6

Ground rules for writing connectors

This chapter covers

- An overview of connectors
- The lifecycle of a connector instance
- Connector UI conventions
- Being a good connector citizen

In this chapter, you will learn the conventions for writing a connector. The knowledge you gain here will be applicable to repository, authority, and output connectors alike. We'll examine the basic connector abstraction, and introduce the lifecycle of a connector class instance. Then we'll describe and demonstrate all the various connector UI conventions, including both graphical conventions as well as coding conventions. Finally, we'll spend some serious time thinking through the architecture of a connector, and how to decide whether to use extra connector processes or not.

The examples presented in this chapter will be narrow in scope, and targeted to the immediate topic at hand. We're not going to be quite ready yet to put it all together and write complete connectors; that will remain the point of chapters 7, 8, and 9.

6.1 Connector classes

Your friend Bert has surprised you. Early on, he successfully learned how to use the ManifoldCF UI, the ManifoldCF API, and even made a successful security integration with his search engine. And just last week, he made it through the toughest experience yet – learning about how to use the locking, caching, and database abstractions of ManifoldCF in a correct manner to build a connector.

Now, he clearly feels like he's getting close. When you talk, it is as if his fingers are poised over the keyboard, as he eagerly awaits your permission to get started coding his first connector. But, eager as he is, you tell Bert that he is going to have to put his hands back into his pockets and take a cold shower, because he's still missing something critical for success. That something is learning how his connector needs to fit into ManifoldCF in a manner that makes it a good citizen and a team player, and not as a sort of software wrecking ball.

Learning the ground rules is, fortunately, not rocket science. But it is absolutely necessary, because a connector in ManifoldCF does not have any degree of isolation from the rest of the ManifoldCF infrastructure. The connector is simply a Java class, as are all of the other connectors registered with ManifoldCF. All ManifoldCF connectors should coexist in such a way that they can all run at the same time, if necessary. But, since many connectors must use third-party Java libraries, or even native code, the truth is that great care is needed to preserve the ability of connectors to not interfere with one another.

But even before we get to that, we're going to need to understand more about what a connector is, and what it does. That's where we'll start.

6.1.1 General overview

When you write a connector in ManifoldCF, you are writing a Java class. No matter what kind of connector it is (repository, authority, or output), that class must implement the `org.apache.manifoldcf.core.interfaces.IConnector` interface. This interface describes the characteristics of all connector classes. The methods related to this interface are summarized in table 6.1.

Table 6.1 IConnector methods and their meanings

Method	Meaning
`install()`	Installs any database tables needed for the connector.
`deinstall()`	Removes any database tables that were installed by this connector.
`connect()`	Establish a set of configuration information to be used by this connector instance.
`check()`	Verify that the connection, as configured, is working, or if not, return an explanatory message describing why not.
`poll()`	Free up any costly resources in use, after an appropriate time has elapsed.
`disconnect()`	Forget about all configuration information.
`getConfiguration()`	Retrieve the current configuration information, or null if the

	connector instances is currently unconfigured.
`clearThreadContext()`	Disassociate this connector class instance from any thread.
`setThreadContext()`	Associate this connector class instance to a specified thread context.
`outputConfigurationHeader()`	Part of the UI; output the header part of the connector configuration component.
`outputConfigurationBody()`	Part of the UI; output the body part of the configuration component.
`processConfigurationPost()`	Part of the UI; process POST data from a form post that includes the configuration component.
`viewConfiguration()`	Part of the UI; view the configuration.

As you can see, the meaning of some of these methods is obvious. The `install()` and `deinstall()` methods are, for instance, quite easy to understand and implement. But to understand the rest, we need to understand the lifecycle of a connector instance within ManifoldCF, and also how the ManifoldCF crawler UI works. We'll start with the lifecycle of a connector class instance first.

6.1.2 Connection instance pooling and lifecycle

Every connection that you define within ManifoldCF describes the details of how to communicate with an external system somewhere. But what it doesn't do is to actually give you the means to do so, in Java. In order to do that, we somehow need to establish the actual connection to the external system. And, before we do **that**, we will need to create and initialize a connector class instance. I'll talk more about that in a moment. But first, let's try to understand how things look from the outside, from the point of view of code that needs to use a connector instance.

From the perspective of a thread trying to get work done, ManifoldCF uses a *handle* model for allocating connector class instances. That is, components which need an external connection must request a *connection handle* from the appropriate *connection pool*. The component then makes use of the handle for a time. When it is done, it explicitly releases the handle back to the pool. See figure 6.1 for a pictorial representation.

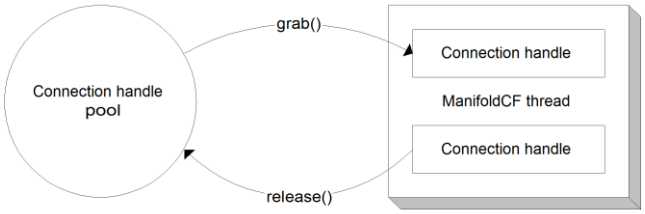

Figure 6.1 How a ManifoldCF thread looks at a connection handle pool. When the thread needs a connection handle, it grabs it, and when it is done, it releases it.

The handle model was chosen because it is able to easily limit the number of outstanding connection handles to some predetermined or configured count. As we have seen in previous chapters, there are often very strong reasons for needing to limit the number of actual connections an external system, and the handle model helps to enforce that restriction. Additionally, it is sometimes expensive time-wise to set up an external connection, and keeping a pool of already-established connections can save a large amount of time. The scenario is such a common one that ManifoldCF was designed with the concept of connection pooling built into it from day one.

Since we know that connector class instances are established and persist for extended periods of time, it makes sense to examine their lifecycle. But we're going to need to disentangle a number of related but distinct concepts before we do this.

CONNECTION INSTANCES

Unfortunately, it's no longer going to be sufficient to just talk about a "connection" without being extremely confusing, since we've already got several kinds of connection-related concepts and objects floating around. The kind of "connection" we've already seen is the kind we can define in the crawler UI. Its full name is *connection definition*, and its purpose is to describe the configuration parameters for how to connect to an external system. The management of connection definitions will be discussed at length in Chapter 11.

When it comes time to actually connect to an external system, we need repository-specific Java classes to manage such connections. That's what a connector class – that is, a class that implements `IConnector` – is for. But there's more involved in getting a working connector class instance than just instantiating one. In order to be useful, the class instance must be properly configured and prepared. For example, we will need to apply configuration information from an appropriate connection definition to it. It's helpful to call these configured class instances by a new name. I call them *connection instances.*

> **Definition** A *connection instance* is a connector class instance that has been configured with the appropriate connection configuration information.

Creating a connection instance from a connector class instance is straightforward. There is a connector method called `connect()`, and its sole argument is a body of configuration information. The configuration information comes from the desired connection definition's database table row, where it is stored as an XML document. The appropriate connection definition manager loads the row, creates a connection definition paper object, and parses the configuration XML into an in-memory data structure before the connector class instance's `connect()` method is called.

Once a connector class instance's `connect()` method is called, then the class instance is considered to be a bona fide connection instance. It can then be reused as many times as needed. Eventually, though, either ManifoldCF will be shut down, or the size of the pool will be reduced by user intervention. At that point, ManifoldCF may disconnect the connection instance, turning it back into just a plain connector class instance.

When the time comes to disconnect a connection instance, ManifoldCF notifies the connector class instance of this event by calling the connector class's `disconnect()` method. It is the responsibility of the connector class at that point to "forget" about all the configuration information it was handed before.

CONNECTION HANDLES

The term for a connector class instance that has been taken from the connection pool and prepared for use with a particular ManifoldCF thread is *connection handle*. You have seen the term before in figure 6.1 and elsewhere. As that figure indicates, code that needs to use a connector should use a handle factory to get hold of one of these handles (using a factory method called `grab()`). When it is done, it must release the handle (using a factory method called `release()`).

