

7

Designing and Writing Repository Connectors

This chapter covers

- Understanding the repository connector methods
- How to make repository connector design decisions
- Implementing a sample repository connector
- Debugging tips and techniques

In this chapter, you will learn how to design and write a ManifoldCF repository connector. You will learn about the choices you have to make in order for your connector to operate properly as part of the ManifoldCF framework. You'll be doing this in the context of an example written against a home-grown example content management system.

Writing a repository connector is actually pretty simple. Making it simple to write connectors was a design goal for ManifoldCF all along. In many ways the trickiest part comes not from the code itself, but from the decisions that need to be made before you start to write it. We'll cover that first, and then we'll move on to an example implementation. We'll wrap up by describing some real-world case studies, and introduce tools that might be helpful for debugging your connectors.

7.1 *Designing a repository connector*

Your friend Bert is beginning to get impatient. He's been stopping by Frank's office at regular intervals to check on his progress. Frank has been responding with superior smiles, but notably he's been going out of his way to make sure Bert never catches a glimpse of

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

anything that is displayed on his monitor. Nevertheless, Bert feels that time is running out, and that he'd better be ready to code a connector as quickly as possible.

Your response to Bert is that he actually **is** ready. All that is missing is the knowledge of what Frank's system actually does, and having that, he could get started. But Bert should be prepared to think through the design carefully before he starts coding anything.

We're going to start by going through these decisions one by one, in the context of an example content management system that was written just for this book. The system is called "Docs4U". It is based on a shared file system area, and is just barely powerful enough to have functionality typical of real-world proprietary content management systems, including security.

7.1.1 *Deciding on the meaning of the document identifier*

The first thing we will need to decide is what our connector should use as a document identifier. A document identifier is just a unique handle for an individual repository document. However, there are sometimes multiple choices for what to use to fulfill this role.

Luckily, you have wide latitude as to your choice. ManifoldCF treats document identifiers as unlimited-length, opaque strings, which have no intrinsic meaning outside of the connector which created them. All that ManifoldCF cares about is that a document identifier has the following characteristics:

- It must uniquely describe a single "document", and provide the context for additions, changes, and deletions
- It must easily be used to obtain a description of the document's version string, which will be compared to detect changes to the document
- It must easily be used to obtain content and security information associated with the document, if any
- If the document has any indexable content, it must easily be used to build a URL or IRI, which is intended to function as the document's unique ID within the search engine, and can uniquely access the document from within search results

That sounds easy enough. So, are we done now? Unfortunately, we aren't; there are potential complications we still need to consider.

MULTIPLE REVISIONS

One of the useful things many content management systems do is keep multiple revisions of an given document around. Each of these revisions can be fetched and examined. An outstanding question we have to ask is whether we should consider indexing multiple revisions of a document independently, and if so, how many? Some repositories, such as IBM's FileNet, take this even a step further, and use a revision identifier as the primary way you get to document content. In these kinds of systems, there isn't any such thing as a document identifier at all! How do we deal with this?

Ultimately, the decision as to how many revisions to index is up to you. As long as you can find a way to describe each revision uniquely in a manner that meets the specified

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

criteria, ManifoldCF does not care just how many knots you need to twist into to index multiple revisions. On the other hand, a user may well care; it's odd for a search engine to return more than one revision of any given document. In my experience, users traditionally expect their search engine to return only the most recent revision, and don't want any chance of returning older ones. This is indeed the way all of ManifoldCF's shipping connectors behave. In the case of the FileNet connector, only the most current revisions are considered to belong to a job's set of documents. Thus, when there is a new revision of a document, the new revision is discovered on seeding. Later, at the end of the job, the older revision is deleted, because it is no longer reachable.

DOCUMENTS WITH NO CONTENT

We've been talking about document identifiers quite a bit, but we've never actually gotten around to defining what a document is. That's unfortunate, because more subtleties lie in that direction. Let's do it now.

Definition A *document* is an entity related to a repository, which is either a unit of content uniquely addressable by a URL, or a source of document identifiers, or both.

In other words, a document may either contain content we want to index, or contain references to other documents, or both. But, wait a minute! Doesn't this mean you can have documents in ManifoldCF that have no content? How can that be?

This is best answered by thinking about certain specific connectors, such as the file system connector, or the RSS connector. In both cases, there are documents which are used only to find references to other documents. In the file system case, those documents correspond to directories. In the RSS connector case, they correspond to feeds. In either case, they serve the critical purpose of allowing ManifoldCF to locate the documents that **should** be indexed.

What usually determines whether a connector has documents with no content is whether all documents for any job you can think of can be located directly at seeding time. If all documents can be found up front, then there is no need for documents that only have references. But if the only way to get to a document is indirectly by means of a directory or folder structure, then the connector will need to have a way of representing those intermediate hops as document identifiers.

DOCS4U DOCUMENT IDENTIFIER

The API code for the Docs4U content store can be found at http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/docs4u/src/org/apache/manifoldcf/examples/docs4u/Docs4UAPI.java. A quick glance this API shows that we seem to need a Docs4U document ID to access document information and content:

```
public boolean getDocument(String docID, D4UDocInfo docInfo)
    throws D4UException;
```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

This is encouraging. What about the requirement to be able to construct a version string? Well, what better place to start than the documents update time? That can be found in the API in this method:

```
public Long getDocumentUpdatedTime(String docID)
    throws D4UException;
```

Finally, what about being able to construct a URL? But yes, as you might guess, there's a method for that too:

```
public String getDocumentURL(String docID)
    throws D4UException;
```

That's all we need! So it looks like we should tentatively try to use this Docs4U document ID as the ManifoldCF document identifier for our example.

Note The Docs4U example repository is actually going out of its way to make our lives easy here. In real life, the most difficult part of doing the homework of making sure that a document identifier will actually work as needed is the rule that you need to be able to construct a URL from it. This is because not all repositories natively provide HTTP access to their content. If this seems to be the case for a repository you are trying to work with, don't give up! It is usually straightforward to build such a web application on your own, using the same API you plan to access from within your connector, if web access is the only thing missing.

7.1.2 *Deciding on the form of an access token*

In addition to obtaining a document's content, our connector is going to be responsible for obtaining security information about each document. As part of the design process, we will need to figure out what an access token should look like for the connector.

This should be easy. In theory, all we have to do is look at the repository API and see what kinds of security information are available for any given document. But once again, there are subtleties. For instance, a repository may make a distinction between the privileges needed to view a document and the privileges needed to change it. There may also be special, implicit privileges, such as the implied ability of an owner of the document to see and edit the document.

What you will need to do to come to grips with a repository's security model is to first learn about it as much as is possible. Remember that your ultimate goal is to write code to convert the information you connector has available per repository document into some set of ManifoldCF access tokens, which grant or deny the ability to see the document's content. And, unless you think you can use an already existing authority connector, you might as well plan out how that's going to obtain the corresponding access tokens for a user at the same time.

ACTIVE DIRECTORY INTEGRATIONS

Many real-world repositories have both a native security model as well as an integration with Microsoft's Active Directory. This can make it hard to decide whether the connector's access tokens should be the native kind, or should be Active Directory SIDs. If your repository

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

software performs the integration by providing an integrated mapping from native security to Active Directory SIDs, you may find that you have a choice as to which to pick. How do you resolve this?

The important factor in this decision is how permanent the association is between native security and Active Directory security. If the relationship is permanent and cannot be altered, then it is acceptable to use SIDs as the access token format for your repository connector. This will save you work, because then you won't need to write a special authority connector for this repository. On the other hand, if the association is fluid and can be changed, then you have to ask yourself a number of questions in order to make the right decision.

- Do you need changes in the mapping to show up in real time, or is it okay if they show up only as a result of the running of a ManifoldCF job?
- How difficult it would be to include something in the each document's version string that describes this mapping, so that we can detect when the mapping it changes?

If you need any changes in association to appear immediately, or if it is very difficult or expensive to represent the mappings as part of a document's version string, then you are probably better off using native access tokens, and writing a special authority.

Docs4U ACCESS TOKENS

As you might expect, the Docs4U API is very simplistic in regards to security. The document access method, `getDocument()`, fills in a `D4UDocInfo` object with everything there is to know about the document. The `D4UDocInfo` object has the following security-related methods for accessing the document's security information:

```
public String[] getAllowed();  
public String[] getDisallowed();
```

These methods are used to get the sets of user or group identifiers that are either explicitly allowed or disallowed from seeing the document. Once again, the Javadoc doesn't help us figure out how these two sets of user and group identifiers are supposed to interact, so let me dive in again and tell you that Docs4U works similarly to Active Directory in this regard; the document is visible to everyone described by the "allowed" list, except if the user happens to also match a user or group on the "disallowed" list. Furthermore, if the document has no "allowed" list at all, the document is invisible to everyone.

The plan, then, regarding Docs4U access tokens and security will be for access tokens to be Docs4U user or group identifiers, and for there to be a Docs4U authority connector which maps the user to her corresponding list of user and group identifiers.

7.1.3 What should a document's URL be?

One of the requirements for a repository connector is that it must provide a "URL" for every indexed document. This is used as the document's primary key in the target index. It is also assumed that it will be provided by the search engine when the search results are displayed to provide a link to allow the searching user to examine the document's content.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

The “URL” does not technically need to be an actual URL. It must, however, be meaningful to a web browser. File IRI’s, e.g. `file:///someserver.com/dir1/dir2/filename.ext`, are also understood by most browsers, and can fill the role of a “URL” when the document exists somewhere in a shared file system.

Another necessary characteristic of the “URL” is that it must provide *secured* access to the document. It will not do at all to provide open web access to documents that are sensitive and access controlled. Luckily, many content management systems provide web applications that provide just such access. EMC’s Documentum, for instance, provides a web application called WebTop. Other content management systems, such as SharePoint and LiveLink, primarily interact with their users through a web interface.

Often the difficult part is just exploring the repository’s web application enough to figure out what the right URL is to use. In my experience, the information about how to get to a document in a repository via a URL is rarely well documented. This means you may need to put in some effort to discover it. If all else fails, a call to Customer Support might yield an answer.

In some cases, additional configuration information, such as the location and name of the web application, will be needed to build document URLs. You will want to keep this in mind when you design your connector’s configuration information. We will go through this process in the next section.

Docs4U DOCUMENT URL

In the case of our Docs4U example repository, there is already a method in the API which provides a file IRI for a document. We’ve seen it before:

```
public String getDocumentURL(String docID)
    throws D4UException;
```

This makes our life very easy, since no additional configuration information is needed to obtain the URL, just the document’s identifier.

Next, we’ll have a look at how we should go about listing the configuration information we will need.

7.1.4 Specifying the connector’s configuration information

The next step is straightforward. Every repository connection requires at least some information in order to connect to the repository. As we design our connector, we will need to decide exactly what information is needed, and think about how we will represent this in a `ConfigParams` object, and how we will want to present it in the connector’s configuration tabs. As we noted in Chapter 6, we have a choice of whether to use the simpler name/value paradigm, or the more complex hierarchical capabilities available. Obviously, this choice depends completely on the characteristics of the repository.

Sometimes it may be difficult to decide whether a particular piece of information should be considered configuration information or not. The general rule is that only information about “how” to communicate with the repository should rightly be considered configuration

information. For example, servers and ports that the connector needs to communicate with are clearly configuration information. Which folders and document names to include are clearly not.

But what about those less clear-cut cases? For instance, the information needed to build a proper URL for the document, as we discussed earlier? That kind of information could go either way. If it seems reasonable to allow the information to vary from job to job, then it should not be considered configuration information. But if it is a characteristic of the system, and ideally should be available for re-use by multiple jobs, then it may belong more naturally with the configuration information. I believe that URL information satisfies the latter test.

Docs4U CONFIGURATION INFORMATION

Looking at the Docs4U API class, it doesn't seem to tell us anything at all about how a connection to the repository should be set up. But that's because we're looking in the wrong place.

The API is meant to be instantiated by means of a factory class. The class can be found here:

http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/docs4u/src/org/apache/manifoldcf/examples/docs4u/D4UFactory.java. In this class, you will find the following method:

```
public static Docs4UAPI makeAPI(String root)
    throws D4UException;
```

The `root` argument is in fact meant to be a directory path that you supply to the Docs4U instance. Since Docs4U is file-system based, it makes sense that that's all we will need to establish a connection. Therefore, our configuration information is about as simple as one can imagine; a single name/value type parameter should suffice. We'll name it `rootdirectory`, and its value will be the complete canonical path of the repository root.

Next, we'll consider how a user of this connector should choose documents, and what parts of the document should be sent to the index.

7.1.5 Choosing the connector's document specification information

We've already talked a bit about how to decide whether a piece of information should be considered configuration information or not. The other kind of information that is managed by a repository connector is document specification information.

Document specification information is meant to answer the question of "what" should be included in a job. The tabs that allow a user to edit this information in the Crawler UI therefore do not appear when you edit a connection definition for the connector, but instead appear when you edit a job that uses the connector. Because the connection definition information is already available at that time, the methods that implement your connector's document specification tabs can set up and use an actual connection to the repository to supply choices and make the user experience better.

But we haven't yet talked about how to determine what document specification information you will actually need for a connector. Content repositories often have multiple ways of organizing information. For example, many of them have a notion of a folder hierarchy, but there are also some repositories that organize content by means of metadata values. Some repositories permit more than one method. What is the right way? The answer is actually pretty simple. Your connector must support the ability to select documents in the manner most consistent with how users of the repository typically do it. The people who contribute content to a repository will obviously be using the paradigm or paradigms that the repository most encourages for organization, so the connector you write should support those same paradigms.

METADATA

Part of what goes into the index for a document from a repository is metadata information. This is usually a set of name/value pairs, but it can also sometimes be more complex structures, such as multi-valued arrays. Clearly, any description of what metadata information to include with a document belongs with the connector's document specification information, since it describes "what" to include.

