing on Source* from the source interface. Live indexing implies that the source is periodically scanned for new, modified or deleted data; therefore, the information CES contains is up to date. Implement this interface in order to enable this feature on the sources of the connector.

When implementing the ILiveIndexing interface, it is also necessary to implement the RunLiveIndexing method. This method is called at regular intervals to retrieve modifications, additions or deletions that have occurred in the source since the last call of the method. This method is similar to Run, but only new, deleted or modified documents are notified to CES.

The RunLiveIndexing method executes in a separate thread; therefore, the time required to execute a refresh is not important. The **CustomCrawler base class** uses a Mutex to avoid calling the RunLiveIndexing method while a Run is being called. Hence, it is not required to handle this behavior.

RunLiveIndexing

```
void ILiveIndexing.RunLiveIndexing(DateTime p_LastRefresh)
{
    // Crawl the source.
    StartCrawling(p_LastRefresh);
}
```

IDeleteDocument and IDeleteFolder Interfaces

The IDeleteDocument and IDeleteFolder interfaces allow administrators to delete documents or folders from the Index Browser interface. Implement one or both of these programming interfaces in order to perform these operations on documents from the connector created.

- **IDeleteDocument.DeleteDocument:** This method receives a URI in parameter. This URI has to be removed by the connector.

DeleteDocument

```
bool IDeleteDocument.DeleteDocument(string p_Uri)
{
    // Remove the specified Document.
    RemoveDocument(p_Uri);

    return true;
}
```

- **IDeleteFolder.DeleteFolder:** This method receives a URI in parameter. This URI and its children have to be removed by the connector.

DeleteFolder

```
bool IDeleteFolder.DeleteFolder(string p_Uri)
{
    // Remove the specified Document and its children.
    RemoveDocumentAndChildren(p_Uri);
    return true;
}
```

IRefreshDocument and IRefreshFolder Interfaces

The IRefreshDocument and IRefreshFolder interfaces allow administrators to refresh documents or folders from the Index Browser interface. Implement one or both of these programming interfaces in order to perform these operations on documents from the connector created.

- **IRefreshDocument.RefreshDocument:** This method takes a URI and a Boolean in parameter. The Boolean represents the Rebuild/Refresh nature of the operation, just like the ForceRefresh property of the CustomCrawler class. The method must refresh or rebuild the specified URI.

RefreshDocument

```
bool IRefreshDocument.RefreshDocument(string p_Uri, bool p_ForceRefresh)
{
    // Crawl the specific document.
    StartCrawling(p_Uri, p_ForceRefresh)
}
```

- **IRefreshFolder.RefreshFolder:** This method takes a URI and a Boolean in parameter. The Boolean represents the Rebuild/Refresh nature of the operation, just like the ForceRefresh property of the CustomCrawler class. The method must refresh or rebuild the specified URI as well as its children.

RefreshFolder

```
bool IRefreshFolder.RefreshFolder(string p_Uri, bool p_ForceRefresh)
{
    // Crawl the specific folder.
    StartCrawling(p_Uri, p_FroceRefresh)
}
```

IBeautifulURI Interface

The IBeautifulURI interface allows the connector to validate and expand the URIs provided in the Administration Tool before they are passed to the connector. This method is always called on a new instance of the connector class. In order to avoid problems, it is suggested not to implement this interface unless it is explicitly required. Also, the validation is usually performed at the beginning of a crawling run.

