CA DataMinder

Content Provider Development Guide

Release 14.6

CA Technologies Product References

This document references the following CA Technologies products:

- CA DataMinder™
- CA Directory
- CA SiteMinder®

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

Contents

Chapter 1: Summary

This guide provides a general description of how to develop a **Content Provider** for the CA DataMinder Content Services.

By default, CA DataMinder uses Autonomy IDOL as the database system for its Content Classification Service. CA DataMinder comes with a Content Provider that interfaces with the Autonomy IDOL database. You can develop a custom content provider that interfaces with your own database system.

The intended audience is the developer or team tasked with the implementation of a Content Provider. The reader must be familiar with CA DataMinder's terminology and general architecture. The reader must be fluent with the Microsoft Component Object Model (COM).

Read this document together with the Content Provider interface definitions file (ICISContentProvider.idl) and the Content Provider XML example and documentation (CISContentProvider.xml).

The introduction gives an overview of Content Providers and includes a summary of the required main characteristics of a Content Provider. The guide then provides a detailed look at the implementation of the entry point of a Content Provider, and its indexing and searching functions.

Note: The CA DataMinder product documentation and log messages refer to Content Providers as '**Content Connectors**'. The term Content Provider is used in the code and in all the APIs, and is therefore used in this guide.

Chapter 2: About Content Provider Development

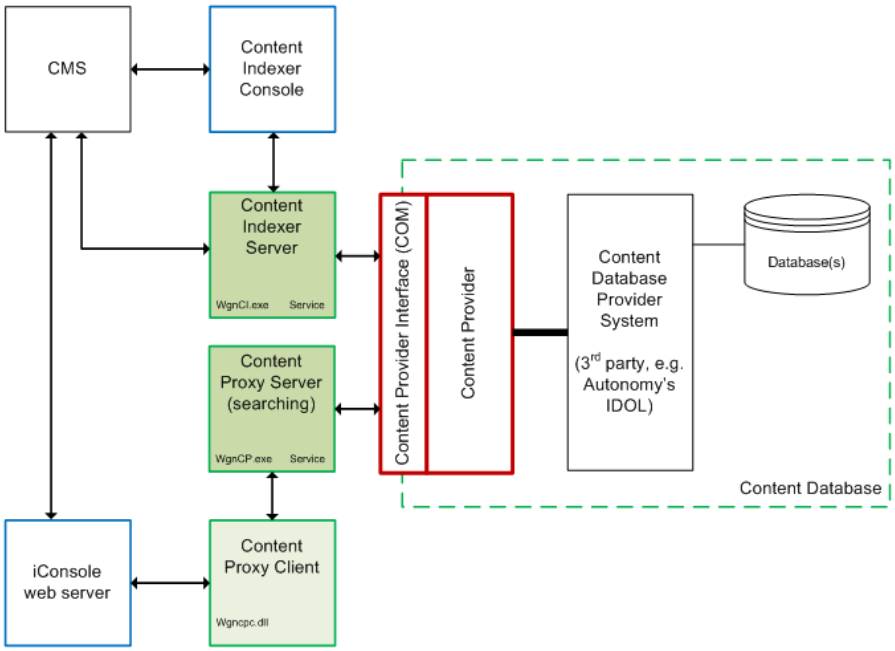
The CA DataMinder Content Services consist of three components (outlined in green in the diagram):

- Content Indexer,
- Content Proxy,
- Content Database

The Content Indexer runs configurable jobs which pull events and their data from a CMS database. Then the Content Indexer pushes the events to the Content Database for indexing. The Content Proxy turns content search requests into queries for the Content Database, and returns search results.

The Content Database stores the indexed content data. The Content Database consists of a third-party database system which interfaces with the rest of the Content Services through a **Content Provider**.

Content Providers form an abstraction layer for the rest of the Content Services. A Content Provider provides an abstraction for a specific database system, and contains all database-specific code. The Content Indexer and Content Proxy are database-agnostic.