But the question about what metadata to send to an index inevitably arises. Should all of it be sent for every document? Should you allow a user to pick and choose what metadata to send on a per-job basis? Also, what about metadata information that doesn't directly come out of the repository as metadata, such as a document's path? This is a decision that depends strongly on how people use the repository. It makes little sense to include metadata in a document index if the metadata does not include indexable user content. If you decide that a document's metadata might contain useful information, then you still need to decide whether to simply take all of it, or allow a user the chance to pick and choose, based on the proportions. There are no hard and fast rules.

The only metadata issue your repository connector probably does **not** need to worry about is the mapping, if any, between the names of metadata in the content repository, and the names of metadata in the search index. This is because, in ManifoldCF, it is the current standard to make that mapping be the responsibility of output connectors.

DOCS4U DOCUMENT SPECIFICATION

Looking at the Docs4U API class, it is clear that the Docs4U repository has no notion of folders, so that cannot be the way we go about selecting documents for a job. Instead, it appears that the repository's `findDocuments()` method accepts one thing in addition to the date range: a map of metadata names and values. Thus, we can conclude that people must look up documents in the Docs4U repository primarily by means of these metadata values. Therefore we'll want to support that way of describing documents in the document specification. But how should it work?

The Docs4U API's `findDocuments()` method works by restriction. When there are no restrictions, you get the entire set of documents in the repository, while the more restrictions you add to the map, the fewer documents get returned. We will need to decide whether that

paradigm is reasonable for a ManifoldCF job, or whether we'd like to do things a bit differently when crawling. We could, for example, decide we should use an additive model, where the more name/value pairs you specify, the more documents you get. In that case, each name/value pair is really behaving like an inclusion rule of some kind. Partly because Docs4U is only an example, and has no real users, let's do it this way to see how that might work out.

What about including metadata information in the index? The `D4UDocInfo` method used to get at it is the following:

```
public String[] getMetadata(String metadataName);
```

As you can see, this metadata can potentially be multivalued. Other than that, there is little or no indication of its utility for an index; it's entirely based on what the people using the repository have chosen to do. Given that, it is reasonable to give users the flexibility of choosing what metadata to include in the document index, and what metadata to ignore, so that is what we'll plan on doing.

Table 7.2 summarizes what we've decided we need for document specification information.

Table 7.2 Document specification information for the Docs4U repository

Specification node	Attributes	Meaning
findparameter	name, value	Independent document metadata criteria to include in separate <code>findDocuments()</code> seeding searches
includedmetadata	name	The name of a metadata item to include along with the document contents for indexing

Congratulations! Now that we've chosen the form of the document identifier and access token, planned for how to build a document URL, and designed what the configuration and document specification information should look like, we're finally done with making the really big decisions. Next, we'll start figuring out how to begin the implementation.

7.2 Preparing to implement a repository connector

Once we've made the basic decisions as to what the connector will look like to the outside world, we can begin to think about the implementation itself. This step is also part of the design process, but it's mostly focused on the details of how the connector will work internally and interact with the ManifoldCF framework. In order to be able to plan this part of the connector's implementation, you'll probably need to know more about the repository APIs you will be using, and how they behave.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

7.2.1 The *IRepositoryConnector* interface

It is time to introduce the interface we will need to implement in order to create our repository connector class. The full class name for the interface is `org.apache.manifoldcf.crawler.interfaces.IRepositoryConnector`. It extends the `IConnector` interface we've discussed already in Chapter 6, so we will not describe those methods again here. The methods that are unique to `IRepositoryConnector`, and their meanings, are summarized in table 7.1.

Table 7.1 `IRepositoryConnector` methods and their meanings

Method	Meaning
<code>getConnectorModel()</code>	Returns the model the connector wishes to use for seeding.
<code>getActivitiesList()</code>	Returns the list of the activity names which the connector will use when it records history records.
<code>getRelationshipTypes()</code>	Returns the list of types of relationships between documents for this connector, which can be used to filter by hop count.
<code>getMaxDocumentRequest()</code>	Returns the maximum number of document identifiers the connector wishes to receive at once, for getting versions or processing.
<code>outputSpecificationHeader()</code>	Output the HTML code meant for the <HEAD> section of a page, for this connector's document specification tab presentation in the Crawler UI.
<code>outputSpecificationBody()</code>	Output the HTML code meant for the <BODY> section of a page, for this connector's document specification tab presentation in the Crawler UI.
<code>processSpecificationPost()</code>	Process the form data posted by <code>outputSpecificationBody()</code> above, and modify a document specification accordingly.
<code>viewSpecification()</code>	Format document specification information into HTML for viewing.
<code>requestInfo()</code>	Called from the ManifoldCF API Service, to allow retrieval of connection-specific information, usually for the purpose of building an external user interface.
<code>getBinNames()</code>	Given a document identifier, obtain the bin names the identifier belongs to.
<code>addSeedDocuments()</code>	Add documents to the job queue, as part of the seeding

	process.
<code>getDocumentVersions()</code>	Obtain version strings for documents given document identifiers.
<code>processDocuments()</code>	Process documents (extract content and discover new document identifiers), given existing document identifiers and document version strings.
<code>releaseDocumentVersions()</code>	Release any temporary data being kept as a result of a <code>getDocumentVersions()</code> method call.

Bert has had a look at these methods as well, and is having some trouble. “I just don’t understand what I’m supposed to do with them,” he mutters. “Each method seems clear enough by itself, but I just don’t see how it all fits together.” You respond that that is exactly what this section is all about. We are about to learn how to use these methods to build a working connector. We’ll start by learning how to choose a seeding model, and move on from there.

7.2.2 Choosing a seeding model

During the “seeding” phase of crawling, it is the connector’s responsibility to add documents to the job queue which represent starting points for the subsequent crawl. The connector code that actually does the work is the `addSeedDocuments()` method.

However, so far we’ve not said anything at all about what set of documents the `addSeedDocuments()` will add to the job queue. Will it include only documents that have been added and changed, or will it include deleted documents also? Will it be able to reliably re-add all documents it added from previous seeding operations, or will it be unable to add some of the documents it was able to add in the past?

ManifoldCF actually knows how to handle many such scenarios. All a connector needs to do is tell the system about the *minimum* set of documents that it is consistently capable of returning via `addSeedDocuments()`. For example, suppose the repository could reliably tell you about documents that had been added or changed, and it could also tell you about deletions that had occurred, but only some of the time, maybe because there was some sort of cleanup taking place periodically. Then, you would want to tell ManifoldCF only that you would be reliably including documents that had been added or changed, since that would be the minimum functionality you could reliably claim to support.

The situation is further complicated by the fact that the `addSeedDocuments()` method receives a time range as an argument. The contract for this method is thus that it must at least add the documents as agreed by its stated contract, which are documents that were variously added, changed, or deleted within the time range specified. The method can choose to return *more* documents, if it has no choice, but it must never return *fewer*.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

The method where you tell ManifoldCF which model it is that the connector will be using is `getConnectorModel()`. The possible values that can be returned from this method are listed in table 7.2.

Table 7.2 Connector models, as returned by `getConnectorModel()`

Model	Meaning
MODEL_ALL	Always returns all documents, regardless of time ranges or any other consideration.
MODEL_ADD	Returns at least the documents that have been added within the specified time range.
MODEL_ADD_CHANGE	Returns at least the documents that have been added or changed within the specified time range.
MODEL_ADD_CHANGE_DELETE	Returns at least the documents that have been added, changed, or deleted within the specified time range.
MODEL_CHAINED_ADD	Like MODEL_ADD, except that the contract is met only when considering documents discovered through chaining. See below.
MODEL_CHAINED_ADD_CHANGE	Like MODEL_ADD_CHANGE, except that the contract is met only when considering documents discovered through chaining. See below.
MODEL_CHAINED_ADD_CHANGE_DELETE	Like MODEL_ADD_CHANGE_DELETE, except that the contract is met only when considering documents discovered through chaining. See below.
MODEL_PARTIAL	May not return the same documents on each request, usually because the information as to what documents are in the set to be seeded changes unpredictably.

Something to consider before deciding on what model your connector says it is going to use is how it defines “change”. The connector can only claim that it properly reports changed documents only if *all possible changes that might cause a document to need to be reindexed* are taken into account. For example, suppose a repository is able to accurately relate when a document’s content changes, but not when its security information changes. Should the connector indicate that it is using `MODEL_ADD`, or should it report `MODEL_ADD_CHANGE` instead? Since the security information is part of the information needed when the document is indexed, then the seeding model **cannot** be `MODEL_ADD_CHANGE`.

Another important consideration for the proper choice of model is what the meaning of *chaining* is. We have seen chaining mentioned in table 7.2, where there is a class of models that include the word `CHAINED` in their names. In some cases, typically in the case of repositories that organize documents into a hierarchy, there may be no way to list all the documents in the repository at seeding time. Instead, documents are located by means of a *discovery path*, where one document leads to another, and from there to yet another, etc. If that is how documents are organized, then one of the `CHAINED` models may well be the appropriate one to use. For example, `MODEL_CHAINED_ADD_CHANGED` means that the seeding method will return at least those documents which have been added or changed, when you consider all the documents that can be reached by means of a discovery path from those which have returned directly by the seeding method. As you might expect, it's also a consideration in a `CHAINED` model whether or not your repository and your connector allows the discovery path to be traversed, so that changes in a leaf document, for instance, cause parents of that leaf document to also be recognized as having been changed.

SEEDING MODEL FOR Docs4U

Looking at the Docs4U API, the method we would need to use for seeding is the following:

```
public D4UDocumentIterator findDocuments(Long startTime, Long endTime,
    Map metadataMap)
    throws D4UException;
```

From the method's Javadoc comment, it appears that this method may have the ability to find documents that have been added or changed within a specified time window. But, unfortunately, there is simply not enough detail provided for us to know exactly what the method means by "change". This is all too typical when one is trying to build a third-party integration – you are largely at the mercy of the available documentation, which sometimes means that you also need to have access to people who know the repository thoroughly in order to make progress. The alternative is to have a willingness to experiment, to write many short throw-away programs using the API for the sole purpose of testing how it really behaves.

Luckily, I wrote the API for Docs4U, so I can assure you that what the API means by "change" is broad. The document is considered to have changed when any part of the document which Docs4U tracks is saved. By our criteria above, this means that we should be returning `MODEL_ADD_CHANGE` from the `getConnectorModel()` method.

7.2.3 What activities should be supported?

Every repository connector has a chance to contribute records to the crawl history, so that it is easy for a user of that connector to see what is happening right from the Crawler UI. The way in which this is done is as follows.

- Return the names of all the activities your connector records by means of the `getActivitiesList()` connector method
- At appropriate times, write the records corresponding to those activities through the

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

methods provided

- Make sure that the names of your activities are limited to a maximum length of 64

Although this sounds simple, and it is, I haven't yet explained completely how your connector is meant to interact with the ManifoldCF framework. For example, you probably have no idea at the moment how you would go about coding up that second bullet above. This is an interesting topic in itself, and worth a short digression.

The three most important connector crawling methods – `addSeedDocuments()`, `getDocumentVersions()`, and `processDocuments()` – each receive one argument which is of the type `IxxxActivity`, where the 'xxx' stands for "Seeding", "Version", or "Process", respectively. These interfaces provide access to ManifoldCF functionality that your connector can use during the execution of the respective method. For example, if you look at the `ISeedingActivity` interface, you will see that there is a method available which your connector code is supposed to call for every seed document it discovers:

```
public void addSeedDocument(String documentIdentifier,
    String[] prereqEventNames)
    throws ManifoldCFException;
```

We'll discuss what the rest of these arguments all mean later. The point to remember now is that this is the place to look when you need to cause the framework to do something.

Sure enough, `ISeedingActivity` extends another interface called `IHistoryActivity`, which provides the following method which is meant for your connector to use to record history information:

```
public void recordActivity(Long startTime, String activityType,
    Long dataSize, String entityIdentifier, String resultCode,
    String resultDescription, String[] childIdentifiers)
    throws ManifoldCFException;
```

The `activityType` argument for this method corresponds directly to one of the return values of `getActivitiesList()`. All you need to do is tell ManifoldCF what activities you can possibly return, and then call the `recordActivity()` method when you wish to create a history record. It's that simple.

ACTIVITIES FOR Docs4U

Thinking again about the Docs4U example, it seems like it might be valuable to record document fetches from the repository using the history mechanism. So let's tentatively plan to return one activity from the `getActivitiesList()` method, called "fetch".

7.2.4 What relationship types should there be?

As we have seen, ManifoldCF provides some fairly sophisticated support for calculating the hop count between a seed and a document. Your connector may or may not be suited to take advantage of this support, and even if it is suited, you may choose not to implement it. But let's first explain what it is all about.

The `getRelationshipTypes()` connector method allows you to tell ManifoldCF the names of types of relationships that you intend to provide tracking information for. For example, documents in a hierarchy might be considered to have a "child" relationship type.

If you thought it might be useful to limit documents based on the depth of the hierarchy, your connector could return `child` as the one of the values from `getRelationshipTypes()`. This would make the Hop Count tab appear in the Crawler UI for all jobs involving your connector, and the user could then limit the number of hops of relationship type `child`.

If your connector claims that it supports a certain type of relationship, then it has the additional responsibility of making sure that whenever it adds a document to the job queue, it provides both the parent document's document identifier as well as the relationship type.

The appropriate method in `IProcessActivity` is the following:

```
public void addDocumentReference(String localIdentifier,
    String parentIdentifier, String relationshipType,
    String[] dataNames, Object[][] dataValues, Long originationTime,
    String[] prereqEventNames)
    throws ManifoldCFException;
```

Once again, I promise to explain these arguments in more detail later. All you need to do at this point is remember that you have the responsibility of providing additional relationship information, if you choose to support hop count.

HOP COUNTS AND DOCS4U

The Docs4U example repository doesn't seem to support any intrinsic notion of relationship between documents. This means there is no reasonable option involving hop counts available, and we will therefore not be implementing it in our example.

7.2.5 Determining a document's throttling bins

We will also need to choose some sort of mapping from a document identifier to one or more throttling bins. As you may recall from Chapter 2, throttling bins are the classes of documents that are considered equivalent from the point of view of throttling. A connector writer must decide what classes there may be, and how to assign individual documents to each class based solely on the document's document identifier.