BeautifulUri

```
string IBeautifulUri.BeautifyUri(string p_Uri,
    bool p_Validate,
    out bool p_IsValid)
{
    // Check that uri is valid.
    if (IsValid(p_Uri)) {
        p_IsValid = true;
    } else {
        p_IsValid = false;
    }
    // Modify the Uri.
    ModifyUri(p_Uri);
    // Return the modified Uri.
    return p_Uri;
}
```

IHasPersistentData Interface

The IHasPersistentData interface allows to persist data according to the CES system of *Precommit*, *Commit* and *Rollback*. If correctly implemented, it is extremely secure and guarantees the data it stores is always in the corresponding state of the connector. However, it is complex to implement and requires a certain amount of knowledge of threading, monitors, locks, etc. These methods are called as part of a few different scenarios. Here are the two most common ones:

- LockData > PreCommitData > CommitData > UnlockData
- LockData > RollbackData > UnlockData

To fully understand these operations, it is important to have a good knowledge of *Two Phase Commits*. Here are the methods that must be implemented in this interface:

- **ClosePersistenDataFile:** Called when the objects kept by the connector can be disposed and closed.
- **LockData:** Called at the beginning of an operation. It is important to have control of the persistent data (lock it) to make sure no other threads, processes, etc. modify it during the following operations.
- **UnlockData:** Called at the end to signal that the operations have ended and the lock over the objects can be released.
- **PreCommitData:** Called when the system wants to synchronize all of its parts in order to be ready to commit them. It is important to be able to roll back to the last valid committed state even when this call is over.
- **CommitData:** Called to commit the latest modifications that have occurred on the system. Once *Commit* has started, it cannot be stopped.
- **RollbackData:** Called to put the system back in its last known state after a failure occurs during a *Precommit*.

ISaveRestorePersistentData Interface

The ISaveRestorePersistentData interface is useful when the connector must save data that has to be persisted. It only has two functions: SaveData and LoadData. For simple operations, this interface is less complicated to implement and use than the IHasPersistentData interface. On the other hand, it does not provide the same level of valid security on the data, as it does not implement the *PreCommit*, *Commit* and *Rollback* systems CES uses.

- **SaveData:** Called to return an object that is persisted on a file. For example, the object can survive shutdowns. The object needs to be serializable.
- **LoadData:** Called to receive the object that is stored using SaveData in parameter.

ISaveRestorePersistentData

```
[Serializable]
public class Info
{
    // A string we want to save.
    public string m_StartingAddress;
    // Any other information to save.
}

private Info m_SaveInfo = null;

object ISaveRestorePersistentData.SaveData()
{
    return m_SaveInfo;
}
//*****
/// <inheritDoc/>
//*****
void ISaveRestorePersistentData.RestoreData(object p_Data)
{
    m_SaveInfo = (Info)p_Data;
}
```

General Knowledge

This section covers various topics useful in order to understand the Open Connector API, connectors and important features of CES:

What is a Connector



A connector is what CES uses to retrieve the content from various systems. Each system requires a specific connector in order to crawl the documents as the methods available to connect to it are unique. The connectors created using the Open Connector APIs are [dlls](#) that can be [added to CES](#) through the Administration Tool.

CES connectors can support various features to make them more efficient and easier to use:


- [Pause/Resume](#)
- [Live Indexing](#)
- [Refresh Document/Folder](#)
- [Delete Document/Folder](#)

How to Add an Additional Connector


Once a connector is ready to be used, it must be manually added to the list of connectors available in CES. To add an additional connector:

1. Access the **Additional Connector** page of the Administration Tool by clicking **Additional Connector** in the left navigation pane of the **Web Connector** page (Configuration > Connectors).
2. Click  **Add**. The **Modify Additional Connector** page is displayed.
3. Enter the appropriate parameters. For more information, refer to the following table.
4. Click  **Save**.

← Connectors - Additiona...

Name	<input type="text" value="MyConnector"/> ?
Description	<input type="text" value="This is an example"/> ?
Assembly Path	<input type="text" value="c:\Crawlers\Coveo.CES.CustomCrawlers.Custom1.dll"/> ?
Type Name	<input type="text" value="Coveo.CES.CustomCrawlers.ODBC.ODBCCrawler"/> ?
Parameters	 Add Parameter ?
Option	<input checked="" type="checkbox"/> Allow custom parameters ?
Available Threads	<input type="text"/> ?
Used By	