- (Green) Content Services components provide the indexing and searching services.
- (Blue) Current consumers of the services:

  The Content Indexer Console (for indexing) connects to the Content Indexer; the iConsole web server (for searching) connects to the Content Proxy. In the Content Indexer's case, connection is direct, through COM, to the Content Indexer Server. In the Content Proxy's case, connection is through COM to the Content Proxy Client, which communicates proprietarily with the Content Proxy Server. The Content Indexer Server also communicates directly with the CMS from which it draws the events to index.

- (Red) The Content Database provides indexing and searching capabilities to the Content Indexer and the Content Proxy. The Content Provider provides the interfacing layer (using COM) for the Content Database to the Content Indexer and the Content Proxy.

Content Provider Characteristics

COM Interfacing

The Content Provider public interfaces are COM interfaces.

- A Content Provider must be a COM server.
- A Content Provider must implement one public coclass with a CLSID assigned by the implementer; the coclass exposes the ICISContentProvider interface with the following IID:
  `{4daaab52-d5c5-11d4-b613-000102027bbb}`.
- The coclass must be multithreaded.
- An implementer can develop the COM server to run as a separate process, or to be an in-process server using a DLL. If the latter, the DLL must be 32-bit, since the Content Indexer and Content Proxy are currently 32-bit processes; also, the threading model must be 'Both' so the server runs in the apartment of the client.
- All other COM interfaces (and whichever COM identities support them) that are exposed by a Content Provider are accessed solely using methods on the ICISContentProvider interface.

Clients and Sessions

A CA DataMinder component (Content Indexer, Content Proxy, or any other) that connects to a Content Provider is called a **client**.

When a client creates an object instance of the public coclass, this instance forms a **session** with the Content Provider. A client may potentially have multiple sessions with a Content Provider. Each session is independent.

Functions

A Content Provider can provide up to two **functions**: an "Indexer" function and a "Query" function.

- A Content Provider may implement only one of the function (for segregation of functionality across multiple hosts), but will typically implement both functions.
- For each function that it supports, a Content Provider must be able to provide logical instances of it to clients. When a client requests any function instance through a session, the Content Provider creates a logical instance of the function. The Content Provider returns to the client a function-specific COM interface pointer to access the instance. You can therefore have **indexer instances** and **query instances**.

- How the COM identity that is underlying an instance is implemented is at the discretion of the Content Provider, but logical segregation must exist. That means, from the point of view of a client session, each function instance is independent, in configuration and in execution, from any other instance and any other function. An exception would be an irrecoverable problem that occurs in a function instance that requires the whole Content Provider to shut down; in this case, all other function instances are obviously affected. Just like for the public coclass, the implementations of the functions must be multithreaded.

Use of XML

The Content Provider and its clients use XML to pass information back and forth.

- Encoding is restricted and constrained to be little-endian UTF-16.
- A Content Provider is responsible for providing XML that is little-endian UTF-16, and that is guaranteed to be valid for the XML schema version it declares to support.

Own Configuration

A Content Provider needs to manage two sets of configuration parameters for itself.

- External database-specific configuration parameters control how the Content Provider interfaces with the third-party database system. For example, this can be a communication port number.
- Internal configuration parameters control how the Content Provider manages itself. For example, this can be a timeout value to report a document indexing operation as failed if the third-party database system has not responded in time.

These configuration parameters are the Content Provider's responsibility and are entirely under its control. Where and how these parameters are located and set up is up to each Content Provider. We recommend the use of the registry. Clients do not have access to these parameters.

Note: CA Technologies has provisionally defined an ICISConfigure interface together with an example of XML structure. The aim is to pass configuration information from and to a Content Provider (including a Content Provider's own configuration, for instance to allow GUI-based configuration management). This interface is currently reserved. Do not implement it.