The method `getBinNames()` is where this logic takes place. For all existing repository connectors, this method returns the server name to which the document belongs. This makes a certain degree of sense, since documents from a particular server represent the best way to define a class of documents for throttling purposes.

DOCS4U THROTTLING BINS

The Docs4U example repository accesses only the local file system. The only kind of throttling criteria that will make any sense is in regards to what physical disk a document comes from. While we could require the user to tell us this as part of a connection definition's configuration information, there is an easier way. We can achieve our goal by simply using the repository's canonical path as the bin name for all documents which come from it.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

7.2.6 Carry-down data

Once in a while, data from one repository document should ideally affect how another document is indexed. The best current example of this linkage is the RSS connector. For the RSS connector, there is a distinction between feeds (which are documents that are not indexed), and content (which are documents which do get indexed). Some of the data from the feed, such as title and date, must be indexed when the content documents are indexed.

Since each document is treated by ManifoldCF as being a completely separate entity, this would seem to be difficult on the face of it. But ManifoldCF has a feature which makes it easy to do something like this. This feature goes under the name of *carry-down data*, and it is generally helpful when you want to transfer information to a document that is referenced from another document. The referenced document is called the *child document*, and the document where the reference is made is correspondingly called the *parent document*.

The information that can be transferred is structured in the form of names and sets of values. A name can have zero or more values associated with it. These values are not ordered, but each value can be of unlimited length. The method you would use to set carry-down information is `IProcessActivity.addDocumentReference()`, and the method you would use to retrieve it is `ICarrydownActivity.retrieveParentData()`.

In order to be able to decide just how to use carry-down data optimally in your connector, you may need to know some more about how it is modeled. For instance, suppose a document is a child of more than one parent document? Or, suppose that the data that the parent is sending to the child is itself coming from carry-down data passed to the parent?

The answers should be somewhat reassuring. Carry-down information from more than one parent is permitted; the child will receive each of the data items sent from the sum total of all of its parents, with duplications removed. For example, if one parent sends "title=me", and the other parent sends "title=you", the child will receive both "me" and "you" for the value of the "title" carry-down data item. Furthermore, the ManifoldCF framework code is smart enough to handle extended transitive carry-down situations properly, where document A transmits data to document B which transmits some of that data to document C. But there is one caveat: it is not guaranteed to be very efficient about it. You will need to visit Chapter 12 for a full explanation of why this is, but what you should take away right now is that the longer the chain of transitive carry-down transmission, the more inefficient ManifoldCF will likely become in order to deal with this. As you design your connector, you will probably want to keep that in mind.

Luckily, Docs4U has no need for carry-down information whatsoever. So we can move on to the next design topic: prerequisite event management.

7.2.7 Prerequisite events

Another feature our connector might benefit from is known as *prerequisite event management*. This is a way of controlling the job queue so that interactions between multiple threads that are trying to do the same thing are coordinated.

The prerequisite events mechanism can best be viewed as a gating mechanism. The way it works is that the connector, when it adds documents to the job queue, can also provide a set of *events* for each such document. When the connector is processing any document, it can choose to *begin an event sequence*, as needed, while sensitive processing is taking place. During the time that an event is asserted, ManifoldCF will not hand off for connector processing any documents that it has recorded as being dependent on that event, until the event sequence is completed. (Documents dependent on the event that are already under the control of the connector will be aborted, to be retried later, if the connector is written properly.) But why would anyone need to use such a feature?

The two ManifoldCF connector examples where prerequisite events are needed both involve web crawling. The simplest use is coordinating the fetch of the `robots.txt` file from a server so that it is only be fetched once, even though many threads might be simultaneously in a position to obtain that file at the same time. It is solved by having a separate “robots fetch event” for each target server, and including the appropriate such event as a dependency of all documents which come from that server. A second, more complicated, case involves how the web connector performs session-based login to a protected web site. A web site’s login sequence may be complicated and involve a number of pages and forms, and during that whole time fetches of protected content pages would obtain incorrect content, so they must be blocked until the login sequence is complete. Thus, there is an event defined for every such body of protected content, and the web connector is coded to attach that event to every page that the connector is told will be login protected.

We’ll be describing the actual `XXXXActivity` methods that support prerequisite events later in the chapter. For now, all we need to decide is whether we’re going to need this functionality for our Docs4U connector, which we won’t. So we’re free to consider the last topic: connector-specific database tables.

7.2.8 Connector-specific database tables

In rare cases, it is necessary to keep unbounded amounts of data somewhere in order for a connector to work well. For example, the web connector needs to keep the contents of the `robots.txt` URL for each site they visit. This data is needed in order to evaluate every discovered document reference, to determine if the reference should be considered valid for the purposes of crawling. As is usually the case, for the web connector examples there were a number of possible implementation options.

One option would have been to keep the `robots.txt` information in memory. But, as we’ve already pointed out, the number of sites is not necessarily bounded, and therefore if we relied on memory for this information, we would be violating one of the basic connector ground rules as laid out in Chapter 6.

A second option would have been to fetch the `robots.txt` URL just prior to every page fetch. This option would not have violated any of the rules in Chapter 6, but it would double

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

the number of page fetches, halving overall performance. This, too, would have been undesirable.

The correct solution for the web connector was to use a connector-specific database table. This is certainly allowed; that is why the `IConnector` methods `install()` and `deinstall()` exist. Combined with the advanced caching available using the `ICacheManager` core component described in Chapter 5, you have plenty of power available to do what is necessary, while obeying the bounded memory requirements of Chapter 6.

The Docs4U connector does not have any use for connector-specific database tables, so our example will not include any. Working with such tables will be left as an exercise for the reader.

We've now made all the decisions we reasonably can make before we begin to write code! By now, you should begin to see how we'll go about writing most of the methods originally laid out in table 7.1. Next, we'll write that code, and try it out.

7.3 Coding the connector

Now it is finally time to write your first connector! We've already made most of the key decisions we'll need. All that remains is putting together the code.

One potential timesaver, when it comes to coding, is to extend a base class that will do much of the work. For every kind of connector, one of these base classes exists. For repository connectors, the recommended base class can be found at `org.apache.manifoldcf.crawler.connectors.BaseRepositoryConnector`. It is generally a good idea to extend this class in any case because when the `IRepositoryConnector` interface changes some effort is usually made to provide backwards compatibility by this means. This also means that it is unnecessary for us to provide implementations for some of the lesser used methods, which will save us some work.

When writing a connector, I like to develop it in stages. First off, I usually write all the methods that `IConnector` specifies, and leave stubs for all the rest. This allows me to try out the configuration portion of the connector's UI, and get the connection logic all working, before I move on.

7.3.1 Writing the `IConnector` methods

The base class provides simple implementations for several of the `IConnector` methods. These are summarized in table 7.3.

Table 7.3 `BaseRepositoryConnector` methods implementing `IConnector` methods, and their functions

Method	Meaning
<code>install()</code>	Does nothing; presumes no additional database tables required

<code>deinstall()</code>	Does nothing; presumes no additional database tables need to be removed
<code>connect()</code>	Records configuration information in a local copy
<code>check()</code>	Returns "Connection working"
<code>poll()</code>	Does nothing
<code>disconnect()</code>	Removes local copy of configuration information
<code>getConfiguration()</code>	Returns local copy of configuration information
<code>clearThreadContext()</code>	Removes local thread context reference
<code>setThreadContext()</code>	Saves local thread context reference

Since the Docs4U connector will not need to install or use any additional database tables, we can leave the `install()` and `uninstall()` methods just as they are implemented by the `BaseRepositoryConnector` class. Likewise, the base class's `getConfiguration()` and `setConfiguration()` methods should work fine, as should `setThreadContext()` and `clearThreadContext()`. But for the `connect()` method, we'll want to get hold of the `rootdirectory` parameter before calling the base class's `connect()` method. For the `poll()` method, we might as well demonstrate good external connection practices, even if there is no real external connection involved, so we'll implement an on-demand actual connection method, and override `poll()` to show how to make the connection expire appropriately.

The major implementation effort will therefore be in writing the connector's Crawler UI configuration tab methods – `outputConfigurationHeader()`, `outputConfigurationBody()`, `processConfigurationPost()`, and `viewConfiguration()`. The single parameter, `rootdirectory`, is well suited to be represented in the Crawler UI by a simple text box. Fortunately, in Chapter 6, we have already demonstrated a similar text box, and we can thus model our Docs4U configuration on that code. On the theory that it's easier to lift working code from elsewhere than write code anew, we'll simply borrow it, and the corresponding Velocity template, and change what we need to. We'll also need to write a `check()` method, based on some trial usage of an appropriate Docs4U API method. Listing 7.1 shows the completed `IConnector`-required methods in their final form. The complete connector can be found at http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/repository_connector_example/src/org/apache/manifoldcf/crawler/connectors/docs4u/Docs4UConnector.java.

Listing 7.1 IConnector-related methods and data

```
protected final static String PARAMETER_REPOSITORY_ROOT = #1
```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

```

    "rootdirectory";                                     #1
protected final static long SESSION_EXPIRATION_MILLISECONDS = #2
    300000L;                                             #2

protected String rootDirectory = null;                 #3

protected Docs4UAPI session = null;                   #4
protected long sessionExpiration = -1L;                #4

public void outputConfigurationHeader(IThreadContext threadContext,
    IHTTPOutput out,
    Locale locale, ConfigParams parameters, List<String> tabsArray)
    throws ManifoldCFException, IOException
{
    tabsArray.add("Repository");                       #5
    Messages.outputResourceWithVelocity(out, locale,    #6
        "ConfigurationHeader.html", null);            #6
}

public void outputConfigurationBody(IThreadContext threadContext,
    IHTTPOutput out,
    Locale locale, ConfigParams parameters, String tabName)
    throws ManifoldCFException, IOException
{
    // Output the Repository tab
    Map<String, Object> velocityContext = new HashMap<String, Object>();
    velocityContext.put("TabName", tabName);            #7
    fillInRepositoryTab(velocityContext, parameters);  #8
    Messages.outputResourceWithVelocity(out, locale,    #8
        "Configuration_Repository.html", velocityContext); #8
}

protected static void fillInRepositoryTab(
    Map<String, Object> velocityContext, ConfigParams parameters)
{
    String repositoryRoot =                            #9
        parameters.getParameter(PARAMETER_REPOSITORY_ROOT); #9
    if (repositoryRoot == null)                         #10
        repositoryRoot = "";                          #10
    velocityContext.put("repositoryroot", repositoryRoot);
}

public String processConfigurationPost(IThreadContext threadContext,
    IPostParameters variableContext, ConfigParams parameters)
    throws ManifoldCFException
{
    String repositoryRoot = variableContext.getParameter( #11
        "repositoryroot");                             #11
    if (repositoryRoot != null)                         #12
        parameters.setParameter(PARAMETER_REPOSITORY_ROOT, repositoryRoot); #12
    return null;
}

public void viewConfiguration(IThreadContext threadContext,
    IHTTPOutput out,
    Locale locale, ConfigParams parameters)

```

```

        throws ManifoldCFException, IOException
    {
        Map<String,Object> velocityContext = new HashMap<String,Object>();
        fillInRepositoryTab(velocityContext,parameters);           #13
        Messages.outputResourceWithVelocity(out,locale,           #13
            "ConfigurationView.html",velocityContext);             #13
    }

    protected Docs4UAPI getSession()
        throws ManifoldCFException, ServiceInterruption
    {
        if (session == null)
        {
            try                                                     #14
            {                                                         #14
                session = D4UFactory.makeAPI(rootDirectory);         #14
            }                                                         #14
            catch (D4UException e)                                   #14
            {                                                         #14
                Logging.connectors.warn("Docs4U: Session setup error: "+ #14
                    e.getMessage(),e);                                #14
                throw new ManifoldCFException("Session setup error: "+ #14
                    e.getMessage(),e);                                #14
            }                                                         #14
        }
        sessionExpiration = System.currentTimeMillis() +           #15
            SESSION_EXPIRATION_MILLISECONDS;                       #15
        return session;
    }

    protected void expireSession()
    {
        session = null;                                             #16
        sessionExpiration = -1L;                                     #16
    }

    public void connect(ConfigParams configParameters)
    {
        super.connect(configParameters);                             #17
        rootDirectory = configParameters.getParameter(              #18
            PARAMETER_REPOSITORY_ROOT);                             #18
    }

    public void disconnect()
        throws ManifoldCFException
    {
        expireSession();                                             #19
        rootDirectory = null;                                         #20
        super.disconnect();                                           #21
    }

    public String check()
        throws ManifoldCFException
    {
        try

```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

```

    {
        Docs4UAPI currentSession = getSession(); #22
        try
        {
            currentSession.sanityCheck(); #23
        }
        catch (D4UException e)
        {
            Logging.connectors.warn("Docs4U: Error checking repository: "+ #24
                e.getMessage(), e); #24
            return "Error: "+e.getMessage(); #24
        }
        return super.check(); #25
    }
    catch (ServiceInterruption e)
    {
        return "Transient error: "+e.getMessage(); #26
    }
}

public void poll()
    throws ManifoldCFException
{
    if (session != null)
    {
        if (System.currentTimeMillis() >= sessionExpiration) #27
            expireSession(); #27
    }
}

```

- #1 Define the parameter names we need
- #2 Sessions will expire after 5 minutes idleness
- #3 The root directory parameter, or null if not connected
- #4 The session handle and timeout value, or null if expired
- #5 There is one configuration tab called "Repository"
- #6 Output Javascript using Velocity template
- #7 Send tab name to template
- #8 Fill in Repository tab values and call template
- #9 Get existing value of root parameter, if any
- #10 If no existing value, set a default
- #11 Obtain the parameter value from the posted data
- #12 Set the parameter to the value
- #13 Invoke the Velocity template for viewing
- #14 If no session, create one
- #15 Set the session expiration time to now plus five minutes
- #16 To expire a session, set it to null and the expiration time to -1
- #17 Call superclass connect method to use base class correctly
- #18 Grab the repository root parameter for later use
- #19 Expire any current session
- #20 Clear any configuration-related data
- #21 Remember to call superclass disconnect method
- #22 Get or create a session
- #23 Try a sanity check
- #24 Report any error as a returned string for display, and log
- #25 If all is OK, return superclass check value
- #26 ServiceInterruptions are transient errors

#27 If we have exceeded the time cutoff, expire the session

At #1, we start by defining the name of the parameter we'll use to contain the Docs4U root repository directory. At #2, we set a variable which will determine the length of time an idle session can hang around.