→ Save ✕ Cancel

Section	Description
Name	Identifies the additional connector. This value is displayed in the Source Type dropdown list of the Add Source page.
Description	Describes the repositories indexed by the connector.
Assembly Path	Indicates the full path of the connector's assembly file (e.g. <code>C:\CES5\Index\Crawlers\Coveo.CES.CustomCrawlers.Custom1.dll</code>). Note: If the path is incomplete, CES searches for the assembly file in its directory bin (by default, <code>C:\Program Files\Coveo Enterprise Search 5\Bin</code>).
Type Name	Indicates the name of the class implementing the connector (e.g. <code>Coveo.CES.CustomCrawlers.ODBC.ODBCCrawler</code>).
Parameters	Allows to define explicit parameters. Click  Add Parameter to display the Modify the parameters of the additional connector page.
Options	Allows the creation of dynamic parameters.

Why Use a ClassLibrary

The Open Connector API works with *dlls* because it is practical. A *dll* is very modular; therefore it is easy to consider them plug-ins for the Open Connector API. This makes it easy to update a connector by overwriting the *dll* with a newer version, transport it and provide support for the API when necessary.

Why Retrieve Metadata

Retrieving metadata is optional for a connector to work. On the other hand, to maximize its usefulness and make it easier to search the crawled documents, it is essential to retrieve metadata.

To customize the interface in order to add new fields and facets, the documents must have metadata; otherwise there are no values in the fields. Hence, the documents are not displayed when selecting a facet as there is no data to differentiate them.

User Identities in CES

User identities in CES are used to store sets of credentials. These credentials can be necessary for multiple sources or security providers. Rather than re-entering them every time, it is possible to create a new user identity with the credentials and then select that identity when necessary.

The Open Connector API supports [user identities](#).

Samples

This section contains sample code written using the Open Connector API. These samples can be compiled and tested anywhere; moreover they help demonstrate how to create a new connector.

Empty Project

This sample is intended to allow users to test a new connector. Hence, the *Visual Studio* configuration steps can be skipped and it is possible to directly develop the connector.

To use the template project, double-click the *TemplateConnectorProject.csproj* file on a computer where *Visual Studio 2005* or *2008* is installed. This creates a new solution and opens the project.

Note: In order to compile, CES5 must have been installed before loading the project in *Visual Studio*.

The project is a ClassLibrary type, which automatically outputs a *dll*. The references are already set; therefore it finds the necessary *dlls* where they have been saved by the installation kit of CES. The Run method is already overridden—but empty—and the class is marked as a connector; meaning the project compiles and creates a valid connector *dll*, but does nothing. By using the Template Project, it is possible to skip to the [How to Retrieve Starting Addresses](#) step.

[Download the Empty project](#)

Gmail Pop3 Connector

The Gmail Pop3 connector is a simple working example of a connector. It is intended to help users learn how to work with the Open Connector API. The Gmail Pop3 connector allows you to crawl the content of a Gmail account using the Pop3 feature available on it and retrieve emails, which can then be searched using CES. To do so, execute the following procedure.

Note: To create a Gmail account, access www.gmail.com.

1. In order for it to work properly, [enable the Pop3 feature](#) on the Gmail account.
2. Compile to project to create the connector *dll*: *Coveo.CES.CustomCrawlers.GmailPop3Connector.dll*.
3. [Add the connector to CES](#) and create a new source: the starting address is pop.gmail.com, which is the name of the Gmail pop server.
4. Create an identity representing the Gmail account using the full Gmail address as username (for example, [acme@gmail.com](#)) as well as the appropriate password.

When rebuilding or refreshing, the connector uses the Gmail pop server feature called recent mode. This feature retrieves all the emails received in the last 30 days; however Live Indexing, on the other hand, retrieves all the new mails. The Gmail pop server is configured to send the same email only once; hence, it is not sent every time you connect to the server, meaning it is impossible to rebuild the source without re-enabling pop directly on the Gmail account.

[Download the Gmail Pop3 Connector project](#)