Own logging

- A Content Provider is free to do its own logging (which is recommended).
- A Content Provider does not do any logging for its clients. Each client does its own logging, and depends on status reports from a Content Provider to do so.

Status Reports

Content Providers use Status reports to report on the state of a function instance, on the progress of an operation (including its start) and its outcome (completion), or on the result of an action by a client (feedback). Status reports are also used to return search results and output from GetStatus() calls.

Apart from returning search results etc, there are two reasons why Content Providers generate status reports:

a. A client depends on status reports to log the state of its functions in use, and to log the progress of its individual operations. This logging is crucial to keep the customer informed of what is happening, and also for diagnostic purposes.

b. A client depends on status reports to control its execution, for example, to keep track of the indexing of documents.

Status Callback Mechanism

- All status reports are made through ICISStatusCallback callback interface pointers supplied by a client. The only exceptions are status reports that are returned directly as an output of an interface method call, for example, ICISContentIndexer::GetStatus().
- A client must pass valid status callback interface pointers to a Content Provider.
- A Content Provider must use the appropriate status callback interface pointers to send status reports, and it must do so on time.
- A Content Provider may aggregate multiple status reports to send through the same status callback interface pointer. It may report them all with one call to ICISStatusCallback::ReportStatus() rather than with multiple individual calls.

Status codes

Status codes are of data type HRESULT.

- Content Providers should only return either standard status codes (for example, E_OUTOFMEMORY), or common status codes which are defined in ICISContentProvider.idl for use by all Content Provider implementations. This applies to HRESULTs returned from interface method calls and also to HRESULTs reported in status reports.
- A Content Provider should not return private HRESULTs. Private status codes are unknown to clients and they cannot interpret them. Also they cannot be resolved into messages that can be logged by clients.
- A Content Provider can ‘attach’ custom messages to common or standard HRESULTs in status reports. For example, database-specific error messages. See “Custom message information” in CISContentProvider.xml.
- Clients log the custom message information (message text, code, severity) verbatim.

- Content Providers are encouraged to provide custom messages because such messages provide valuable information for diagnostic purposes.

  Note: The current list of status codes was defined to support CA DataMinder's first Content Provider implementation. It is generic and limited in scope. Content Provider implementers are welcome to suggest new common status codes that they feel are valuable to return.

Specifying Which Content Provider to Use

By default, the CA DataMinder Content Services use the coclass with the CLSID {521fdd42-d5c5-11d4-b613-000102027bbb}. By extension this identifies the Content Provider that is used.

You can override the default by setting the CLSID of the core coclass of the desired Content Provider in the following registry value:

On 32-bit Windows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\ComputerAssociates\CA DataMinder
    \CurrentVersion\Content Services
CustomProviderCLSID REG_SZ
```

On 64-bit Windows (pointing to the 32-bit registry):