Next we define some local variables that the connector will use to manage parameters and sessions. At #3, we declare a variable which will contain the value of the root directory parameter, if `connect()` has been called, or will contain `null` if the connector class instance is disconnected. At #4 we define the local session variable, which will contain a Docs4U API object reference if a session has been created, or will contain `null` if the session has either not been created yet, or has been expired. Along with it, we define a variable which will indicate the time (in milliseconds since Jan 1, 1970) when the session will no longer be valid.

Moving on to the UI methods, at #5 we declare a single configuration tab called "Repository". We output a Javascript check method at #6 using Velocity which works together with the form elements used for the display of the tab, to make sure it is not blank, and to alert the user and prevent the form from being submitted if it is. In the `outputConfigurationBody()` method, at #7 we set the tab name in the Velocity context, and then fill in the Repository tab's parameter values and invoke its Velocity template at #8. At #9 we load the "root" parameter value from the current configuration information. If it is not yet present, then at #10 we set it to a default value. For this particular situation, the default value is blank. The `processConfigurationPost()` method first obtains the parameter value from the posted data, at #11, and sets the value into the configuration data at #12. Finally, the `viewConfiguration()` method uses the same method as the `outputConfigurationBody()` method does to fill in values required by the Velocity template, and renders the template, at #13.

Session management is largely done through the `getSession()` method, which creates a new session if it does not yet exist at #14, converting any exceptions from this action into either a `ManifoldCFException`, or a `ServiceInterruption` exception. `ServiceInterruption` exceptions are meant to capture transient problems that may resolve with the passage of time. While it is not possible for Docs4U to have an error of this kind, I've left the infrastructure in place in the Docs4U connector to demonstrate how this should work. It will be up to you to determine what repository exceptions or return values correspond to permanent problems (in which case `ManifoldCFException` should be thrown), versus transient ones. In any case, once we have a session, even if we did not create it this time, we reset the expiration time to the current time plus five minutes, at #15. The `expireSession()` method, which is called from various places later on, clears out both the session reference as well as the expiration time, at #16.

Now we reach the connection pool logic, starting with the `connect()` method. Much of the functionality is handled by the superclass, so we must not forget to call the superclass

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

`connect()` method, at #17. But we keep around the current value of the repository root, at #18. The `disconnect()` method first expires any existing session, at #19. Then it clears out anything kept around from the `connect()` method earlier, at #20. Finally, the job is finished by calling the superclass `disconnect()` method, at #21.

Finally, we implement the `check()` method, which is designed to give the user feedback as to the connection's correctness and health, either through the UI or the API. At #22, we start by getting a session handle. We use the handle at #23 to perform a repository sanity check, which will throw an exception if there is a problem. We catch non-transient errors and incorporate the error messages into the return value at #24. If we don't see any errors, we return the "Connection working" message that is returned by the superclass `check()` method, at #25. At #26, we capture any transient exceptions, and report those with a slightly different message.

The last method implemented is the `poll()` method. All that this method does is expire the session when the current time exceeds the expiration time variable, at #27.

The Velocity templates themselves are pretty straightforward. See listing 7.2 for the Javascript template (`ConfigurationHeader.html`).

Listing 7.2 The `ConfigurationHeader.html` Velocity template

```
<script type="text/javascript">
<!--
function checkConfigForSave()                                #A
{
    if (editconnection.repositoryroot.value == "")            #B
    {
        alert("Enter a repository root");
        SelectTab("Repository");                               #C
        editconnection.repositoryroot.focus();                 #D
        return false;                                          #D
    }
    return true;
}
//-->
</script>
#A Check things before we save the data
#B Repository root can't be empty
#C Be sure to select the tab where the problem is
#D Focus on the problem field, and return false
```

The template for the "Repository" tab is equally straightforward. See listing 7.3.

Listing 7.3 The `Configuration_Repository.html` Velocity template

```
#if($TabName == 'Repository')                                #A

<table class="displaytable">
  <tr><td class="separator" colspan="2"><hr/></td></tr>
  <tr>
    <td class="description"><nobr>Repository root:</nobr></td>      #B
    <td class="value">
      <input type="text" size="64" name="repositoryroot">
    </td>
  </tr>
</table>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>


```

        value="$Encoder.attributeEscape($repositoryroot) "/>                                #C
    </td>
</tr>
</table>

#else                                                                                      #D

<input type="hidden" name="repositoryroot"
    value="$Encoder.attributeEscape($repositoryroot) "/>

#end
#A If we are rendering the Repository tab, output something visible
#B Use description/value style
#C Don't forget to escape the attribute!
#D If not the Repository tab, preserve data in hiddens

```

The template for viewing the configuration information is shown in listing 7.4.

Listing 7.4 The ConfigurationView.html Velocity template

```

<table class="displaytable">
  <tr>
    <td class="description"><nobr>Repository root:</nobr></td>                                #A
    <td class="value">$Encoder.bodyEscape($repositoryroot)</td>                            #B
  </tr>
</table>
#A Use description/value style again
#B Escape in a manner appropriate for body this time

```

BUILDING AND RUNNING THE CONNECTOR

Now, let's try what we have so far. Building and running the repository connector example is straightforward. First, check out or unpack the repository connector example. For instance:

```

svn co
http://manifoldcf.inaction.googlecode.com/svn/trunk/edition_2_revised/reposi
tory_connector_example

```

Next, change to the `repository_connector_example` directory, and build the connector jar using the supplied ant build script:

```
ant jar
```

You can now add the connector jar and the supporting Docs4U jar to the Quick Start example. To do this, just copy the `build/jar/d4u-repository-connector.jar` file and the `lib/docs4u-example.jar` files from the example directory to your ManifoldCF build's `dist/connector-lib` directory. Then, edit the ManifoldCF Quick Start connector registration file `dist/connectors.xml` file to add the following XML tag at the end of the file:

```

<repositoryconnector name="Docs4U"
    class="org.apache.manifoldcf.crawler.connectors.docs4u.Docs4UConnector"/>

```

This tells the ManifoldCF Quick Start that you've added a new repository connector, which it should try to register when the Quick Start is started. In theory, you can now start

ManifoldCF in the normal way, and the new Docs4U connector should be registered and should become accessible. If you don't remember how to do that, it's just:

```
java -jar start.jar
```

Note: ManifoldCF also supplies a complete build environment for custom connectors. If you organize your connector source code and resources in a standard manner, you can leverage the Apache Ant build scripts that ship with the ManifoldCF 1.6 distribution and later. It is beyond the scope of this book to demonstrate how to do this, but ManifoldCF's included suite of connectors provides multiple examples.

Once ManifoldCF is up and running, we can open a browser and see if our new connector appears in the repository connector pulldown, and if it seems to work to the degree that it should. So, let's try to create a new connection definition, using the Docs4U repository connector. Browse to <http://localhost:8345/mcf-crawler-ui>, and click on the List Repository Connections navigation link. Then, click the Add new connection link. Provide a connection definition name on the Name tab, and then click the Type tab. You should see a pull-down that looks something like figure 7.1.

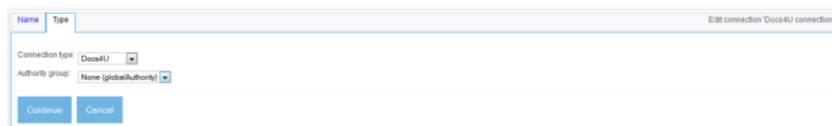


Figure 7.1 The Docs4U connector appears in the list of available connectors for a new repository connection definition.

Well, it looks like the connector has been registered, at least! Now let's see what it does. Select the Docs4U connector from the pulldown, and click the Continue button. You should see the Repository tab appear, as shown in figure 7.2.

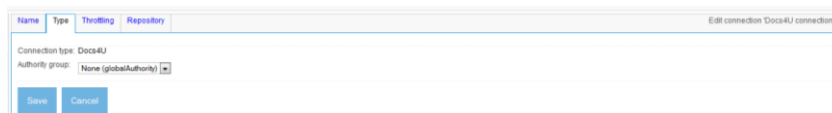


Figure 7.2 The Docs4U connector's outputConfigurationHeader() method has been called and the Repository tab has been added to the tab list.

We don't yet have a Docs4U repository built, but that's okay; we'll explore what happens when the configuration information is incorrect. First, try just clicking the Save button with an empty path. You should see an alert box pop up, similar to figure 7.3.

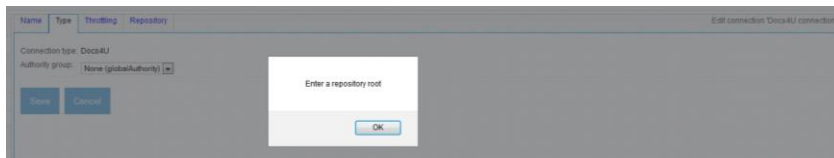


Figure 7.3 Checking out the form verification Javascript by trying to save the form without a repository root value.

This is because the `checkConfigForSave()` method we wrote explicitly prohibits the repository root parameter from being an empty string. So, click the **OK** button. The `Repository` tab should be selected and focus sent to the repository path field. Let's next enter a path which does not exist, and click the **Save** button. See figure 7.4.

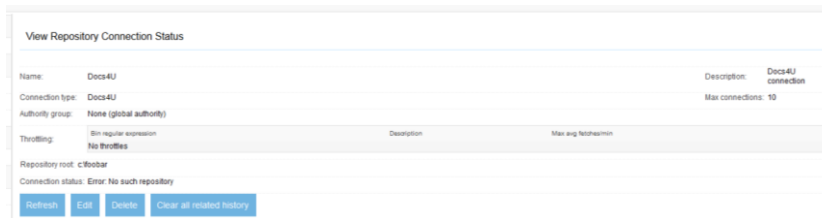


Figure 7.4 The view page for a Docs4U repository connection definition. Note that the connection status properly reflects that the repository root is incorrect.

It looks like we've successfully saved our Docs4U connection definition! Notice that the connection status is reporting that all is not well; our connector's `check()` method has figured out that the directory we gave was bogus. So this behavior is expected and welcome.

The last step in the tryout of the `IConnector` portion of our connector is to create a real Docs4U example repository, and point the connection definition at it. I've created an ant target specifically intended to help you create an appropriate example. In the `repository_connector_example` directory, type the following:

```
ant sample-repository
```

This should create a repository with some documents, metadata, users, etc. The repository will be located in the `repository` subdirectory underneath `repository_connector_example`. Then, edit the Docs4U connection definition once again, and change the bogus directory name to instead point at the repository you've just created. Click the **Save** button. You should see something like figure 7.5.

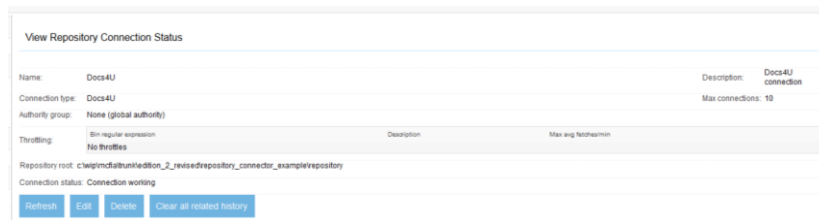


Figure 7.5 The Docs4U view connection page, with a correct repository root, showing the connection status of “Connection working”.

Congratulations! The `IConnector` portion of your new connector is now working as designed! Now we are ready to finish the job, and write the methods that implement the rest of the `IRepositoryConnector` interface.

7.3.2 Writing the `IRepositoryConnector` self-description methods

Now that we have working connection and session logic, we can move on to add implementations for those methods that are unique to the `IRepositoryConnector` interface.

Many of these methods are extremely simple, and we’ve in fact already just about written them as we walked through the design process for the connector. These super simple methods include `getConnectorModel()`, `getActivitiesList()`, `getRelationshipTypes()`, `getMaxDocumentRequest()`, and `getBinNames()`. Let’s start by formally writing those down. See listing 7.5.

Listing 7.5 The simplest `IRepositoryConnector` methods, and associated data

```
protected final static String ACTIVITY_FETCH = "fetch";           #A

public int getConnectorModel()
{
    return MODEL_ADD_CHANGE;                                     #B
}

public String[] getActivitiesList()
{
    return new String[]{ACTIVITY_FETCH};                         #C
}

public String[] getRelationshipTypes()
{
    return new String[0];                                        #D
}

public String[] getBinNames(String documentIdentifier)
{
    return new String[]{rootDirectory};                          #E
}
```