> **Definition** A *connection handle* is a connector class instance that has been prepared properly for use by a particular ManifoldCF thread. It is a kind of connection instance, so it has already been configured with connection information.

Figure 6.2 shows the states that a connector class instance can be in, the `IConnector` methods that are called to transition between them, and the names I will use to describe a connector class instance that is in each state.

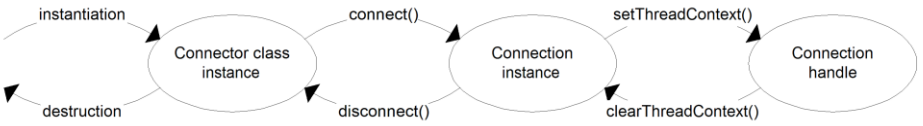

Figure 6.2 The states a connector class instance can be in, and the IConnector methods used to transition between them.

> **Remember** All of the objects we've been talking about are in fact just instances of the underlying connector class. If they have been configured, they are called connection instances. If they've been prepared for use by a thread, they are called connection handles.

THREAD CONTEXTS

This business of assigning or associating a connector class instance with a thread is somewhat mysterious. What exactly does it mean? What is happening from the perspective of the connector class instance?

The answer involves thread contexts. As you may recall, in Chapter 5 we discussed what a thread context is, and several models that can be used to share objects across threads when thread contexts are involved. Connection instances are shared objects, and thus they need to use one of these models. Associating an object with a thread means that the object is given the current thread's thread context, via the `setThreadContext()` connector method. The object may then choose to keep the thread context around, and/or use it for constructing derivative objects, such as the correct database abstraction handle. Disassociating the object from the thread, conversely, means that the object must "forget" all references that were established using that thread context, and revert to a state where none of its direct or indirect members retain any such references. This should be done when the `clearThreadContext()` connector method is called.

ACTUAL EXTERNAL CONNECTIONS

One potentially confusing aspect of the connection lifecycle is figuring out exactly when an actual connection to an external system will be created or destroyed. One might think that the connector class's `connect()` method would be the right place to set up such a connection. But this is not the case. As we've discussed, all that it means for a connector class instance to be "connected", according to ManifoldCF, is that it has been handed the appropriate configuration parameters, nothing more.

The connector class is expected to use these parameters to set up the connection to the external system **only at the time when it is actually needed**, as a side effect of some other connector method getting called. Indeed, a connector class is expressly limited in what it can do in the `connect()` method, because an exception cannot be thrown from within it. This limitation is in effect largely to encourage connector writers to use an on-demand model for external system connection setup.

Destroying an actual external connection usually takes place either when the connector's `disconnect()` method is called, or when enough time has elapsed so that it would be impolite to keep the actual connection around. We'll talk about how that is done next.

POLLING

Often, a connector class instance would like to close an actual external connection it has to an external system when the class instance has been sitting unused in the connection pool for a long-enough period of time. There are a number of reasons why this might be a good

idea. The typical reason is because of a limited number of allowed connections to the external repository.

To support connectors that have this need, a connector class may implement a method called `poll()`. This method is called by ManifoldCF at regular intervals on every connection instance that resides in the associated connection pool. When `poll()` is called, the connector class instance may safely choose to expire its internal connection to the external system if the period of inactivity has been sufficiently long.

> **Note** Expiring an actual connection to an external system is **not** the same as disconnecting the connection instance! ManifoldCF considers the connection instance to be connected until it calls the disconnect() method.

RECAP

To summarize, figure 6.3 is a pictorial state diagram of the full connector class instance lifecycle, including the `poll()` operation.

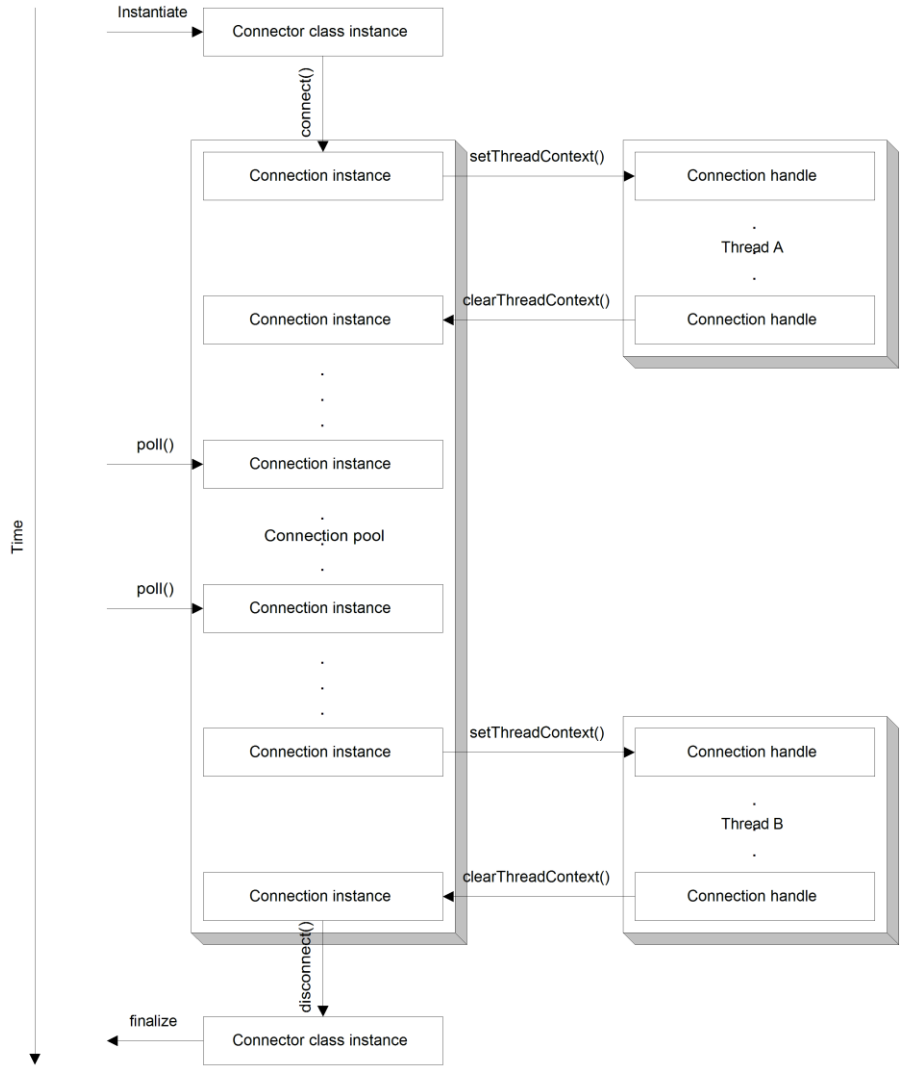

Figure 6.3 The lifecycle of a connector class instance, from the perspective of the connector class instance, passing from instantiation, through entering the pool, with two excursions into use by ManifoldCF threads for various purposes.

As you can see, the same class instance changes state based on where it currently resides in the diagram. Before the connector class instance enters the pool, it is not configured, and therefore does not represent any particular connection definition. In order to get into the connection handle pool, a configuration step is needed, which is what the

`connect()` method call does. Similarly, when a connection handle is pulled from the pool by way of the `grab()` method, its thread context is set via the `setThreadContext()` method. On return to the pool via the factory's `release()` method, the connection handle's `clearThreadContext()` is called to undo all thread affinity. Finally, when it is time to remove the connector class instance from the pool, the `disconnect()` method is called to insure all internal connections are cleaned up.