```
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\ComputerAssociates\CA DataMinder
    \CurrentVersion\Content Services
CustomProviderCLSID REG_SZ
```

Chapter 3: Implementing the Content Provider CoClass

An instance of the Content Provider public coclass forms a session with a client.

The Content Provider coclass does very little. Its ICISContentProvider interface has three methods, GetInformation(), CreateIndexerInstance(), and CreateQueryInstance().

ICISContentProvider Interface Methods

GetInformation()

This method returns to a client an XML document entity which contains information about the Content Provider.

- A client uses this information for logging purposes, but also to verify that a function that it intends to use is supported (refer to the <Functions> element), and has the required characteristics (such as synchronous indexing).
- A client is responsible for verifying the XML schema version reported as supported by the Content Provider. A client which continues, and uses the Content Provider functions, is implicitly guaranteeing the Content Provider that it passes it XML according to that schema version.

CreateIndexerInstance()

The Content Provider returns to the client a pointer to an ICISContentIndexer interface. As this is COM, the interface pointer must have been AddRef()ed before being given out; it is the client's responsibility to Release() it.

There is no restriction on the COM identity of the implementation behind the ICISContentIndexer interface, but the Content Provider must maintain logical separation of each "instance" thus returned, from other indexer instances, but also from query instances, and from other sessions.

CreateQueryInstance()

The Content Provider returns to the client a pointer to an ICISContentQuery interface.

There is no restriction on the COM identity of the implementation behind the ICISContentQuery interface, but the Content Provider must maintain logical separation of each "instance" thus returned from other query instances, but also from indexer instances, and from other sessions.

Sequence of Operation from a Client (ICISContentProvider)

1. A client cocreates an instance of the Content Provider coclass, asking for the ICISContentProvider interface.
2. The client calls GetInformation(), and if successful, verifies the XML schema version indicated in the XML returned, and the functions supported by the Content Provider, and their characteristics.
3. If the client is satisfied, it calls CreateIndexerInstance() or CreateQueryInstance().
4. Depending on what the client does, further calls to CreateIndexerInstance() or CreateQueryInstance() or both, may be made. A client may also create new sessions and bypass calling GetInformation() on these, since it would already have done the verification previously.

Note: A client can be a discovery agent or be solely interested in the Content Provider's information, and not need access to indexing or searching.

Chapter 4: Implementing the Indexer Function

An Indexer [instance] is a state machine. It has three sets of states: pre-active, active, and post-active.

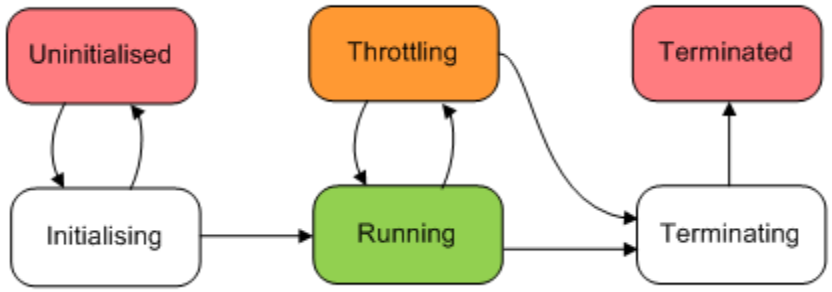
- The pre-active states are ”uninitialised” (as spelled here), and the intermediate state ”initialising” (as spelled here).
- The active states are “running” and “throttling”.
- The post-active states are the intermediate state “terminating” and “terminated”.

Indexer Instance

State Transitions Diagram

A terminated indexer cannot be re-started, and that a terminating indexer cannot be stopped terminating.

While active, an Indexer may also throttle, if supported by the Content Provider. Throttling is a temporary state from which The Indexer either recovers (back to “running”), or which the Indexer leaves by terminating.



Initializing

Initialization is implemented by the ICISContentIndexer::Init() method. The method supplies the ICISStatusCallback interface pointer that the Indexer uses. See ICISContentProvider.idl for more behavior details.

Terminating

A client can actively terminate an Indexer by calling either ICISContentIndexer::Abort() or Terminate(), or by the client releasing the last reference to the ICISContentIndexer interface.

Alternatively, the Content Provider can terminate the Indexer. Such a self-termination may occur for many reasons. For example, if the Content Provider cannot recover from an error situation; if the content database is in a bad state that cannot be recovered from; or if “throttling” is going on for too long and has hit a timeout limit.

Throttling

A Content Provider must be in control of its content database system. This includes monitoring its health, managing its connections, managing the data that is pushed to it.

- Throttling is a state controlled by a Content Provider [Indexer]. Throttling is triggered when the Content Provider detects that the content database is not keeping up with document indexing.