```

public int getMaxDocumentRequest()
{
    return 1;
}

```

#A One activity to be recorded, called “fetch”
#B Seeding will conform to ADD_CHANGE model
#C Return the one activity we’ve decided to support
#D No child relationships will be supported
#E Every document from the current connection returns the same bin name
#F We process one document at a time

There is nothing very exciting here; we have already discussed most of these earlier in the chapter, with the exception of the `getMaxDocumentRequest()` method. But this method requires a little explanation.

In many content repositories, it is much more efficient to work with batches of documents than access one document at a time. For this reason, several of the connector methods are capable of handling multiple documents at once, especially the `getDocumentVersions()` method and the `processDocuments()` method.

The connector writer is given the option of specifying the maximum number of documents that will be given to these methods at one time. This is not a simple choice, because there are tradeoffs involved. A number too small may not perform well, while a number too large has a different kind of problem – specifically, a single error processing any one of the documents will cause **all** of the documents in the entire set to be retried. Furthermore, there tend to be diminishing returns – it is not usually much of an additional advantage specifying one hundred documents rather than ten. For the Docs4U repository, which is based on the local file system, there may not be any advantage whatsoever in processing more than one document at a time, which is why I have implemented `getMaxDocumentRequest()` in the manner that I have.

7.3.3 Coding the *IRepositoryConnector* document specification UI methods

The next set of *IRepositoryConnector* methods we’ll tackle once again involve support for the Crawler UI. But in this case, we’re talking about the ones that build and manage the document specification, specifically `outputSpecificationHeader()`, `outputSpecificationBody()`, `processSpecificationPost()`, and `viewSpecification()`. In design, these methods are very similar to the methods that manage configuration, but there is one very important difference: you can be sure that the connector class instance will be connected and configured for thread use before any of these methods are called. Thus, they do not need or have an *IThreadContext* argument, and even more importantly, it is okay for them to communicate with the connector’s underlying repository to make the UI that they present to the user as friendly as possible.

That’s exactly what our example connector is going to do. We’re going to have it use the “connection” to the repository to populate a list of the metadata names which we can filter our documents based on. We’ll allow the user to build metadata matching criteria one name

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

and value at a time, and also allow the deletion of criteria that have already been added. For a change, we will use the tabular method of display, which will yield a more compact representation. And, since the relative complexity of the methods is higher, we'll break them up into a method per tab, which will make the code easier to read. See listing 7.6.

Listing 7.6 Principle IRepositoryConnector methods for manipulating Docs4U document specifications

```

public void outputSpecificationHeader(IHTTPOutput out, Locale locale,
    DocumentSpecification ds, List<String> tabsArray)
    throws ManifoldCFException, IOException
{
    tabsArray.add("Documents");
    tabsArray.add("Metadata");
    Messages.outputResourceWithVelocity(out, locale,
        "SpecificationHeader.html", null);
}

public void outputSpecificationBody(IHTTPOutput out, Locale locale,
    DocumentSpecification ds, String tabName)
    throws ManifoldCFException, IOException
{
    outputDocumentsTab(out, locale, ds, tabName);
    outputMetadataTab(out, locale, ds, tabName);
}

public String processSpecificationPost(IPostParameters variableContext,
    DocumentSpecification ds)
    throws ManifoldCFException
{
    String rval = processDocumentsTab(variableContext, ds);
    if (rval != null)
        return rval;
    rval = processMetadataTab(variableContext, ds);
    return rval;
}

public void viewSpecification(IHTTPOutput out, Locale locale,
    DocumentSpecification ds)
    throws ManifoldCFException, IOException
{
    Map<String, Object> velocityContext = new HashMap<String, Object>();
    fillInDocumentsTab(velocityContext, ds);
    fillInMetadataTab(velocityContext, ds);
    Messages.outputResourceWithVelocity(out, locale,
        "SpecificationView.html", velocityContext);
}

```

#1 Add the two Docs4U tabs
#2 Render the Javascript Velocity template
#3 Output each tab's HTML body independently
#4 Process each tab's post data independently
#5 Fill in the Velocity context with each tab's data
#6 Render the 'view' Velocity context

As you can see, the code above is mainly infrastructure; the methods and templates that do the actual work have yet to be presented. At #1, we add the tabs we will need. At #2, we output the Velocity template containing the Javascript. Clearly, we'll need to see the template to understand what is going on. For the body output, each tab has its own method, at #3. The post data is processed a tab at a time at #4. For viewing, we use a single template, but fill in data from each tab independently, at #5, and render the template at #6.

The rest of the specification UI code consists of Velocity templates and helper methods. For brevity, I'm going to present here a complete picture of only one of the two tabs, specifically the 'Metadata' tab. The rest of the code can be seen by inspecting the example. Let's start with the specification editing template for the 'Metadata' tab. This will provide some context for the rest of our discussion. See listing 7.7.

Listing 7.7 The Specification Metadata Velocity template

```
#if($TabName == 'Metadata') #1

<table class="displaytable">
  <tr><td class="separator" colspan="2"><hr/></td></tr>
  <tr>
    <td class="description"><nobr>Include:</nobr></td> #2
    <td class="value">

      #if($error != '') #3
        $Encoder.bodyEscape($error) #3
      #else
        #foreach($metadataattribute in $metadataattributes) #4
          #if($metadataselections.contains($metadataattribute)) #5
            <input type="checkbox" name="metadata" #5
              value="$Encoder.attributeEscape($metadataattribute)" #5
              checked="true"/> #5
          #else #6
            <input type="checkbox" name="metadata" #6
              value="$Encoder.attributeEscape($metadataattribute)"/> #6
          #end #6
          $Encoder.bodyEscape($metadataattribute) <br/> #7
        #end
      </td>
    #end
  </tr>
</table>

#else #8

  #foreach($metadataselection in $metadataselections) #9
    <input type="hidden" name="metadata" #9
      value="$Encoder.attributeEscape($metadataselection)"/> #9
    #end
  #end

#end
```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

- #1 Check we should display the tab or not
- #2 Display using the description/value paradigm
- #3 Preferentially display any error getting the metadata names
- #4 Loop over the list of metadata attributes
- #5 If the attribute is present in the selected set, make it be 'checked'
- #6 If the attribute was not selected, render it 'unchecked'
- #7 Display the attribute name beside the checkbox
- #8 If Metadata tab not on display, output hidden information instead
- #9 Loop through selected attributes only, and output as hidden

At #1, we decide whether we're rendering the tab for display or not. If we need to display it, then at #2 we use the description/value paradigm, where the value will be a list of checkboxes. But before we display the checkboxes, we check for any earlier errors communicating with the repository, and display those if present at #3. If no such errors, then at #4 we iterate over the list of metadata attributes that came from the repository, and at #5 and #6 we decide whether to render each attribute as already checked or not. We mustn't forget to display the attribute name itself, at #7. If the Metadata tab was not supposed to be displayed, at #8, we still must output appropriate hidden form elements contain its data, at #9.

Next, we'll clearly need to see the code that fills in the Velocity context data and calls the template in order to make sense of all of this. See listing 7.8.

Listing 7.8 Supporting Java methods for Metadata tab editing template

```
protected final static String NODE_INCLUDED_METADATA = #1
    "includedmetadata"; #1
protected final static String ATTRIBUTE_NAME = "name"; #1

protected void outputMetadataTab(IHTTPOutput out, Locale locale,
    DocumentSpecification ds, String tabName)
    throws ManifoldCFException, IOException
{
    Map<String, Object> velocityContext = new HashMap<String, Object>();
    velocityContext.put("TabName", tabName); #2
    fillInMetadataTab(velocityContext, ds); #3
    fillInMetadataSelection(velocityContext); #4
    Messages.outputResourceWithVelocity(out, locale, #5
        "Specification_Metadata.html", velocityContext); #5
}

protected static void fillInMetadataTab(
    Map<String, Object> velocityContext,
    DocumentSpecification ds)
{
    Set<String> metadataSelections = new HashSet<String>();
    int i = 0;
    while (i < ds.getChildCount())
    {
        SpecificationNode sn = ds.getChild(i++); #6
        if (sn.getType().equals(NODE_INCLUDED_METADATA)) #6
        {
            String metadataName = sn.getAttributeValue(ATTRIBUTE_NAME); #7
        }
    }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>


```

        metadataSelections.add(metadataName);                                #8
    }
}
velocityContext.put("metadataselections",metadataSelections);              #9
}

protected void fillInMetadataSelection(
    Map<String,Object> velocityContext)
{
    try
    {
        String[] matchNames = getMetadataNames();                          #10
        velocityContext.put("metadataattributes",matchNames);              #11
        velocityContext.put("error","");                                    #11
    }
    catch (ManifoldCFException e)
    {
        velocityContext.put("error","Error: "+e.getMessage());            #12
    }
    catch (ServiceInterruption e)
    {
        velocityContext.put("error","Transient error: "+e.getMessage());  #13
    }
}

protected String[] getMetadataNames()
    throws ManifoldCFException, ServiceInterruption
{
    Docs4UAPI currentSession = getSession();                                #14
    try                                                                    #15
    {                                                                    #15
        String[] rval = currentSession.getMetadataNames();                #16
        java.util.Arrays.sort(rval);                                       #17
        return rval;
    }
    catch (InterruptedException e)                                        #18
    {                                                                    #18
        throw new ManifoldCFException(e.getMessage(),                    #18
            e,ManifoldCFException.INTERRUPTED);                          #18
    }
    catch (D4UException e)
    {
        throw new ManifoldCFException(e.getMessage(),e);                #19
    }
}

protected String processMetadataTab(IPostParameters variableContext,
    DocumentSpecification ds)
    throws ManifoldCFException
{
    removeNodes(ds,NODE_INCLUDED_METADATA);                                #20

    String[] metadataNames =
        variableContext.getParameterValues("metadata");                    #21
    if (metadataNames != null)

```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

```

    {
        int i = 0;
        while (i < metadataNames.length)
        {
            String metadataName = metadataNames[i++];
            addIncludedMetadataNode(ds, metadataName);
        }
    }

    return null;
}

protected static void removeNodes(DocumentSpecification ds,
    String nodeName)
{
    int i = 0;
    while (i < ds.getChildCount())
    {
        SpecificationNode sn = ds.getChild(i);
        if (sn.getType().equals(nodeName))
            ds.removeChild(i);
        else
            i++;
    }
}

```

#1 Declare the node names and attribute names we will need

#2 Put the current tab name into the Velocity context

#3 Fill in the Metadata tab info

#4 Fill in the metadata attributes

#5 Invoke the Velocity template

#6 Look for all metadata nodes in the Document Specification

#7 Get the metadata name

#8 Add to the set of selected metadata items

#9 Put the whole set into the Velocity context

#10 Try to get the metadata names from repository

#11 If success, put names into context, and a blank error message

#12 If error, put an error message into the context

#13 If transient error, put a transient error into the context

#14 Use the standard method to get a Docs4U session handle

#15 Catch any exceptions thrown by Docs4U

#16 Get the list of metadata names, unsorted

#17 Sort them

#18 If InterruptedException thrown, rethrow exception type INTERRUPTED

#19 All other exceptions throw type GENERAL

#20 Remove all old metadata nodes from specification

#21 Look for 'metadata' items in the post context

#22 For all posted 'metadata' items, add a node to the specification

#23 Null return indicates no error

#24 Remove all nodes whose type matches the passed-in type

At #1, we describe the node types and attribute names we will be using to store metadata selections. For the actual rendering, at #2 we put the current tab name into the Velocity context for the template to use. At #3 we add the set of selected metadata names

to the context. At #4 we add the complete list of metadata names retrieved from the repository, and we invoke the Velocity template at #5.

To find the set of already-selected metadata names, we look for all nodes with the correct type at #6, get the name from the appropriate attribute at #7, and add to the set of selected names at #8. We put the completed set into the Velocity context at #9.

To get the complete list of metadata names known by the repository, we call the appropriate general method at #10, and if success put the names in the context and a blank error in the error variable, at #11. If failure, we generate the appropriate error string and put it into the context instead, at #12 and #13.

The general method to find metadata names from Docs4U starts by getting a session handle at #14. We catch any exceptions accessing Docs4U at #15, and obtain the list at #16, sorting them prior to return at #17. If an interrupted exception is thrown at #18, make sure we throw a ManifoldCF INTERRUPTED exception. All other exceptions use type GENERAL, at #19.

Handling post data starts by removing all old metadata specification nodes, at #20. Then, we look for posted metadata items at #21, and add a specification node for each at #22. Returning 'null' indicates no error, at #23.

Finally, we remove all nodes of the specified type, at #24.

The Javascript that supports the Metadata tab editing template is minimal. Listing 7.9 includes general infrastructure, plus those portions involved in the 'Metadata' tab.

Listing 7.9 Select portions of the SpecificationHeader.html Velocity template

```
<script type="text/javascript">
<!--

function checkSpecification()
{
    if (checkDocumentsTab() == false)                #A
        return false;                                #A
    if (checkMetadataTab() == false)                  #B
        return false;                                #B
    return true;
}

function checkMetadataTab()
{
    return true;                                     #C
}

//-->
</script>
#A Check the Documents tab
#B Check the Metadata tab
#C For the Metadata tab, nothing further need be done
```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

That was easy! Of course, the Metadata tab is not a very complex one, and doesn't need to repost in order to work. If it did, the amount of Javascript involved would be significantly larger, as it is with the Documents tab.

The viewing Velocity template is the only remain piece of the puzzle. See listing 7.10.

Listing 7.10 Selected portions of the SpecificationView.html Velocity template

```
<table class="displaytable">

  <tr>
    <td class="description"><nobr>Included:</nobr></td>           #A
    <td class="value">

#set($seendata = false)                                           #B
#foreach($metadataselection in $metadataselections)              #C
  #if($seendata)                                                  #D
    ,                                                              #D
  #end                                                            #D
  #set($seendata = true)                                          #D
    $Encoder.bodyEscape($metadataselection)                      #E
#end

    </td>
  </tr>
</table>
#A Stick with description/value paradigm
#B Set a flag so we know to output first comma
#C Iterate over selected metadata
#D Conditionally output a comma
#E Output the metadata name
```

Well, that was a lot of work! People often underestimate the amount of effort needed to support an acceptable UI for a connector. UI development of this kind can easily exceed the effort required for the actual crawling methods themselves. Luckily it is that it is often the case that existing UI pieces from other connectors can be readily reworked into code that will work in yours.

Next we're going to try out what we just wrote. We'll also try out the corresponding code for the Documents tab, since we haven't seen that yet.

TRYING OUT THE COMPLETED DOCUMENT SPECIFICATION UI

We're ready, once again, to try out the code we've written to see whether it works as we expect. Once the new jar is built and deployed, and ManifoldCF restarted, open a browser window and click the `List all jobs` link in the navigation area. Then, click the `Add new job` link, give your job a decent name, and click the `Type` tab. Finally, select the `Null` output connection you created back in Chapter 1, and the `Docs4U` repository connection you created earlier in this chapter, and click the `Continue` button. You should see a screen that looks something like figure 7.6.