Now that we understand the lifecycle of a connector class instance, we have the beginnings of the picture of what makes a connector a good citizen within ManifoldCF. In the next section, we'll go beyond what is required by the `IConnector` interface, and explore another aspect of every connector: the connector's UI component.

6.2 Crawler UI conventions

In addition to supporting the standard connection lifecycle, every connector also has a UI component, which defines how the connector presents itself in ManifoldCF's Crawler UI. The UI component typically consists of at least four UI-related methods, which all connectors need to implement. We'll discuss only the methods mentioned in table 6.1 in this chapter – those that are part of the `IConnector` interface – but the truth is that **all** UI-related connector methods have a very similar setup, so once you learn how it's done, you should have no trouble applying the same techniques everywhere.

The four methods must be viewed in the context of HTML forms, with Javascript, because that is the technology that is used for the Crawler UI. So, we'll begin by examining the HTML page structure within which these methods operate. If you are unfamiliar or rusty with HTML forms, this would be a good time to review them. There are several excellent on-line tutorials, which should be sufficient to refresh your memory.

6.2.1 Form structure

The canonical ManifoldCF configuration editing HTML page consists of a HEAD section and a BODY section. The BODY section further contains an HTML multipart form, which is named `editconnection`.

It is in relation to this page structure that you can best understand the purpose and function of the `outputConfigurationHeader()` and `outputConfigurationBody()` connector methods. These methods are called by the ManifoldCF framework in the HEAD section, and within the form, respectfully. Listing 6.2 shows the canonical relationship.

Listing 6.2 Canonical structure of a ManifoldCF configuration editing form

```
<html>
  <head>

    ... outputConfigurationHeader() ...

  </head>
  <body>
```

```
      ...

      <form name="editconnection" method="POST"
        enctype="multipart/form-data">

        ... outputConfigurationBody() ...

      </form>

      ...

    </body>
  </html>
```

Thus, the `outputConfigurationHeader()` method must only output HTML that is consistent with an HTML HEAD section, while `outputConfigurationBody()` may output HTML that is consistent with being in a form. The `outputConfigurationHeader()` method typically outputs any Javascript functions needed by the connector's HTML, while the `outputConfigurationBody()` method receives the current configuration data for the connection definition as an input argument, and presents this data in the form of HTML tables and form elements.

Astute readers will note that I've heretofore glossed over exactly how the canonical form is actually posted. At least two different kinds of posts are required: the form can either post effectively to itself (which is sometimes known as *reposting*), or it can post to a page that will save the connection definition information in the database and then display it (a *view* page). In either situation, a connector's `outputConfigurationBody()` method likely created form elements that only the connector knows how to deal with. So, how are these handled?

The answer is by means of the connector's `processConfigurationPost()` method. The purpose of this method is to gather up form data that has been posted, and put it back together into a connection definition's configuration structure. This involves no HTML at all, and this method has no way to output any. It will be called before the start of whatever the post's target page is, without disrupting the HTML for that page in any way. The ManifoldCF crawler UI framework makes sure that the connector's `processConfigurationPost()` method will be called, whether the post's target page was the edit page or the view page, if the connector's `outputConfigurationHeader()` and `outputConfigurationBody()` methods were called on the corresponding edit page.

The final UI method, `viewConfiguration()`, is called only from a connection definition's view page. It receives the connection definition's configuration information as an argument, and outputs HTML in a viewable manner, usually in the form of HTML tables.

Now that we understand a bit about the relationships between the HTML edit page and the UI-related connector methods, let's go into more detail about each method, and its responsibilities and framework support.

6.2.2 Form Javascript

The `outputConfigurationHeader()` method is used to output the Javascript functions that the connector's UI needs. This Javascript is used to check values in the form, pop up an error if the form data needs to be corrected, and actually permit the form to be submitted if everything checks out. The way this works is that the Crawler UI HTML pages look for specific Javascript functions which a connector may define. For instance, the Crawler UI's pages look for a Javascript function called `checkConfig()` just prior to the posting of every configuration edit page. In addition, just prior to posts that are the result of a save request, the Crawler UI pages look for a function called `checkConfigForSave()`. If the sought function exists, it will be called, and its return value will then determine whether or not the form will be posted – if the value returned is `true`, then the form will be submitted as it currently stands, otherwise the post will not take place.

Every connector's `outputConfigurationHeader()` method may therefore output one or both of these Javascript functions, which may do whatever is needed to insure the integrity of the data. To assist in this task, the crawler UI framework makes available a number of Javascript functions that you can safely use. These methods are summarized in table 6.2.

Table 6.2 Crawler UI helper Javascript functions for connector configuration editing form support

Javascript function	What it does
`SelectTab()`	This function reposts the form, if necessary, and selects the specified tab. Typically this function is used to allow a connector's Javascript to highlight a bad value on a specific tab, usually from within the `checkConfigForSave()` Javascript function.
`postForm()`	This function reposts the form. It is rarely used for configuration forms, because there is little need to repost these.
`postFormSetAnchor()`	This function reposts the form, and sends browser after the repost to the specified anchor value. It is rarely used for configuration forms, because there is little need to repost these.
`isInteger()`	This function returns true if the specified quantity is a numeric integer, or false otherwise. It is typically used to check an individual form value for validity.
`isRegularExpression()`	This function checks to be sure the supplied value is a valid regular expression. It is typically used to check an individual form value for validity.

There are, of course, plenty of standard Javascript functions and methods available as well. It is often useful, for instance, to alert the user to a bad value. This is easily accomplished using the Javascript `alert()` function.

```
alert("You messed something up");
```

You can also set the focus on a bad value after letting the user know that something is wrong. The following standard Javascript accomplishes this.

```
editconnection.myformelement.focus();
```

A short example of what a simple `outputConfigurationHeader()` might look like is presented in listing 6.3.

Listing 6.3 A simple outputConfigurationHeader() implementation

```
  public void outputConfigurationHeader(IThreadContext threadContext,
    IHTTPOutput out, Locale locale, ConfigParams parameters,
    ArrayList tabsArray)
    throws ManifoldCFException, IOException
  {
    tabsArray.add("Server");                                          #A
    out.print(                                                        #B
"<script type=\"text/javascript\">\n"+
"<!--\n"+
"function checkConfigForSave()\n"+                                    #C
"{\n"+
"  if (editconnection.server.value == \"\")\n"+                       #D
"  {\n"+
"    alert(\"Need a server name\");\n"+
"    SelectTab(\"Server\");\n"+                                       #E
"    editconnection.server.focus();\n"+                               #E
"    return false;\n"+                                                #E
"  }\n"+
"\n"+
"  if (editconnection.server.value.indexOf(\"/\") != -1)\n"+          #F
"  {\n"+
"    alert(\"Server name cannot include path information\");\n"+
"    SelectTab(\"Server\");\n"+
"    editconnection.server.focus();\n"+
"    return false;\n"+
"  }\n"+
"  return true;\n"+
"}\n"+
"//-->\n"+
"</script>\n"
    );
  }
```

#A Enumerate the connector tabs
#B Output HTML
#C Create Javascript function called on save post
#D Look for empty value in form element
#E Error: select appropriate tab, send focus to the right element, return false
#F Look for "/" character

Of course, none of this makes sense in the absence of the connector form elements themselves. But hopefully you get the general idea. It is not my intention here to provide a complete Javascript tutorial, but to give you a rough idea of the kind of things you should be thinking about in terms of a connector's configuration Javascript. One of the best ways to learn how to write Javascript for connectors (and connector code in general) is to look at the code for existing connectors that are part of ManifoldCF. Also, in chapters 7, 8, and 9 we will be developing complete examples of code for connectors that includes the UI component.

There are better ways to output Javascript and HTML which do not require us to embed code inside our Java classes. I'll be describing one such method in the next section.