- When a Content Provider [Indexer] decides to throttle, it delays the execution of the ICISContentIndexer::Index()calls. The only effect from a client’s point of view is that calls to Index() take longer to return, both in asynchronous and synchronous behaviors. The length of the delay for each call can be fixed, variable, or fully dynamic; that is entirely up to the Content Provider and the sophistication of its throttling control and management.
- The throttling continues while the Content Provider takes remedial action. For example, this may involve doing nothing but waiting for queues to shrink.
- Upon entering the throttling state, the Indexer must send a status report. This may make a client pause calling ICISContentIndexer::Index(), independently of the delaying mechanism.
- The Indexer should send status reports at regular intervals if throttling continues.
- Finally the Indexer must send a status report when it recovers. If recovery is not possible, it must start to terminate.

Indexing

A client queues a document for indexing by calling ICISContentIndexer::Index().Index() queues the document and returns CIS_S_PENDING.

By default, indexing is an asynchronous operation, with Index() returning when the document is queued or if the queuing fails.

If queued, the document is indexed in the background. When the document indexing is complete, the Indexer sends a status report through the callback interface. See also Asynchronous Behavior below.

a. Queuing a document should succeed, and Index() should return CIS_S_PENDING.

b. A Content Provider should fail to queue a document only if the Indexer is not "active" or if the parameters of the Index() call are structurally invalid (for example, an invalid NULL pointer). Apart from these conditions, the queuing of a document can only fail due to exceptional circumstances, such as failing to allocate memory or failing to notify the client of the queuing.

   Note: Notifying the client of the queuing by sending the "queued" status report through the callback interface is an integral part of the queuing process. If calling ICISStatusCallback::ReportStatus() fails, then the Indexer should fail the queuing. This way, it is not possible to have a situation where the Indexer has a document queued, but the client has not received the "queued" status report for it.

c. If Index() fails to queue the document, it returns an error code. No status report is generated for the document.

d. If Index() succeeds in queuing the document, it returns CIS_I_PENDING. The Indexer sends a "queued" status report for the document through the callback interface, typically before Index() returns. When the document's indexing completes, the Indexer sends a "completed" status report for the document, regardless of whether the indexing was successful. The completed report can contain a failure code.

This mechanism helps ensure that besides incorrect API usage (an issue at the client) and a minimum of unexpected failures (for example, running out of memory), the start and end of the document indexing process are always logged. In other words, the acceptance of a document for indexing is the expected behavior, even if the indexing fails almost immediately because of trivia such as an invalid parameter value.

Note: Report any failure reason (such as invalid XML syntax, invalid configuration parameter, connection failure to the content database service, inconsistent or invalid document description, and others) through the "completed" status report. This approach maximizes information provided through logging.

The Index() call supplies four separate pieces of data: the document to index (i.e. its content), its description, its metadata, and some user reference data.

The user reference data is data that the Indexer must pass back in status reports about the document. That data is for the sole use of the client. It cannot be interpreted by a Content Provider.

A client may pass the document content directly, using an IStream interface, or through a file, by indicating contentInFile="true" in the document description.

- Passing document content using files is an optional feature offered by a Content Provider. A Content Provider may support it, but it is not mandatory.
- If the Content Provider supports it, this capability must be indicated in the Indexer function information element (see the XML schema), which alerts the client that the capability is available.
- If a client erroneously calls Index() with contentInFile="true", and the Content Provider does not support the capability, then the Indexer should report that the document's indexing failed (for example, with CIS_E_NOT_SUPPORTED). As per the rules, this means that the Index() call succeeds with CIS_S_PENDING, the document is reported as "queued", and then reported as "completed" with a failure code.

An Indexer may delay the execution of the method if it is currently throttling.

See ICISContentProvider.idl for more behavior details.

Asynchronous Behavior

A Content Provider must support indexing document asynchronously. This is the default behavior.