Figure 7.6 A Docs4U job, showing the Documents and Metadata tabs.

That is good news - our tabs have appeared! Let's click on one of them. We'll start with the Metadata tab, which we just finished presenting the code for. See figure 7.7.

Figure 7.7 A Docs4U job, showing the Metadata tab.

Now, check one or two of the metadata names, and then click the Documents tab. You should see something like figure 7.8.

Figure 7.8 A Docs4U job, showing the Documents tab.

Here, you should be able to add metadata name/value pairs to the match list presented in the table by selecting a metadata name from the pull-down, filling in the value, and clicking the Add button. Try it and see what happens. See figure 7.9.

Figure 7.9 The Docs4U job Documents tab, with a match constraint added.

Now, you may add more criteria, or delete a row by clicking the Delete button next to it. When you are done, click the Save button. See figure 7.10.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

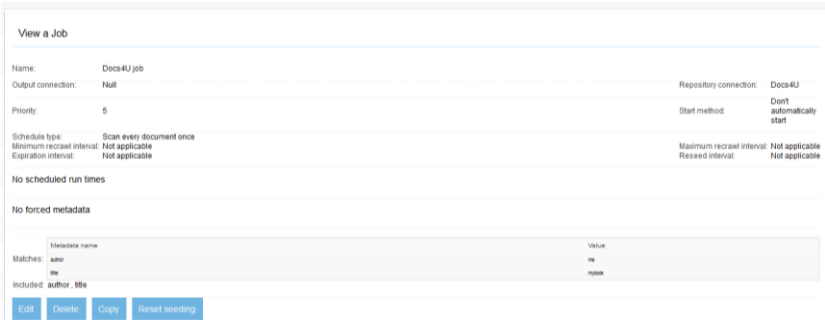


Figure 7.10 A Docs4U job view screen.

Since it all seems to work, we’re ready to start working on the final bit of code – the actual methods that do the crawling!

7.3.4 Implementing the IRepositoryConnector crawling methods

The crawling methods of IRepositoryConnector consist of addSeedDocuments(), getDocumentVersions(), processDocuments(), and sometimes releaseDocumentVersions(). The first method adds the initial seed documents to the job queue. The second method obtains version strings for a set of documents. The third method extracts document references and/or indexes a set of documents. The last method is used to free any resources that were reserved earlier by the getDocumentVersions() method.

We’ve already described the phases of crawling back in Chapter 1. The connector methods above support those phases, as we’ll discuss in more detail in Chapter 12. But we don’t need to know or care about any of that while we write our connector. In fact, all we need to know about right now is what the methods do, and what the connector writer’s responsibilities are, which were broadly specified in table 7.1. Let’s clarify those responsibilities, and write the code.

THE ADDSEEDDOCUMENTS() METHOD

This method is the easiest of the three major crawling methods to write. All it has to do is process the document specification it is given, find the corresponding document identifiers, and add those document identifiers to the job queue. But how does it do that?

The addSeedDocuments() method receives a parameter of type ISeedingActivity. We’ve described how these activity objects provide framework functionality. Table 7.4 describes that functionality in detail.

Table 7.4 ISeedingActivity methods and their meanings.

Method	Meaning
--------	---------

<code>addSeedDocument()</code>	Add a document identifier as a seed to the job queue.
<code>addUnqueuedSeedDocument()</code>	Add a document identifier as a seed to the job queue, but don't indicate that it needs to be processed.
<code>recordActivity()</code>	Record an activity as a history record.
<code>checkJobStillActive()</code>	See if the current job is active, and throw an appropriate exception if not. Convenience method to improve ability to abort a job promptly.
<code>createGlobalString()</code>	Turn a simple string into a global event name.
<code>createConnectionSpecificString()</code>	Turn a simple string into a connection-specific event name.
<code>createJobSpecificString()</code>	Turn a simple string into a job-specific event name.

We're particularly interested in the `addSeedDocument()` method above; that looks like just what we need to write our Docs4U connector `addSeedDocuments()` method. See listing 7.11.

Listing 7.11 The `addSeedDocuments()` implementation for the Docs4U connector

```

public void addSeedDocuments(ISeedingActivity activities,
    DocumentSpecification spec,
    long startTime, long endTime, int jobMode)
    throws ManifoldCFException, ServiceInterruption
{
    Docs4UAPI currentSession = getSession();           #1
    int i = 0;                                         #2
    while (i < spec.getChildCount())                  #2
    {
        SpecificationNode sn = spec.getChild(i++);    #3
        if (sn.getType().equals(NODE_FIND_PARAMETER)) #3
        {
            String findParameterName = sn.getAttributeValue(ATTRIBUTE_NAME);
            String findParameterValue = sn.getAttributeValue(ATTRIBUTE_VALUE);
            Map findMap = new HashMap();               #4
            findMap.put(findParameterName, findParameterValue); #4
            try
            {
                if (Logging.connectors.isDebugEnabled()) #5
                    Logging.connectors.debug("Docs4U: Finding documents where "+ #5
                        findParameterName+"="+ findParameterValue+""); #5
                D4UDocumentIterator iter = currentSession.findDocuments( #5
                    new Long(startTime), new Long(endTime), findMap); #5
                while (iter.hasNext())
                {
                    String docID = iter.getNext();
                    activities.addSeedDocument(docID); #6
                }
            }
        }
    }
}

```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

document's version string will change. As long as the version string cannot completely revert back to a previous value, ManifoldCF will eventually catch up to reality.

Another way of minimizing the problem is to encode meaningful information into the version string, which the `processDocuments()` method can use both to save work and to keep the indexed document and version string in as close a correlation as possible. For example, on some connectors the access tokens for a document must be obtained at `getDocumentVersions()` time and incorporated into the version string, because that is the only way to detect changes in the document's security configuration. It makes sense, then, to unpack the access tokens from the version string at `processDocuments()` time, rather than fetch them again. This is a common technique. We will use this technique in the Docs4U connector in order to track changes to the specified metadata names to include in the indexing. The version string we construct will thus include these metadata names, as well as the modification timestamp of the document.

There are also plenty of other things your connector can do within `getDocumentVersions()`. The method receives an `IVersionActivity` object, which has the key methods listed in table 7.5.

Table 7.5 Key `IVersionActivity` methods and their meanings.

Method	Meaning
<code>retrieveParentData()</code>	Retrieves specified parent carry-down data as Strings.
<code>retrieveParentDataAsFiles()</code>	Retrieves specified parent carry-down data as temporary files.
<code>checkJobStillActive()</code>	See if the current job is active, and throw an appropriate exception if not. Convenience method to improve ability to abort a job promptly.
<code>beginEventSequence()</code>	Begins a specified event sequence, or returns false if the sequence is already underway by another thread.
<code>completeEventSequence()</code>	Completes the specified event sequence.
<code>retryDocumentProcessing()</code>	Marks the specified document as needing to be retried later due to a blocking event sequence.
<code>checkMimeTypeIndexable()</code>	Checks to see if the mime type specified is indexable by means of the current output connector.
<code>checkDocumentIndexable()</code>	Checks to see if the document data specified is indexable by means of the current output connector.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

In addition to the methods listed above, `IVersionActivity` also has some methods in common with `ISeedingActivity` that I have not repeated, such as `recordActivity()` and the event name generation methods. You may want to look at the complete list in the `ManifoldCF Javadoc`.

For `Docs4U`, as we already determined early in the chapter, we need none of these tricks. Thus, `getDocumentVersions()` can be simple and straightforward. The related method, `releaseDocumentVersions()`, will also be simple, since there are no temporary files created by `getDocumentVersions()`. See listing 7.12.

Listing 7.12 The `getDocumentVersions()` method implementation for the `Docs4U` connector

```
public String[] getDocumentVersions(String[] documentIdentifiers,
    String[] oldVersions, IVersionActivity activities,
    DocumentSpecification spec, int jobMode, boolean usesDefaultAuthority)
    throws ManifoldCFException, ServiceInterruption
{
    List<String> metadataNames = new ArrayList<String>();           #1
    int i = 0;                                                     #2
    while (i < spec.getChildCount())                               #2
    {
        SpecificationNode sn = spec.getChild(i++);
        if (sn.getType().equals(NODE_INCLUDED_METADATA))          #3
            metadataNames.add(sn.getAttributeValue(ATTRIBUTE_NAME)); #3
    }
    String[] namesToVersion = metadataNames.toArray(new String[0]); #4
    java.util.Arrays.sort(namesToVersion);                         #4

    Docs4UAPI currentSession = getSession();                       #5
    String[] rval = new String[documentIdentifiers.length];        #6
    try                                                             #7
    {                                                                #7
        int i = 0;                                                 #8
        while (i < documentIdentifiers.length)                    #8
        {
            if (Logging.connectors.isDebugEnabled())              #9
                Logging.connectors.debug("Docs4U: Getting update time for '"+ #9
                    documentIdentifiers[i]+"");                   #9
            Long time = currentSession.getDocumentUpdatedTime( #9
                documentIdentifiers[i]);                           #9
            if (time == null)                                       #10
                rval[i] = null;                                     #10
            else
            {
                StringBuilder versionBuffer = new StringBuilder();
                packList(versionBuffer, namesToVersion, '+');      #11
                versionBuffer.append(time.toString());             #12
                rval[i] = versionBuffer.toString();
            }
            i++;
        }
        return rval;                                              #13
    }
}
```

```

    catch (InterruptedException e)
    {
        throw new ManifoldCFException(
            e.getMessage(), e, ManifoldCFException.INTERRUPTED);
    }
    catch (D4UEException e)
    {
        Logging.connectors.warn("Docs4U: Error versioning documents: "+
            e.getMessage(), e);
        throw new ManifoldCFException(e.getMessage(), e);
    }
}

```

#1 Accumulate a list of metadata names
#2 Examine the document specification child nodes
#3 Add all metadata names to the list
#4 Convert to an array, and sort the array
#5 Get or create a Docs4U session
#6 Create the return array for the document versions
#7 Catch any Docs4U exceptions
#8 Loop through all the document identifiers
#9 Get the document's updated time, and log
#10 If document no longer exists, the return version is null
#11 Pack the metadata names into the version string
#12 Add the timestamp to the version string
#13 Return the array of version strings

At #1, we start the process of building a sorted list of version names. It must be sorted, since the resulting string will be used for version comparisons. We examine all the document specification child nodes at #2, add the specified metadata names to the list at #3, and sort the results at #4. Now we're ready to create the Docs4U session at #5, and create the version string return array at #6. We catch all Docs4U exceptions starting at #7.

Looping through all the document identifiers at #8, we get each document's updated time at #9. If the document no longer exists, we return a version string of null for it, at #10. Otherwise, we pack the sorted metadata names into the version string at #11, and add the timestamp to the version string at #12. We return the entire array at #13.

THE PROCESSDOCUMENTS() METHOD

The `processDocuments()` method does the actual crawling. It has the responsibility of fetching documents, extracting document references from these, and indexing the documents if appropriate. These requirements are supported by the `IProcessActivity` object, which is passed to `processDocuments()`, which provides the functionality as described in table 7.6.

Table 7.6 Key `IProcessActivity` methods and their meanings.

Method	Meaning
<code>addDocumentReference()</code>	Add a document reference to the job queue.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

<code>recordDocument()</code>	Record a document's version, but don't index it.
<code>ingestDocument()</code>	Index a document and record its version, using the current output connector.
<code>deleteDocument()</code>	Delete a current document from the index, via the current output connector.
<code>setDocumentScheduleBounds()</code>	Set the time parameters for when the specified document will be re-evaluated (under continuous crawling conditions).
<code>setDocumentOriginationTime()</code>	Set the specified document's time of origin (which is used for calculating document expiration).

The `IProcessActivity` method list is, again, incomplete in that I have not repeated methods that were described for `ISeedingActivity` and `IVersionActivity`, such as `retrieveParentData()`. Please refer to the ManifoldCF Javadoc for a complete list.

The `processDocuments()` method for the Docs4U connector will need to index documents, using the `ingestDocument()` `IProcessActivity` method, but there is no reference extraction necessary, since that's not how the Docs4U repository works. It will, however, need to generate a history record for each document fetched from the repository, as we previously discussed. The method to do that is `recordActivity()`, which you will see used in listing 7.13.