6.2.3 Writing tabs

The `outputConfigurationBody()` connector method is responsible for creating the HTML that represents all tabs that the connector wishes to display as part of the connection definition editing screen. Not only must this method know how to display any of the connector's configuration tabs, it must also translate configuration data that belongs to inactive tabs into hidden form elements, so that all the configuration information is posted in one form or another during a repost or a save. Fortunately, there is a very straightforward technique for making sure this all happens correctly, as we'll explore shortly.

Each time the `outputConfigurationBody()` method is invoked, it receives as arguments both the current connection definition's configuration information, as well as the name of the currently active tab. The tab specified may or may not be one of the tabs the connector knows how to present. But if you use the recommended form for the `outputConfigurationBody()` method, that will not matter. See listing 6.4.

Listing 6.4 One recommended form of the outputConfigurationBody() connector method

```
public void outputConfigurationBody(IThreadContext threadContext,
  IHTTPOutput out, Locale locale, ConfigParams parameters, String tabName)
  throws ManifoldCFException, IOException
{
  ...                                                                  #A

  if (tabName.equals("tab#1"))                                         #B
  {
    ...                                                                #C
  }
  else
  {
    ...                                                                #D
  }

  if (tabName.equals("tab#2"))                                         #E
  {                                                                    #E
    ...                                                                #E
  }                                                                    #E
```

```
    else                                                                    #E
    {                                                                       #E
      ...                                                                   #E
    }                                                                       #E

    ...                                                                     #F
}
```

#A Do any prep work needed
#B Is the current tab "tab#1"?
#C If tab is "tab#1", display the tab
#D If tab isn't "tab#1", store the tab data in hiddens
#E Also take care of "tab#2"
#F Do the rest of the tabs

The conditional logic of the recommended structure insures that hidden form data will be created for all tabs that are not active. Thus, this code will do the right thing for a wide variety of circumstances. In particular, because of the decentralized way form data is handled, we can be pretty sure that the form won't break no matter how it is posted or reposted in the end.

You can also break up the method in figure 6.3 into calls to individual sub-methods. This is particularly useful when there are a lot of tabs, or the HTML produced by any single tab is very complicated. The recommended breakdown is to have one sub-method per tab. The `outputConfigurationBody()` method then becomes a series of calls out to individual tab methods, for example:

```
outputTabXXX(out, parameters, tabName);
outputTabYYY(out, parameters, tabName);
outputTabZZZ(out, parameters, tabName);
```

The connection definition data is represented as an `org.apache.manifoldcf.core.interfaces.ConfigParams` object. If this object sounds familiar, it is because we've worked with this sort of object before, back in Chapter 3, where it appeared in an API context to represent the configuration information for a connection definition. This is also the way the connector's `processConfigurationPost()` receives the configuration information it is supposed to edit, and the `viewConfigurationPost()` gets the configuration information it is supposed to display. The `ConfigParams` object can be accessed and edited in two ways – the first being as a hierarchical arrangement of named nodes with attributes, and the second being as a simpler set of name/value pairs. The latter is easier to work with and suffices for most connectors. That's how it's done in listing 6.5, which is a simple example of the `outputConfigurationBody()`, `processConfigurationPost()`, and `viewConfiguration()` methods corresponding to the `outputConfigurationHeader()` method presented in listing 6.3.

Listing 6.5 Simple in-line example outputConfigurationBody(), processConfigurationPost() and viewConfiguration() methods

```
public void outputConfigurationBody(IThreadContext threadContext,
```

```
    IHTTPOutput out, Locale locale, ConfigParams parameters,
    String tabName)
    throws ManifoldCFException, IOException
  {
    String server = parameters.getParameter("Server");                    #A
    if (server == null)                                                   #A
      server = "";                                                        #A

    if (tabName.equals("Server"))                                         #B
    {
      out.print(                                                          #C
"<table class=\"displaytable\">\n"+                                       #C
"  <tr><td class=\"separator\" colspan=\"2\"><hr/></td></tr>\n"+          #C
"  <tr>\n"+                                                             #C
"    <td class=\"description\"><nobr>Server:</nobr></td>\n"+             #C
"    <td class=\"value\">\n"+                                           #C
"      <input type=\"text\" size=\"32\" name=\"server\" value=\""+       #C
  org.apache.manifoldcf.ui.util.Encoder.attributeEscape(server)+"\"/>\n"+#C
"    </td>\n"+                                                          #C
"  </tr>\n"+                                                            #C
"</table>\n"                                                            #C
      );                                                                #C
    }
    else
    {
      out.print(                                                        #D
"<input type=\"hidden\" name=\"server\" value=\""+                      #D
  org.apache.manifoldcf.ui.util.Encoder.attributeEscape(server)+"\"/>\n"+#D
      );                                                                #D
    }
  }

  public String processConfigurationPost(IThreadContext threadContext,
    IPostParameters variableContext, ConfigParams parameters)
    throws ManifoldCFException
  {
    String server = variableContext.getParameter("server");             #E
    if (server != null)                                                 #F
      parameters.setParameter("Server",server);                         #F
    return null;                                                        #G
  }

  public void viewConfiguration(IThreadContext threadContext,
    IHTTPOutput out, Locale locale, ConfigParams parameters)
    throws ManifoldCFException, IOException
  {
    out.print(                                                          #H
"<table class=\"displaytable\">\n"+                                     #H
"  <tr>\n"+                                                           #H
"    <td class=\"description\"><nobr>Server:</nobr></td>\n"+           #H
"    <td class=\"value\">\n"+                                         #H
"      "+                                                             #H
  org.apache.manifoldcf.ui.util.Encoder.bodyEscape(                   #H
  parameters.getParameter("Server"))+"\n"+                            #H
"    </td>\n"+                                                        #H
```

```
"  </tr>\n"+                                                    #H
"</table>\n"                                                    #H
    );                                                          #H
  }
```

#A If "Server" parameter is null, make it empty
#B See if the current tab is "Server"
#C Display a form element with the server name
#D Include a hidden form element with the server name
#E Get a post variable called "server"
#F Set the "Server" parameter to the new value
#G Signal that no error occurred
#H Output the parameter value of "Server"

If you are thinking at this point just how ugly this code is, I agree. It is because of the presence of multiple languages all embedded in Java. But as we alluded to earlier, there are much better ways to represent the same functionality.

One of the best is by using Apache Velocity. ManifoldCF has supported Velocity for HTML templates starting in the 0.5-incubating release. Velocity is a templating tool to which you provide named objects in a *Velocity Context*. Then, you can access those objects, evaluate conditionally, and even loop repeatedly inside the template. Listing 6.6 shows what a Velocity template for the above `outputConfigurationBody()` method might look like.