This means that calls to ICISContentIndexer::Index() return quickly (unless the Indexer is throttling) with CIS_S_PENDING.

A Content Provider can also support synchronous document indexing. If so, this capability must be indicated in the Indexer function information element (see the XML schema).

The ICISContentIndexer::Init() method implementation must verify if a client is wanting to configure the Indexer for synchronous document indexing, and fail if appropriate (for example, with CIS_E_NOT_SUPPORTED).

See ICISContentProvider.idl for more detail about the expected behavior of synchronous indexing and Index() in particular. Put simply, a synchronous version of Index() wraps up the asynchronous behavior implementation and synchronizes the "queued" and "completed" status reports before returning.

Note: The CA DataMinder 14.0 and 14.1 Content Indexer Server (client) does not currently use synchronous indexing. This may change in future releases.

Outstanding Indexing Operations on Termination

If an Indexer starts to terminate through a client request or through self-termination, it goes into the “terminating” state. Two things must happen before the Indexer moves on to the “terminated” state:

a. Any Index() call that was made before termination started and is currently in progress, but with its document not yet queued, must fail with CIS_E_SHUTDOWN.

b. Any document queued, but not completed, is abandoned immediately if the termination is traumatic, for example, an Abort() call. If the termination is normal, for example, a Terminate() call, then the document is allowed to complete or to time out.

Any call to Index() after termination starts must fail with CIS_E_SHUTDOWN.

Status Reports

The Indexer sends status reports by calling the ReportStatus() method on the ICISStatusCallback interface pointer that it received when the client called ICISContentIndexer::Init().

The Indexer must send status reports when:

- Initialization starts.
- Initialization completes (independent of whether it was successful).
- Throttling starts.
- Throttling ends and normal running resumes.
- Termination starts.
- Termination completes.
- The client asks for a flushing operation (see ICISContentIndexer::Flush()).
- A document is queued for indexing.
- A document’s indexing completes (independent of whether it was successful).

The Indexer may send optional status reports when, for example:

- The indexer does not timely move on from a temporary state (initializing, throttling, terminating).
- A document’s indexing is taking too long.
- Periodically, for instance to report the current value of monitored counters.

Multithreading

The implementation of the ICISContentIndexer interface must be multithreaded.

A client may call methods on an ICISContentIndexer interface pointer from multiple threads (for example, worker threads).

Sequence of Operation from a Client (Indexer)

1. The client first verifies the Content Provider's capabilities regarding indexing. For more information see Sequence of Operation from a Client (see page 16).
2. The client (typically the Content Indexer service) uses its ICISContentProvider interface pointer and calls CreateIndexerInstance() to get a ICISContentIndexer interface pointer.
3. The client initializes the Indexer instance by calling ICISContentIndexer::Init().
4. Upon a successful initialization of the Indexer instance, the client cycles through the documents it wants to index using that Indexer, calling ICISContentIndexer::Index() for each of them.
5. If a client is batching documents, it may sometimes call ICISContentIndexer::Flush() and possibly pause while it waits to be told of the completion of all the pending documents at that point.
6. After having sent all its documents to be indexed, the client calls ICISContentProvider::Terminate() or releases the ICISContentIndexer interface pointer. The client also waits to be told of the completion of all the documents still pending.

   Note: Calling Terminate() may occur after waiting for the documents to complete; that is entirely up to the client.

If the Content Indexer console operator stops an indexing job, the client that is managing that job stops sending documents. The process effectively arrives at step 6 early. In exceptional circumstances, the client may stop sending documents, and call ICISContentIndexer::Abort().

Chapter 5: Implementing the Query Function

A Query instance is a handler of search queries. Unlike an Indexer instance, which is configured once using **ICISContentIndexer::Init()** and whose configuration applies to all documents it indexes, a Query instance has each search call take its own configuration parameters.

Query Instance

Searching

A client sends a query by calling ICISContentQuery::Search(). Search() executes the query and returns results.

Searching is a synchronous operation by default, with Search() returning only when it has search results, or if the query fails. See also Synchronous Behavior below.