Listing 7.13 The Docs4U implementation of `processDocuments()`

```

public void processDocuments(String[] documentIdentifiers,
    String[] versions, IProcessActivity activities,
    DocumentSpecification spec, boolean[] scanOnly, int jobMode)
    throws ManifoldCFException, ServiceInterruption
{
    Docs4UAPI currentSession = getSession();           #1
    try                                                 #2
    {                                                  #2
        for (int i = 0; i < documentIdentifiers.length; i++) #3
        {
            if (!scanOnly[i])                          #4
            {
                String docID = documentIdentifiers[i];   #5
                String version = versions[i];            #5

                long startTime = System.currentTimeMillis(); #6
                long dataSize = 0L;                      #6
                String status = "OK";                    #6
                String description = null;               #6
                boolean fetchOccurred = false;           #6

                D4UDocInfo docData = D4UFactory.makeDocInfo(); #7
                try                                       #7
                {

```

```

String url = currentSession.getDocumentURL(docID);           #8
if (url == null ||                                           #9
    currentSession.getDocument(docID,docData) == false)      #9
{
    activities.deleteDocument(docID);                         #10
}
else
{
    fetchOccurred = true;
    RepositoryDocument rd = new RepositoryDocument();        #11
    InputStream is = docData.readData();                      #12
    if (is != null)
    {
        try
        {
            dataSize = docData.readDataLength().longValue(); #13
            rd.setBinary(is,dataSize);                         #14

            List<String> metadataNames = new ArrayList<String>(); #15
            unpackList(metadataNames,version,0,'+');           #15
            for (String metadataName : metadataNames)          #16
            {
                String[] metadataValues =                      #17
                    docData.getMetadata(metadataName);         #17
                if (metadataValues != null)
                {
                    rd.addField(metadataName,metadataValues);  #18
                }
            }

            rd.setSecurityACL(                                  #19
                RepositoryDocument.SECURITY_TYPE_DOCUMENT,    #19
                docData.getAllowed());                         #19
            String[] disallowed = docData.getDisallowed();     #19
            List<String> list = new ArrayList<String>();        #19
            list.add(GLOBAL_DENY_TOKEN);                       #19
            for (String disallowedToken : disallowed)           #19
            {
                list.add(disallowedToken);                     #19
            }                                                    #19
            rd.setSecurityDenyACL(                              #19
                RepositoryDocument.SECURITY_TYPE_DOCUMENT,    #19
                list.toArray(disallowed));                     #19

            activities.ingestDocument(docID,version,url,rd);    #20
        }
        finally
        {
            is.close();                                         #21
        }
    }
}
}
catch (D4UException e)
{

```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

```

        status = "ERROR";
        description = e.getMessage();
        throw e;
    }
    finally
    {
        docData.close(); #22
        if (fetchOccurred)
            activities.recordActivity(new Long(startTime), #23
                ACTIVITY_FETCH, #23
                dataSize, docID, status, description, null); #23
    }
}
}
}
catch (InterruptedException e)
{
    throw new ManifoldCFException(
        e.getMessage(), e, ManifoldCFException.INTERRUPTED);
}
catch (IOException e)
{
    Logging.connectors.warn("Docs4U: Error transferring files: "+
        e.getMessage(), e);
    throw new ManifoldCFException(e.getMessage(), e);
}
catch (InterruptedException e)
{
    throw new ManifoldCFException(
        e.getMessage(), e, ManifoldCFException.INTERRUPTED);
}
catch (D4UException e)
{
    Logging.connectors.warn("Docs4U: Error getting documents: "+
        e.getMessage(), e);
    throw new ManifoldCFException(e.getMessage(), e);
}
}
}

```

- #1 Create a Docs4U session**
- #2 Catch any Docs4U exceptions**
- #3 Loop through all document identifiers**
- #4 Only index if we're not just scanning for new identifiers**
- #5 Get a single document's ID and version**
- #6 Initialize data for recording the fetch activity**
- #7 Create the Docs4U document object, and start a try/finally**
- #8 Get the document's URL**
- #9 Get the document information**
- #10 If the document doesn't exist, delete it from index**
- #11 Create RepositoryDocument document structure**
- #12 Get the document contents as a stream**
- #13 Get the document length in bytes**
- #14 Set the content stream into the RepositoryDocument**
- #15 Unpack metadata names from version string**
- #16 Cycle through the metadata names**
- #17 Get the corresponding metadata values**

#18 Add values to RepositoryDocument object
#19 Transfer security information to RepositoryDocument
#20 Index the document
#21 Close the input stream, as per contract
#22 Close the Docs4U document info object, as per contract
#23 Record the activity

At #1, we create a Docs4U session, and catch any Docs4U exceptions at #2. At #3 we loop through all the passed-in document identifiers. Since this repository does not have any references within content, we do nothing if the `scanOnly` flag for the document is `true`, at #4, but if it is `false`, we prepare to index a document, at #5. We initialize the data we'll need to record the fetch activity at #6. Then we fetch the document at #7, and start a try/finally block for cleaning up the Docs4U document object. At #8 we get the document's URL, and at #9 we get the rest of the document information we need. If the document no longer exists, we signal its deletion from the index at #10.

To index the document, at #11 we create a `RepositoryDocument` object. We obtain the document's content input stream at #12, and the length at #13, and add these to the `RepositoryDocument` object at #14. Then, for metadata, we unpack the version string at #15, and cycle through the metadata names at #16. We obtain the metadata values at #17, and also apply these to the `RepositoryDocument` object at #18. At #19, we transfer the security information as well, making sure to include a global deny token as part of the documents deny tokens. We'll talk more about that in Chapter 8. Finally, at #20, we do the index operation.

Cleaning up involves closing the input stream, at #21, as per the Docs4U contract, and closing the Docs4U document info object, at #22. Finally, when we are all done, we record the activity information, at #23.

UNDERSTANDING WHEN TO THROW `SERVICEINTERRUPTION` EXCEPTIONS

All the crawling methods have the potential to throw `ServiceInterruption` exceptions. But we've never really explained in depth how these exceptions work.

The purpose of a `ServiceInterruption` exception is to signal to ManifoldCF that a problem occurred with the operation that was of a transient nature. By "transient", I mean that it is possible for the operation to succeed if it is simply retried. This is different from throwing a `ManifoldCFException`, which designates a permanent problem which will not typically resolve with retries.

It is sometimes very challenging to know whether a given error should be classified one way or another. While the right decision is easy for some kinds of errors, it is often not possible to describe a proper course of action for all kinds. For these kinds of errors that live in a gray area, the right way to address them is usually to throw `ServiceInterruption` exceptions, but with some caveats that we'll get to in a moment.

The `ServiceInterruption` exception not only signals ManifoldCF that retries should be attempted, it actually gives the connector a chance to tell ManifoldCF how often to retry, how

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

long to retry before giving up, and, if giving up is necessary, whether to abort the job should that occur. For example, see listing 7.14.

Listing 7.14 Throwing a `ServiceInterruptedException`

```
Exception e = ...;
String docID = ...;
long currentTime = System.currentTimeMillis();
throw new ServiceInterruptedException("Repository was down reading '"+docID+
    "'", retrying: "+e.getMessage(),e,
    currentTime + 300000L,
    currentTime + 6L * 60L * 60000L,
    -1,
    true);
#A Include the original exception
#B Retry every 5 minutes
#C Retry up to 6 hours max
#D No retry count limit
#E Abort job when exceeded
```

All of this flexibility gives the connector writer a lot of responsibility to carefully think through the error cases that might occur. Failure to treat error cases appropriately is the number one reason why connectors run into problems in the field – but obviously there’s no perfect answer for how to handle some error conditions, so you may be forced to largely “wing it”. If you really don’t know what to do with an error case, my suggestion is to start with the following, which should not get you into too much trouble:

- If in doubt, throw a `ServiceInterruptedException` exception rather than a `ManifoldCFException`
- Have it retry every five minutes, up to six hours maximum
- Tell it to abort the job once the maximum is exceeded

Docs4U is unusual. Because it is completely file-system based, it doesn’t really have cases where `ServiceInterruptedException` exceptions should be thrown. So our repository connector is now complete. It’s time to try it out!

TRYING OUT THE CONNECTOR

As before, build the connector jar and copy it, plus the Docs4U jar, to the `connector-lib` folder under the ManifoldCF directory `dist/example`. Then, start ManifoldCF. We can use the Docs4U connection definition that’s already in existence from earlier in the chapter, and use the `Null` output connection definition we created back in Chapter 1. We can even use the Docs4U job we created to try out the UI components earlier, but we’ll have to change it, to make it look like figure 7.11.

View a Job

Name:

Docs4U job

Output connection:

Null

Repository connection:

Docs4U

Priority:

5

Start method:

Don't automatically start

Schedule type:

Scan every document once

Minimum recrawl interval:

Not applicable

Maximum recrawl interval:

Not applicable

Expiration interval:

Not applicable

Re-assign interval:

Not applicable

No scheduled run times

No forced metadata

Matches:

Metadata name

Value

Included:

author, title

excludes

Edit

Details

Copy

Reset settings

Figure 7.11 The sample job we'll use to try out the Docs4U connector.

Now, let's go to the job status page and click the `Start` button. After some refreshing, we see something like figure 7.12.

Status of Jobs							
	Name	Status	Start Time	End Time	Documents	Active	Processed
Start Start manual	Docu4U job	Done	Mon Apr 28 13:22:10 EDT 2014	Mon Apr 28 13:23:11 EDT 2014	3	0	3
Refresh							

Figure 7.12 A completed Docs4U crawl against the sample repository.

The crawl completed, but what did it do? Let's get a simple history report, so we can see what happened. We'll also get to see if our activity logging code worked. See figure 7.13.

Simple History Report

Connection:

-- Not specified --
[Select a connection](#)

Start Time:

10 am

11 am

12 am

22

23

0

February

March

April

26th

27th

28th

2012

2013

2014

End time:

Not specified

Not specified

Not specified

12 am

0

1

January

February

March

1st

2nd

3rd

2005

2006

2007

Entity match:

Result code match:

Go

Start Time

Activity

User/Identifier

Result Code

Bytes

Time

Result Description

24-08-2014 11:02:00:00	account registration (init)	USP070887C2C000000000	OK	0	1	
24-08-2014 11:02:00:00	per email		OK	0	1	
24-08-2014 11:02:00:01	account register (init)	USP070887C2C000000000	OK	62	1	
24-08-2014 11:02:00:01	account register (init)	USP070887C2C000000000	OK	61	1	
24-08-2014 11:02:00:01	account register (init)	USP070887C2C000000000	OK	61	1	
24-08-2014 11:02:00:01	account register (init)	USP070887C2C000000000	OK	61	1	
24-08-2014 11:02:00:07	Not	?	OK	61	1	0007
24-08-2014 11:02:00:07	Not	?	OK	61	1	0008
24-08-2014 11:02:00:08	per email		OK	0	1	

Previous Page

Page 1 of 1

Next Page

Rows: 6/6/60

Rows per page: 20

Figure 7.13 A simple history report of a crawl of the sample Docs4U repository.

So it appears that three documents were indeed fetched and indexed. But one of them took 17 seconds to grab! Is something wrong?

There **is** something wrong – but the problem is beyond the ability of the connector to solve. Docs4U is, after all, just an example. It resolves concurrent access situations with a back-off-and-retry strategy. The back off time is a random number between zero and sixty seconds, so any contention can cause large delays. The connector is working fine, but the repository is behaving in an ugly manner.

Nevertheless, this is quite a milestone! Your first connector is up and running, and seems to be crawling properly! I leave it as an exercise for the student to verify that the incremental features of the connector work properly as well, and that the connector recognizes changes to the metadata specification properly too.

7.3.5 The `requestInfo()` method

The only method we've not yet implemented from `IRepositoryConnector` is the `requestInfo()` method, and yet our connector seems to work without it. What does it do?

The purpose of the `requestInfo()` method is to provide a general hook for the ManifoldCF API to access repository information, so as to allow people to write their own UI's for ManifoldCF and its connector suite using the API Service. It is therefore in some sense optional, although we will provide an implementation for the Docs4U connector in order to learn how it's done.

DESIGNING THE URL SPACE

If you recall Chapter 3, the ManifoldCF API is RESTful. This means that every piece of data that can be retrieved by the API user must have a URL, since it is considered to be a resource. What resources should we allow access to for the Docs4U repository?

The only data that the Docs4U connector's Crawler UI methods make use of is the list of metadata names. It seems reasonable to expect anyone who is creating a UI to need the same information. So we'll assume that all we need to do is provide access to the list of metadata names, and we'll be done.

The `requestInfo()` method receives an output `Configuration` object, that is meant to be filled in, plus a command, which is what is left after the leading portions of the URL are removed that describe the specific repository connection definition. Since we need only metadata names right now, but may conceivably add more functionality in the future, let's reserve the command `metadata` to mean the list of metadata names. Then, since we already coded a method to communicate with the repository and get this list, we can just write the method. See listing 7.15.

Listing 7.15 The Docs4U connector implementation of `requestInfo()`

```
public boolean requestInfo(Configuration output, String command)
    throws ManifoldCFException
{
    if (command.equals("metadata")) #1
    {                               #1
```

```

try                                                    #2
{                                                       #2
    String[] metadataNames = getMetadataNames();      #3
    for (String metadataName : metadataNames)         #4
    {                                                  #4
        ConfigurationNode node = new ConfigurationNode("metadata"); #5
        ConfigurationNode child = new ConfigurationNode("name");    #5
        child.setValue(metadataName);                 #5
        node.addChild(node.getChildCount(), child);    #5
        output.addChild(output.getChildCount(), node); #6
    }
}
catch (ServiceInterruptedException)                 #7
{
    ManifoldCF.createServiceInterruptedException(output,e);
}
catch (ManifoldCFException e)                        #8
{
    ManifoldCF.createErrorNode(output,e);
}
else                                                  #9
    return super.requestInfo(output,command);
return true;
}

```

#1 Check for a known command
#2 Catch any repository errors
#3 Get the metadata names
#4 Loop over all the names
#5 For each name, build a node with a child node that contains the data
#6 Add the new node to the output response
#7 If a service interruption takes place, build an appropriate response
#8 If a harder error takes place, build a different appropriate response
#9 We don't recognize the command: ask the superclass.

At #1, we look for the command. If we find one we know, we enter a try/catch block at #2 to capture any errors thrown by the repository communication, because we want to return these as JSON. At #3, we obtain the metadata names from the repository, using the same method the Crawler UI tabs used. At #4, we loop over the returned names, building an appropriate node structure at #5. We add the new node to the output at #6. If we got a `ServiceInterruptedException` exception, then at #7 we encode a special response indicating that. Similarly, a `ManifoldCFException` generates an appropriate response at #8. If we didn't find a command we recognized, we send it on to the superclass at #9.

TRYING IT OUT

After you rebuild the Docs4U connector jar, deploy it, and restart ManifoldCF, it should be possible for the API Service to respond to the metadata names request appropriately. Let's give it a try, using `curl` like we used to in Chapter 3.

```

curl "http://localhost:8345/mcf-api-
service/json/info/repositoryconnections/Docs4U/metadata"
{"metadata":[{"name":"author"}, {"name":"date"}, {"name":"title"}]}

```

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

That looks great! It looks like we are done with our first repository connector! But should we declare victory, pack up, and go home?

Experienced software developers are probably smiling knowingly right about now. They know that it is one thing to test software by hand in a controlled environment, and yet another to challenge that software with the myriad environmental conditions it might find itself in one day. That's what we're going to talk about next.