Listing 6.6 A simple Velocity template

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

<table class="displaytable">
  <tr><td class="separator" colspan="2"><hr/></td></tr>
  <tr>
    <td class="description"><nobr>Server:</nobr></td>
    <td class="value">
      <input type="text" size="32" name="server"                #B
        value="$Encoder.attributeEscape($server)"/>             #B
    </td>
  </tr>
</table>

#else                                                           #C

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

#end
```

#A Conditionally render based on tab name
#B Output the server name from context, escaping properly
#C Render as hiddens if tab is not visible
#D Same variables should be in context

In this template, the Velocity context is presumed to contain a variable called `server` and another one called `TabName`. The ManifoldCF integration with Velocity automatically

provides other objects in the Velocity context, which is where the Encoder variable comes from. Table 6.3 describes the provided objects and their names.

Table 6.3 Standard Velocity context variables and their meanings

Variable	Meaning
`$Encoder`	Access to the escaping utility class `org.apache.manifoldcf.ui.util.Encoder`
`$Formatter`	Access to the formatting utility class `org.apache.manifoldcf.ui.util.Formatter`
`$MultilineParser`	Access to the multi-line parsing class `org.apache.manifoldcf.ui.util.MultilineParser`
`$ResourceBundle`	For internationalization, access to a `ResourceBundle` appropriate for the provided locale

Listing 6.7 shows what the `outputConfigurationBody()` method might look like when using Velocity.

Listing 6.7 Outputting a configuration body using Velocity

```
public void outputConfigurationBody(IThreadContext threadContext,
    IHTTPOutput out, Locale locale, ConfigParams parameters,
    String tabName)
    throws ManifoldCFException, IOException
  {
    Map<String,Object> velocityContext = new HashMap<String,Object>();
    velocityContext.put("TabName",tabName);                          #A
    String server = parameters.getParameter("Server");
    if (server == null)
      server = "";
    velocityContext.put("server",server);                            #B
    Messages.outputResourceWithVelocity(out,locale,                  #C
      "Configuration_Server.html",velocityContext);                  #C
  }
```

#A Put tab name into context
#B Put server name into context
#C Render the template

ManifoldCF Velocity templates are typically accessed as resources with the same package path as the class that references them. To make this easier, it's something of convention to place a `Messages` class in a package corresponding to the location where the templates are going to be placed. As long as your `Messages` class extends `org.apache.manifoldcf.ui.i18n.Messages`, all the necessary functionality will come along with it. You can find such convenience classes in every connector that is provided with ManifoldCF.

This is all pretty straightforward; the most difficult technical aspect of it is that there are some references within to styles and helper methods we have not yet been introduced to. But a formal introduction is next. We'll start with ManifoldCF's utility methods for HTML and Javascript encoding support.

6.2.4 HTML and Javascript escaping

The construction of HTML often involves the conversion of Java strings (which are Unicode) to strings that represent the same thing but are escaped properly for an HTML context. For example, in an HTML body the string "A > B" must be converted to "A > B", because the ">" character is not allowed in HTML as a free-standing character, but only as part of a tag.

ManifoldCF provides a number of helper methods for making this escaping easy. All you need to know is what context you are escaping for, and you can select the proper method easily. The escaping methods are all methods of class `org.apache.manifoldcf.ui.util.Encoder`. I've summarized them in table 6.4.

Table 6.4 Encoding methods and where to use them

Method	Purpose
`attributeEscape()`	Escape a string for inclusion in an HTML tag attribute.
`bodyEscape()`	Escape a string for inclusion in HTML, but not as part of any HTML tag.
`attributeJavascriptEscape()`	Escape a string for inclusion as part of a Javascript string within an HTML tag attribute.
`bodyJavascriptEscape()`	Escape a string for inclusion as part of a Javascript string within an HTML body section.

In case it's not clear, here's a short Java code example showing the proper usage of three of these methods.

```
out.print(
  "<input type='button' value='" + attributeEscape(myButtonText) +
  "' onclick='Javascript:myJavascriptFunction(\"" +
  attributeJavascriptEscape(myJavascriptArgument) + "\")'/>" +
  bodyEscape(myText)
);
```

The corresponding Velocity template snippet might look like this:

```
<input type='button' value='$Encoder.attributeEscape($myAttributeText)'
  onclick='Javascript:myJavascriptFunction(
    "$Encoder.attributeJavascriptEscape($myJavascriptArgument)"'/>
$Encoder.bodyEscape($myText)
```

Once you include the correct escaping, your connector methods should all be functional. But will they be pretty, or even readable? That will depend on your use of ManifoldCF's styles. We'll explore what's available next.

6.2.5 Using ManifoldCF styles

The styles that ManifoldCF makes available for a connector's UI component to use are necessarily limited, since every connector must present itself within the confines of the tab structure in the crawler UI. This may offend those readers who bring a graphics design skill set to the process of writing a connector.

Limiting as it may be, readers are strongly urged to stay within the boundaries of ManifoldCF's currently accepted graphical structure, unless they are willing to contribute their new graphical paradigm back to the ManifoldCF project. The reason for this is simple consistency; your connector's HTML will be only a part of the HTML that ManifoldCF generates for any page containing it. So it won't look good unless you change more than just your connector. And, if you change the styles for the crawler UI as a whole, which is probably the right way to go about making such an improvement, it's essential that everyone else's connector UI code comes along for the ride.

Generally, the styles that ManifoldCF expects you to use for connectors are based around two basic paradigms. You are free to use whichever paradigm fits your needs, and even mix and match them.

THE DESCRIPTION/VALUE PARADIGM

The first paradigm is centered around presentation of description/value pairs, where there is a table in which each row represents a description and a value, or a light separator or a dark separator. Tables can be nested so that the value can indeed be a table in its own right, drawn with a box around it.

Table 6.5 lists the styles that you would use, and where you would use them, to execute this paradigm.

Table 6.5 ManifoldCF CSS styles involved in the description/value paradigm

Style name	Where applied	What it does
`displaytable`	`<table>`	Style for a table containing description/value data.
`separator`	`<td>`	Style for a separator between rows of description/value data.
`lightseparator`	`<td>`	Style for a lighter-colored separator between rows of description/value data.
`message`	`<td>`	Style for a message that is centered in the row.
`description`	`<td>`	Style for the description part – the left side – of a description/value pair.

value	<td>	Style for the value part – the right side – of a description/value pair.
boxcell	<td>	Style for a boxed cell, usually replacing the value part of a description/value pair and meant to have a nested <table> within.

We can combine these into a simple HTML example, so you can get an idea what these styles currently look like. Consult listing 6.8 for this example.

Listing 6.8 Simple example HTML for the description/value paradigm

```
<table class="displaytable">
  <tr>
    <td class="description"><nobr>Description:</nobr></td>
    <td class="value">Value #1</td>
  </tr>
  <tr><td class="lightseparator" colspan="2"><hr/></td></tr>
  <tr><td class="message" colspan="2">Important message</td></tr>
  <tr><td class="separator" colspan="2"><hr/></td></tr>
  <tr>
    <td class="description"><nobr>Description #2:</nobr></td>
    <td class="boxcell">
      <table class="displaytable">
        <tr>
          <td class="description"><nobr>Inner description:</nobr></td>
          <td class="value">Inner value</td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

When the HTML in listing 6.8 is displayed with the ManifoldCF style sheet in effect, the result looks like the screen capture in figure 6.4.

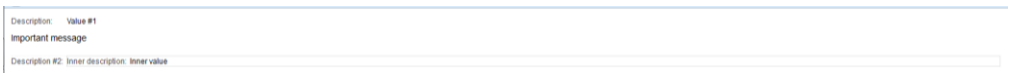

Figure 6.4 Screen capture of the HTML in listing 6.8

THE TABULAR PARADIGM

The second paradigm is provided for situations where the data being displayed needs to be in a more compact form. This paradigm places data in a table with an arbitrary number of columns. The table rows are meant to alternate in color between even-colored rows and odd-colored rows. Table 6.6 lists the CSS styles that are used with this paradigm.

Table 6.6 CSS styles used with the tabular paradigm

Style name	Where applied	What it does
formtable	<table>	Style for a table containing tabular information.
formheaderrow	<tr>	Style for a header row.
formcolumnheader	<td>	Style for a header cell.
evenformrow	<tr>	Style for even table rows.
oddformrow	<tr>	Style for odd table rows.
formrow	<tr>	Style for special table rows.
formcolumncell	<td>	Style for table cells.
formcolumnmessage	<td>	Style for table cells which contain special messages.
formseparator	<td>	Style for a separator cell.

Once again, we can build a simple HTML example out of all of these styles, so you can see what they do. See listing 6.9.