a. The first thing that Search() does is try to "accept" the query. This is conceptually similar to the queuing of a document with Index(), and is signaled by the Query instance sending a "queued" status report to the client. To send the status report, it uses the ICISStatusCallback interface pointer given as a Search() call parameter.

b. A Content Provider should fail to accept a query only if the Search() call parameters are structurally invalid (for example, a NULL ICISStatusCallback interface pointer), or for other exceptional circumstances, such as failing to allocate memory, or failing to notify the client of the query's acceptance.

   Note: Notifying the client of the query's acceptance is an integral part of the process, in a similar fashion to the queuing mechanism for indexing. If calling ICISStatusCallback::ReportStatus() fails, then the Query instance should fail to accept the query. This way, it is not possible to have a situation where a query is accepted but the client has not yet received the "queued" status report for it.

c. If Search() fails to accept the query, it returns an error code. No status report is generated for the query.

d. If Search() succeeds in accepting the query, it sends a "queued" status report for the query. When the query completes, it sends a "completed" status report, independent of whether it was successful. It then returns, either with a success code and the results, or with a failure code (copied from the "completed" status report) and no results.

This mechanism helps ensure that besides incorrect API usage (an issue at the client) and a minimum of unexpected failures (for example, running out of memory), the start and end of a query is always logged. In other words, the acceptance of a query is the expected behavior, even if it fails almost immediately because of trivia such as an invalid parameter value. You should report any failure reason, such as invalid XML syntax, invalid configuration parameter, connection failure to the content database service, inconsistent or invalid search parameters, and others, through the "completed" status report. This approach maximizes information provided through logging.

The Search() call supplies two separate pieces of data: the query configuration parameters, and the query search parameters. It also supplies the ICISStatusCallback interface pointer that the Query instance uses to send status reports to the client about the query.

The Search() method returns search results (if applicable) in the ppSearchResultsXml output parameter. If the search is asynchronous, results (if applicable) are fed through status reports.

See ICISContentProvider.idl for more behavior details.

Canceling a Search

A client can cancel ongoing searches by calling ICISContentQuery::CancelSearch().

See ICISContentProvider.idl for more behavior details.

Synchronous Behavior

A Content Provider must support searching synchronously. This is the default behavior. Synchronous searching means that a call to ICISContentQuery::Search() returns only when it has search results, or if the query fails.

A Content Provider can also support asynchronous searching. If so, this capability must be indicated in the Query function information element (see the XML schema).

The ICISContentIndexer::Search() method implementation must verify if a client wants to run an asynchronous search, and fail if appropriate (for example, with CIS_E_NOT_SUPPORTED).

See ICISContentProvider.idl for more detail about the expected behavior of asynchronous searching and Search() in particular.

Note: The CA DataMinder 14.0 and 14.1 Content Indexer Server (client) does not currently use synchronous indexing. This may change in future releases.

Status Reports

The Query instance sends status reports for a query by calling the ReportStatus() method on the ICISStatusCallback interface pointer that the Query instance received when the client called ICISContentIndexer::Search().

The Query instance must send status reports when:

- The query is accepted.
- The query completes (independent of whether it was successful).

The Query instance may send optional status reports when, for example the query is taking too long.

Multithreading

The implementation of the ICISContentQuery interface must be multithreaded.

A client may call methods on an ICISContentQuery interface pointer from multiple threads.

Sequence of Operation from a Client (Query)

1. The client first verifies the Content Provider's capabilities regarding querying. For more information see Sequence of Operation from a Client (see page 16).
2. The client (typically the Content Proxy service) uses its ICISContentProvider interface pointer and calls CreateQueryInstance() to get a ICISContentQuery interface pointer.
3. The Content Proxy service manages Content Proxy Client connections. For each connection, the Content Proxy service typically creates one Query instance.
4. Every time the Content Proxy Client launches a search on that connection, the Content Proxy service calls ICISContentQuery::Search().
5. If the Proxy Content service needs to cancel a search, it calls ICISContentIndexer::CancelSearch().