7.4 *Debugging*

There is an art to debugging. It is simultaneously the most rewarding and the most frustrating aspect of programming, and there are no two people who go about it in precisely the same way. In this section, however, we'll explore some ideas for ways you can debug your repository connector that I hope will be helpful to you. But first, to get you in the proper frame of mind, I'd like explore a few case studies. My goal here is to help expand your horizons beyond mere code and into systems, because that's where all the really interesting problems arise.

7.4.1 *What could possibly go wrong?*

It's one thing to write your own connector, and get it working in your own controlled environment. It's a whole different experience moving such a connector into the varied situations it may encounter in the field. Subtle differences in system architecture, network architecture, software version, or configuration can sometimes have profound effects on the behavior of some content repositories.

This means that it is very likely that you will one day find a repository installation for which your connector won't work, no matter how carefully you test it yourself. If you've taken any shortcuts in the design and implementation, this day will come sooner rather than later. But rest assured, it will come.

BEWARE THE SERVER-SIDE PLUG-IN

This case study shows the benefit of examining what is happening on the repository server. If there are problems with your connector, it will not always be the case that you have made a coding error, although you certainly have a responsibility to rule that out as best you can before assuming otherwise.

Developing the ManifoldCF LiveLink connector was easy. But after the connector was deployed at a large company, the problems began. The first symptom was that the connector kept aborting with random errors that were coming from the LiveLink API. At first, these errors seemed like they could be explained by poor network connectivity, so I dutifully added code to the connector to make sure it properly threw the appropriate `ServiceInterruption` exception for all places where I thought network reliability issues might cause an error, and redeployed the connector.

At first, it seemed like I'd fixed the problem, but then even stranger stuff began to happen. A second symptom now appeared: the connector seemed to operate very slowly in

the company's environment. And, despite all my changes, really strange errors that could not possibly be due to network issues began to shut the jobs down.

The really confusing part of it was that I could not reproduce any of these problems on a test version of LiveLink, no matter how much I kicked it around. The connector was capable of crawling 50,000 LiveLink documents in 15 minutes on the test machine. And yet, in the field, the same number of documents was taking days.

It was at this point that I began to get daily calls. They were always the same – demanding to know when I could fix the connector. I asked for every diagnostic aid I could think of for the connector itself, including the connector debug log output. The logs confirmed a picture where the connector was doing fine for a while, but soon entered a state where most document fetches failed, and there were a ridiculously large number of `ServiceInterruption` exceptions and retries. But I still had no idea why this was happening.

In desperation I asked for the LiveLink server side logs, but – and this was the first clue – my tormentors did not know how to obtain them or interpret them. When it finally was made clear that I had no hope of solving the problem without these logs, they arranged to allow me to talk with the company's LiveLink administrator.

It only took fifteen minutes on the phone to solve the problem. The issue was that the LiveLink server was leaking memory, or, rather, a third-party plug-in that LiveLink was calling was leaking memory. The administrator could see this, because LiveLink was writing intermittent out-of-memory errors to its logs after it had been running for a time. As time went on and memory got tight, pretty near every transaction would fail due to memory issues. This explained the problem perfectly. The solution the administrator arrived at was to restart LiveLink every twenty-four hours, which seemed a little draconian, but it worked. And, because I'd already added the extra robustness to the connector in the face of connection difficulties, even a restart of LiveLink no longer would cause LiveLink connector jobs to abort anymore.

INFINITELY MIS-CONFIGURABLE

I include the following case study because it shows how subtle problems can have a wide variety of potential causes, and thus the connector writer must have quite a number of tricks up her sleeve. It also shows that one really cannot afford to be shy, and must be willing to make a fool out one's self for a good cause.

The story involves another strange intermittent symptom. This symptom occurred using the jCIFS connector, crawling documents on a company-wide shared file system. In this case, the job would run for many hours, but would eventually abort. Turning on connectors debug and examining the logs showed that certain documents, connected in no obvious way by path, disk, or security, were causing the connector to throw a `ServiceInterruption` over and over until eventually the conditions of the `ServiceInterruption` were being

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

exceeded. The job would abort the first time one of these special documents made it all the way through this process.

Looking at the documents themselves, the only thing that really seemed related about them was their length. They were all over a megabyte. But why should that matter?

And then it dawned on me that this could be the crucial factor. jCIFS has a number of internal timeouts. If a transaction with the shared filesystem takes too long, I figured that one of those timeouts might be being exceeded. I began to suspect that a network switch might be involved. You see, the installation had many specialized subnets configured, and the connector was being used on one subnet to crawl documents that were saved on another subnet. Between them was a network switch – and network switches these days are almost sentient. They can be configured to do all sorts of load shaping. Perhaps this switch was drastically slowing down traffic after a certain point.

Unfortunately, this hunch turned out to be wrong. Much to my embarrassment, a network administrator could find no switch configuration which might have lead to the effect I was seeing. The only recourse remaining was to obtain a packet capture.

Packet captures can be extraordinarily useful in many circumstances. You can gather one of these using several different packages, such as tcpdump on Linux and WireShark on Windows. If so configured, all packets on the local subnet can be captured, not just those from the computers involved in the transactions you are interested in. Thus, they can represent a huge security risk for enterprises, and I typically use them only as a last resort.

The packet capture showed something very odd. Timeout errors were occurring, sure enough, but the errors were not occurring only on the large files! They were occurring on all sizes of file, fairly randomly. But how could this lead to the symptom I was seeing?

It turned out that this was the fault of the jCIFS library. This library has an interesting way of handling timeout errors. Upon timeout, it presumes that the underlying connection to the server is defunct, and closes it. Thus, since the CIFS protocol permits the same connection to be shared for several simultaneous transactions, **all** of the active transactions sharing the connection wind up getting aborted when any single one of them takes too long.

So this was the explanation! All of this carnage was due to nothing more than a slow or overburdened network file system. If the file system did not respond to a certain transaction within the timeout window, jCIFS would forcibly interrupt all other operations ongoing with that server at that time. Any such files being transferred would abort with an error condition also. Because this happened frequently enough, large files would simply never manage to make it over in one piece, and they'd eventually cause the connector to abort.

Once the analysis was done, the solution was obvious. I simply increased the default jCIFS timeouts to values more consistent with what I was seeing from the field.

As you can see from both of these studies, bringing the right tool to bear is a critical part of the connector debugging process. We'll talk about that next.

7.4.2 Tools and techniques

The bag of tools available for connector debugging varies considerably from connector to connector. Some tools, like the ManifoldCF log, never go out of style, while other tools, such as Apache Axis SOAP debugging are only of any use when SOAP is involved. I've summarized the most useful tools, from my perspective, for currently existing connectors in the ManifoldCF stable in table 7.7. I'll go on to talk about some of these tools in more depth afterwards.

Table 7.7 The best tools for debugging current ManifoldCF connectors

Connector	Tools
Documentum (EMC)	History, connector logs, webtop, DQL, server-side logs
FileNet (IBM)	History, connector logs, server-side logs
File system	History, connector logs
jCIFS (Microsoft)	History, connector logs, packet capture
JDBC (Oracle etc.)	History, connector logs, SQLPlus (or the equivalent)
LiveLink (OpenText)	History, connector logs, server-side logs
Meridio (Autonomy)	History, connector logs, web service logs, Windows event logs
SharePoint (Microsoft)	History, connector logs, web service logs, Windows event logs
RSS, Web	History, connector logs, browser http logging

I've arranged these tools in order of priority for each of the connectors currently included with ManifoldCF. But you can clearly generalize the pattern to whatever connector you plan on writing. For instance, I'd recommend in all cases that you start by looking at the Simple History report for any crawl that is failing. This is the easiest thing to do, and often is sufficient to find the underlying issue, provided the connector writer thought to add the appropriate activities to the connector in the first place. By now, you should know how to use the Simple History report quite well, since we've visited it repeatedly in the early chapters of the book.

Note You may have noticed that I did not mention any popular IDEs, such as Eclipse or Intelli-J, in this list. That is because I am assuming you already know how to perform basic code-level debugging in Java, with whatever tools you know and love. This section is about how to debug more subtle problems. If you need help figuring out basic Java code execution problems, there are a number of online resources available, and many books.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

CONNECTOR LOGGING

If the Simple History report isn't enough to determine what the problem is, it's time to move on to using the connector logging output. By this I also mean turning including the connector debugging property in the `properties.xml` file, as follows:

```
<property name="org.apache.manifoldcf.connectors" value="DEBUG"/>
```

I strongly recommend making sure your log format, as it is set up in the `logging.ini` file, includes time and date information in all log entries to the nearest millisecond. This can be done by simply adding this line to the standard example `logging.ini`:

```
log4j.appender.MAIN.layout.conversionPattern = %d [%t] %-5p %c- %m%n
```

After you restart ManifoldCF, the logging changes will take effect, and you should see output to your designated log file. The default log file for the Quick Start example is `dist/example/logs/manifoldcf.log`. You can watch what happens in this file by using a utility such as `tail`, or you can wait until you are done generating output and then bring the whole file up into your favorite editor.

In order to make sense of the logging output, you will probably need to be looking at the connector code as you examine the results. Only then will it become obvious what the outputs really mean. The good thing about open source is that it is possible to do this; the bad thing is that you will often need to understand someone else's code.

Indeed, you may feel you are at the mercy of the connector writer in more ways than that. It may well be the case that you are missing some key logging output that would help you figure out the problem. But I encourage you not to be afraid to add the code you need. You may even want to submit your logging additions to the ManifoldCF project as improvements.

But using the logs is not always enough to solve every problem. We must sometimes look at the next level down, and that is going to differ from connector to connector. We'll start with the most generally useful, and move on from there.

WEB SERVICES LOGGING

For connectors which use web services with SOAP as their primary communication protocol, it is often very useful to examine the request and response XML as it goes by. If such a connector uses Apache Axis for its web service implementation, as the SharePoint and Meridio connectors do, then it is possible to turn on specific loggers within Axis that make it possible to capture these transactions. All you need to do to enable these loggers is to add one or both of the following line to the `logging.ini` file, and restart ManifoldCF:

```
log4j.logger.org.apache.http=DEBUG
log4j.logger.org.apache.http.wire=DEBUG
```

The first line logs the HTTP transactions that are taking place, including the HTTP headers and HTTP result codes. The second logs the contents of the request and response, in raw form. This becomes essential if the protocol used is not just HTTP but rather HTTPS. Packet captures will be useless for the latter, because the contents will be encrypted.

In order for this tool to work, you not only need to be using Apache Axis, but you also need to be using `HttpComponents HttpClient` for the HTTP transport mechanism. This is the recommended approach for web services regardless, because Axis' built-in HTTP transport

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

does not have anywhere near the configurability of `HttpComponents HttpClient`, and, as of this writing, does not support NTLM either, which limits its ability to interact with a Windows-based web service. If this is something you find that you need to do, I recommend looking at the code for the SharePoint and Meridio connectors, which both use this technology.

PACKET CAPTURE

For some technologies that use well-known protocols, packet capture and examination is a critical tool for debugging connector/repository interactions. Open-source tools that capture packets and interpret them visually can allow you to see the interactions between computers at the network level.

A packet capture works by means of a computer's network adapter. The capture software listens in to all network packets that the adapter sees going by, and records the ones you specify to a disk file. Then, later, this packet dump file can be opened by a software tool for inspection. A good tool will then identify each packet and decode the bytes within to the extent it can.

The two tools I am most familiar with are the Linux utility `tcpdump` and the Windows program called WireShark. Both are freely available and complement each other well.

The `tcpdump` utility is used for obtaining captures on Linux machines. It has some ability to filter packets based on various criteria, and (for example) only capture packets whose source or destination is a given IP address. This is important because, as you might imagine, the volume of network traffic passing any given network adapter is pretty high.

WireShark is a Windows program that has the ability to view and interpret packets from a packet dump file. It also has the ability to capture packets on a Windows machine. The list of protocols it understands is impressively long – everything from TCP/IP to HTTP to CIFS are in its repertoire.

Packet captures are useful because they help you understand the interaction that is going awry. A certain sequence of packet exchanges takes place for a successful interaction, and a different sequence takes place for an unsuccessful one. It is thus essential for the person doing the debugging to know what a correct interaction should look like! This often means getting a capture of a correct interaction, using a client program that already communicates properly with the repository in question using the same parameters and credentials.

SERVER-SIDE LOGS

I can only speak to the importance of server-side logs, not to any details as to how to find them. For any given repository, these will be in different places, accessible by different tools. But in all cases, the techniques are similar.

First, look for errors. The server-side logs may well be recording very basic problems with your connector's usage of a third-party library, for instance. Or, it may be recording system-wide problems such as disk errors accessing files.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=727>

Second, try to correlate what is happening in the server-side logs with what is happening in the connector logs. If both logs have time stamps, this should be straightforward. You may notice patterns related to the problem you are trying to fix.

BROWSER-BASED HTTP LOGGING

In some cases a connector needs to simulate a correct interaction with a repository which takes place through a browser, such as FireFox. This is especially helpful in understanding session-based login sequences for configuring the Web connector.

You might be able to learn something about what is going on by using packet capture, but if the HTTP interaction uses secure sockets (e.g. HTTPS), then a packet capture will be pretty useless. But, luckily, many browsers support plug-ins. Some plug-ins allow a user to inspect the details of the HTTP interaction with the server. On Mozilla FireFox, I've used the plugin called "Live HTTP Headers" for this purpose and found it to be easy to use and install, and very helpful. The URL for this plugin is <https://addons.mozilla.org/en-us/firefox/addon/live-http-headers>.

7.5 Summary

In this chapter, we've designed and implemented a ManifoldCF repository connector. We learned the critical design decisions that need to be made up front in order to be able to code the connector effectively. We made these decisions for an example repository, Docs4U, and wrote and tried out a corresponding connector. Finally, we learned about some case studies of problems that required debugging beyond the standard Java code issues, and explored the tools that exist for tough connector-related debugging tasks.

In the next chapter, we'll move on to writing authority connectors. We'll write and test an authority which provides search engine security for Docs4U documents, and we'll describe case studies of real security integrations and their pitfalls.