Listing 6.9 HTML example of tabular paradigm

```
<table class="formtable">
  <tr class="formheaderrow">
    <td class="formcolumnheader"><nobr>Title #1</nobr></td>
    <td class="formcolumnheader"><nobr>Title #2</nobr></td>
    <td class="formcolumnheader"><nobr>Title #3</nobr></td>
  </tr>
  <tr class="evenformrow">
    <td class="formcolumncell">Data #1, 1</td>
    <td class="formcolumncell">Data #2, 1</td>
    <td class="formcolumncell">Data #3, 1</td>
  </tr>
  <tr class="oddformrow">
    <td class="formcolumncell">Data #1, 2</td>
    <td class="formcolumncell">Data #2, 2</td>
    <td class="formcolumncell">Data #3, 2</td>
  </tr>
  <tr class="formrow">
    <td class="formseparator" colspan="3"><hr/></td>
  </tr>
  <tr class="formrow">
    <td class="formcolumnmessage" colspan="3">Some message</td>
  </tr>
</table>
```

Displaying the HTML in listing 6.9 in a browser yields something like figure 6.5.

Figure 6.5 Screen capture of the HTML in listing 6.9.

Now that we know how to write the connector methods that present the configuration UI, it's time to move on to the next stage: understanding how to make a connector be properly behaved in the ManifoldCF environment.

6.3 Making your connector a good citizen

All of the clever code in the world will not help you if somewhere you have made a fundamental miscalculation about the acceptable parameters in which your code must behave. In many large organizations, software developers spend most of their working lives writing extremely detailed specifications designed to prevent such oversights. I find this approach to be very cumbersome. My philosophy is a bit different: I prefer to convey understanding and ideas, rather than details. This hopefully empowers you, the software developer, to find your own best solution.

This section is about understanding the critically important principles you will need to embrace before you write even the simplest connector on your own. Many of these principles may be obvious in retrospect, but since you may be new to ManifoldCF, we're likely to save a lot of time and headache by enumerating them up front.

6.3.1 Avoiding external configuration

One of the most basic but also the most easily overlooked principle pertains to what goes into a connection definition, and what doesn't. It is often the case when dealing with third-party external repositories that there are many different ways to configure the connection to these repositories. Sometimes you can do it in a repository-specific configuration file, or you can configure the connection in software, or maybe even through some kind of configuration UI. This plethora of choices tends to confuse software developers, who often wind up trying to take the easiest way out, and define a ManifoldCF connection definition in the simplest possible way, while expecting any actual user of the connector to go off and (for instance) edit a configuration file somewhere.

I'm sure that I don't really need to tell you that this basic instinct should be fought. To the extent possible, your connector's connection definition should contain everything needed to properly configure every kind of working connection to the repository it is meant to address. Any other solution will severely limit your connector's ability to work with multiple repositories at the same time, which, as an approach, may well be obsolete almost immediately.

Once in a while, however, some degree of external configuration turns out to be unavoidable. A prime instance of this can be found in ManifoldCF's Documentum connector. The Documentum Foundation Classes upon which this connector is built depend on a

configuration file, `dmcl.ini`, which tells the DFC classes how to talk with the local Documentum "docbroker". In the Documentum design, it is the docbroker's responsibility to locate and connect ManifoldCF to one of the content servers that the docbroker knows about.

Thus, because of the design of the underlying DFC libraries that the connector depended upon, there was little choice but to play along with the Documentum model. This is despite the obvious limitation that a single instance of ManifoldCF cannot then simultaneously work with more than one Documentum docbroker at a time. It is my sincere hope that you will not find yourself in a similar situation, but if you do, I encourage you to do what you can to limit the amount of external configuration as much as possible.

6.3.2 Obeying thread rules

In Chapter 5, we discussed the difference between a ManifoldCF persistent thread, and a ManifoldCF transient thread. Every good connector needs to be implemented in a fashion that uses threads properly.

You may assume that any public connector methods are called only by persistent threads. It is then up to your connector to appropriately decide when to create transient threads, start them, and wait for them to complete. Listing 6.10 shows the canonical form of how you might spin up such a transient thread.

Listing 6.10 Transient thread canonical form

```
protected MyResultObject performMyAction(MyArgumentObject object1,      #1
  MyArgumentObject object2)                                             #1
  throws ManifoldCFException                                            #1
{
  MyActionThread t = new MyActionThread(object1,object2);               #2
  try
  {
    t.start();                                                          #3
    t.join();                                                           #3

    Throwable thr = t.getException();                                   #4
    if (thr != null)                                                    #4
    {                                                                   #4
      if (thr instanceof ManifoldCFException)                           #4
        throw (ManifoldCFException)thr;                                 #4
      else if (thr instanceof RuntimeException)                         #4
        throw (RuntimeException)thr;                                    #4
      else                                                              #4
        throw (Error)thr;                                               #4
    }                                                                   #4
    return t.getResult();                                               #5
  }
  catch (InterruptedException e)                                        #6
  {                                                                     #6
    t.interrupt();                                                      #6
    throw new ManifoldCFException("Interrupted: "+e.getMessage(),       #6
      ManifoldCFException.INTERRUPTED);                                 #6
  }                                                                     #6
```

```
  }

  protected MyResultObject doMyAction(MyArgumentObject object1,          #7
    MyArgumentObject object2)                                            #7
    throws ManifoldCFException                                           #7
  {                                                                      #7
    ...                                                                  #7
  }                                                                      #7

  protected class MyActionThread extends java.lang.Thread
  {
    protected MyArgumentObject argument1;
    protected MyArgumentObject argument2;
    protected Throwable exception = null;
    protected MyResultObject result = null;

    public MyActionThread(MyArgumentObject argument1,
      MyArgumentObject argument2)
    {
      super();
      setDaemon(true);                                                   #8
      this.argument1 = argument1;                                        #9
      this.argument2 = argument2;                                        #9
    }

    public void run()
    {
      try
      {
        result = doMyAction(argument1,argument2);                       #10
      }
      catch (Throwable e)                                               #11
      {                                                                 #11
        exception = e;                                                  #11
      }                                                                 #11
    }

    public Throwable getException()
    {
      return exception;
    }

    public MyResultObject getResult()
    {
      return result;
    }

  }
```

#1 Declare persistent-thread method
#2 Create the transient thread
#3 Start transient thread and wait for completion
#4 Interpret and rethrow any exceptions
#5 Return result if no exception
#6 Handle interruption
#7 Declare transient-thread method
#8 Always set thread to be daemon type

#9 Save references to arguments
#10 Invoke transient-thread method
#11 Catch all exceptions and record them

At #1, we declare a method meant to be used by a persistent thread. This method creates a transient thread at #2. It starts it and waits for it to complete at #3. At #4, it looks for and rethrows any exceptions the thread may have encountered. If there was no exception, the result is obtained and returned at #5. Should this persistent-thread method have been interrupted, at #6 we signal the transient thread about the interruption, as a courtesy, and throw an appropriate `ManifoldCFException`.

At #7, we declare the method that actually does the work, but must do it within a transient thread. Because it is meant to run only in a transient thread, it cannot use any of the ManifoldCF core services.

The actual transient thread is straightforward; at #8 the thread is set to be of Java daemon type, and at #9 the arguments are saved in the constructor. At #10 we invoke the transient-thread method. At #11, we catch all possible exceptions, and save them for later consideration.

The times when your connector should consider creating transient threads are:

- When your connector needs to directly communicate through a socket connection
- When your connector calls a third-party library method that may communicate through a socket connection
- When your connector communicates to another process using RMI

Astute readers may note that Java RMI, or Remote Method Invocation, does in fact use sockets under the covers. You may wind up needing to use this technology if your connector relies on third-party code that has certain characteristics. We'll be discussing those characteristics shortly.

6.3.3 Using local trust stores

Java's Secure Sockets Layer (SSL) implementation is often used to communicate securely to another process or system. Many repositories support, or even require, communications using this technology.

However, the technology of secure sockets requires the construction of what is known as a *trusted certificate chain*. In such a chain, each entity is represented by a certificate. Furthermore, a certificate may be *signed* by another certificate, which establishes a mechanism whereby this chain of trust can be built.

Secure communications can be set up only if the Java SSL implementation is convinced that the target server's certificate should be trusted, either directly or transitively. This means that Java must keep one or more "trusted" certificates around in order to be able to set up such connections. The place that such certificates are kept is called a *trust store*.

Normally, for a process in Java, there is exactly **one** trust store, which is identified via command-line switches when the process starts. But astute readers will have noticed that Java's standard approach violates the "avoiding external configuration" rule we described earlier in this chapter. So how does ManifoldCF fix this problem?

ManifoldCF addresses the trust store problem elegantly, by providing core infrastructure that allows each connector to have its own trust store. The trust store is kept as a large string within the normal configuration data of a connection definition. A *keystore object* can be constructed from the string representation, and certificates can be added to it or removed. Then, when all the editing is complete, the keystore object can be turned back into a string, which is then placed back into the configuration data.

The same keystore object can be used to create a *secure socket*, which extends `java.net.Socket` and works much like any other socket, except that it negotiates SSL encryption under the covers. Thus, local trust stores give one complete flexibility to add SSL in a connection definition-specific way.

THE TRUST STORE ABSTRACTION

The trust store management abstraction is described by the interface `org.apache.manifoldcf.core.interfaces.IKeystoreManager`. You instantiate one of these objects using an appropriate method in the `org.apache.manifoldcf.core.interfaces.KeystoreManagerFactory` class. You can either create one that is empty, or create one from an ASCII string.

The trust store management abstraction's methods are described in table 6.7.

Table 6.7 Trust store management methods and their meanings

Method	Meaning
`getContents()`	Gets a list of certificate alias names from the trust store.
`getDescription()`	Gets the description of a certificate from the trust store, given the certificate's alias name.
`importCertificate()`	Imports a certificate as a binary stream, giving it an alias name.
`remove()`	Removes the certificate corresponding to a specified alias name.
`getString()`	Converts the trust store into an ASCII string.
`addCertificate()`	Adds a certificate to the trust store, using the specified alias and a Java certificate object.
`getSecureSocketFactory()`	Constructs a secure socket factory from the current trust store, which can be used to create secure sockets.

The trust store is therefore a set of certificates, each of which has a unique alias that needs to be meaningful only to the software that's trying to use the trust store. Certificates can be added, along with the chosen aliases, either as Java certificate objects, or as a stream of binary data. They may also be removed, or inspected to return a descriptive string. Finally, when all editing is complete, the trust store can be converted to a java String, which happens to be the same format that can be used to create a trust store in the first place.

> **Note** Not just any binary data can be brought into a trust store. The binary data must represent a certificate, and obey the formatting restrictions of such certificates. An attempt to add binary data that is not a legal certificate will result in an exception being thrown.

The last method, `getSecureSocketFactory()`, returns a Java secure socket factory object, which is used to generate secure sockets which are aware of the trust store's contents. Socket factories (`javax.net.SocketFactory`) in Java are the standard way sockets are created.

How these are used depends on the entity that is setting up the connection. For example, the commons-httpclient library is used extensively in ManifoldCF for HTTP communication, and most of these support SSL using a local trust store. Other libraries will have different ways of using socket factories.

USING THE TRUST STORE ABSTRACTION

Our trust store example is not very complicated; it builds and manipulates a trust store, adding a couple of certificates to it and removing them. The example code, and the code for all examples in this chapter, can be checked out by:

```
svn co
http://manifoldcfinaction.googlecode.com/svn/examples/edition_2/ground_rule
s_example
```

As usual, I've provided an ant build file to go along with it. The ant target you'll need for the trust store example is `run-trust-store-example`.

When you execute this target, you should see output that looks something like this:

```
run-trust-store-example:
     [java] Configuration file successfully read
     [java] Creating empty trust store...
     [java] Adding first certificate...
     [java] Adding second certificate...
     [java] Converting to string...
     [java] The trust store contents are:
     [java]
/u3+7QAAAAIAAAACAAAAAgAHY2VydCAjMgAAAS5ARAvXAAVYLjUwOQAABQ8wggULMIIE ...
     [java] Building a trust store from string...
     [java] Restored trust store has 2 certs
     [java]  cert #2:[
     [java] [
     [java]   Version: V3
     [java]   Subject: CN=*.apache.org, OU=Infrastructure, O=Apache
```

```
Software Foundation, L=Forest Hill, ST=Maryland, C=US...
     [java] ]
     [java]  cert #1:[
     [java] [
     [java]   Version: V3
     [java]   Subject: EMAILADDRESS=sysadmin@metacarta.com,
O=metacarta.com, L=C
ambridge, ST=Massachusetts, C=US, CN=MetaCarta CA...
     [java] ]
```

The example code is importing two certificates (which I've checked in), and then converting the whole trust store to an ASCII string, reconstituting it as a trust store, then printing the contents. Listing 6.11 shows the code that does this.

Listing 6.11 Trust store manipulation example

```
public class TrustStoreExample
{
  private TrustStoreExample()
  {
  }

  protected static void addCertificate(IKeystoreManager trustStore,
    String fileName, String aliasName)
    throws IOException, ManifoldCFException
  {
    InputStream is = new FileInputStream(fileName);                    #1
    try                                                                #1
    {                                                                  #1
      trustStore.importCertificate(aliasName,is);                      #1
    }                                                                  #1
    finally                                                            #1
    {                                                                  #1
      is.close();                                                      #1
    }                                                                  #1
  }

  public static void main(String[] argv)
  {
    try
    {
      ManifoldCF.initializeEnvironment();                              #2

      System.out.println("Creating empty trust store...");             #3
      IKeystoreManager trustStore = KeystoreManagerFactory.make("");   #3

      System.out.println("Adding first certificate...");               #4
      addCertificate(trustStore,"certificate1.crt","Cert #1");         #4

      System.out.println("Adding second certificate...");
      addCertificate(trustStore,"certificate2.crt","Cert #2");

      System.out.println("Converting to string...");                   #5
      String trustStoreContents = trustStore.getString();              #5

      System.out.println("The trust store contents are:");
```

```
        System.out.println(trustStoreContents);

        System.out.println("Building a trust store from string...");        #6
        IKeystoreManager restoredTrustStore =                               #6
          KeystoreManagerFactory.make("",trustStoreContents);               #6

        String[] contents = restoredTrustStore.getContents();               #7
        System.out.println("Restored trust store has "+                     #7
          Integer.toString(contents.length)+" certs");                      #7
        int i = 0;                                                          #7
        while (i < contents.length)                                         #7
        {                                                                   #7
          System.out.println(" "+contents[i]+":"+                           #7
            restoredTrustStore.getDescription(contents[i]));                #7
          i++;                                                              #7
        }                                                                   #7
      }
      catch (Exception e)
      {
        e.printStackTrace(System.err);
        System.exit(2);
      }
    }
  }
```

#1 Add a certificate from a file
#2 Always initialize the environment
#3 Create a trust store with password “”
#4 Add the first certificate
#5 Convert the trust store to a string
#6 Build a new trust store from the string, also with password “”
#7 Print the contents of the new trust store

At #1, we write a method that imports a certificate from a file and gives it a specified alias name. Since we need to use ManifoldCF infrastructure, always remember to initialize the environment, at #2. At #3 we create an empty trust store, with a blank password. At #4, we add the first certificate to the trust store. At #5, we convert the whole trust store to a string. At #6, we build a whole new trust store from the string we produced. At #7, we iterate through the certificates in the trust store, and print their aliases and contents.

LIMITATIONS

Local trust stores are very flexible, and arguably easier to use than Java’s native way of doing things. But in some cases, your ability to use a local trust store will be thwarted by the design of a third-party repository client library.

It is often the case that client libraries that abstract away from basic socket communication also hide the ability to change how they use SSL under the covers. In such cases, you may have no choice but to (once again) resort to global configuration for what should be connection-specific information.

6.3.4 Using RMI helper processes or custom class loaders

Once in a while, the requirements of the software needed to communicate with a repository are incompatible with the "good citizenship" requirements of ManifoldCF. For example, the communication may require a third-party library that requires customized versions of Java jars that ManifoldCF actually uses to run. Or, some third-party libraries require native code or special process setups. What should a connector developer do in these kinds of situations?

One answer might be to experiment with ManifoldCF to see if you can get it to coexist with your third-party library's special requirements. You might even have some success at doing this. But there will be a cost – which is that everyone else in the ManifoldCF connector community will need to adopt the proprietary special sauce necessary to make your connector work. This is not likely to go over well.

An alternative, which may well be acceptable in some cases, is to use a connector-specific class loader to partially isolate the third-party library from the rest of ManifoldCF. This approach can indeed be used if the conditions of isolation are not too severe. We'll examine this approach a little later.

But if the third-party library restrictions are too dire, there is really no alternative than to isolate use of the problematic library within its own process. No matter how messed up the library is, this approach will always succeed. But the cost of all that communication seems very high.

USING RMI FOR ISOLATION

Luckily, Java has a technology available that makes process isolation of the kind we're talking about relatively straightforward. The technology is called RMI, which stands for *Remote Method Invocation*. Using this technology, you write two Java classes for each object – one of them an interface class, and the other an implementation class. The implementation class is included in the external "server" process, while the interface class is used by the connector in the ManifoldCF process. Lightweight helper classes that are referenced by the interface must be included in both processes. RMI automatically takes care of the socket communication. See figure 6.6.

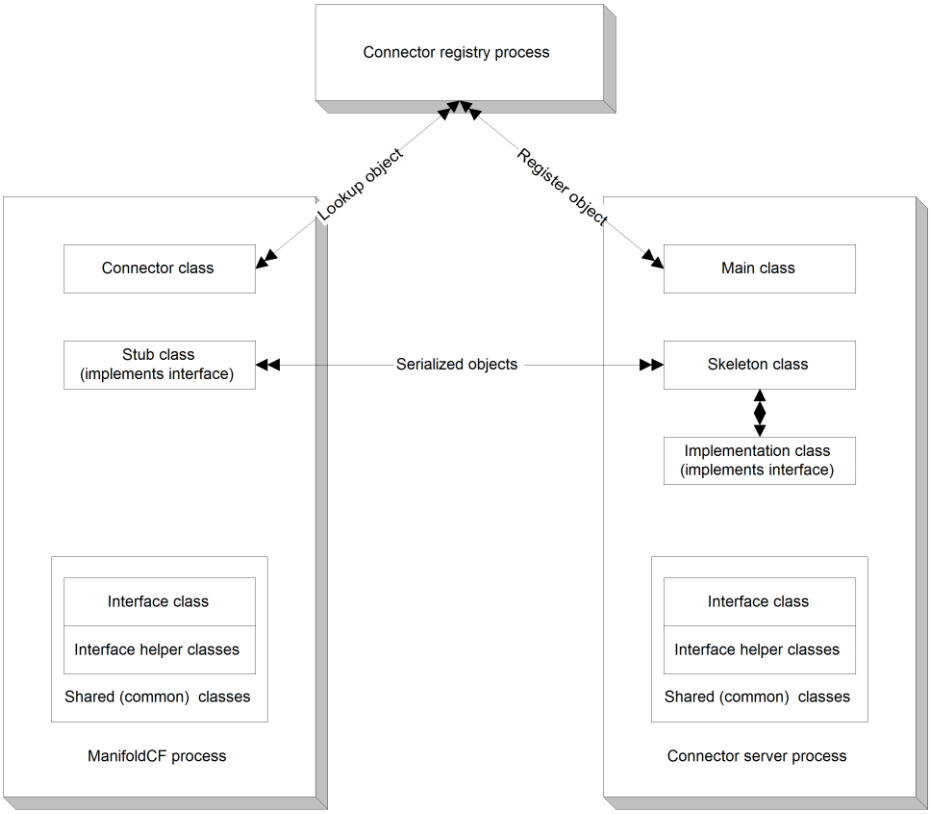

Figure 6.6 Relationships between processes, interfaces, and classes, using RMI isolation.

In the RMI approach, classes involved in the isolation of the questionable library come in two flavors. The first kind is the common helper class, which is ideally a simple *paper object*, a very lightweight class that contains only data and the methods needed to access it. Objects of this kind must be serializable, that is they must implement `java.io.Serializable`, since instances of this class will be passed back and forth across a socket connection. The second kind of class is the kind that RMI bifurcates, so that for each class instance, the ManifoldCF process contains only an instance of a stub class, while the connector server process contains a corresponding instance of a skeleton class and implementation class. The stub class implements the same interface as the implementation class, so your connector does not need to deal with the object any differently than it otherwise would, except for the need to handle a possible `java.rmi.RemoteException`. But invoking a method in the stub class communicates to the corresponding skeleton class instance in the server, which then invokes the corresponding implementation class instance.

Getting hold of the first stub object which represents a server object is a problem. RMI solves this by introducing an *object registry*. The registry is managed by a third process, called the connector registry process. The registry process keeps track of well-known objects in the server process, which can then be looked up by the ManifoldCF process.

In my RMI implementations, I typically only register one such well-known object: a *session factory*. The factory has one method whose job it is to instantiate a secondary objects, which I call a *session object*. A connector class instance usually contains an RMI reference to exactly one such session object. The session object is written to perform all the real work needed to use the isolated library.

> **Note** Since RMI uses sockets to communicate with the server process, the rules prohibiting socket communication using ManifoldCF persistent threads come into play. In practice this means that a connector using RMI must create a transient thread for every access to a stub object, as well as for the session factory lookup.

This is clearly more work to construct than just writing a single connector class! Luckily, there are examples of connectors built using sidecar RMI processes for you to refer to and emulate. The Documentum connector is one such connector. But to make things clear, we'll look at a simple example right now.

AN RMI EXAMPLE

Our RMI example is designed to demonstrate the proper usage of RMI within a connector environment. The example sets up three processes – a registry process, a server process, and a client process. The client process is, however, in no way based on an actual connector, or even a bona-fide ManifoldCF process – instead, it merely shows what your connector will need to do to use RMI as a client. Important considerations, such as the need to create transient threads for RMI interactions are therefore not demonstrated at all. I will presume that the reader is perfectly capable of applying more than one principle at the same time.

If you've checked out the trust store example, you've already checked out the RMI example. The RMI-related ant targets are `run-registry-process`, `run-server-process`, and `run-client process`. Since both the registry process and the server process must be running in order for the client process to run properly, you will need three separate shell windows in order to run this example. So let's go ahead and try it out, and then we'll have a look at the code.

Start the registry process first. When you do, there will be a flurry of compilation activity, and then the registry process will start, and wait:

```
run-registry-process:
     [java] Registry started and is awaiting connections.
```

So far, so good! Now, in another window, start the server process. It, too, should start and wait:

```
run-server-process:
     [java] Server started and is awaiting connections.
```

Use the final window to run the client process. Its output should look like this:

```
run-client-process:
     [java] Obtaining a factory handle...
     [java] Getting a session handle...
     [java] Calling remote method...
     [java] Result: This is my response
     [java] Done!
```

Very nice, but how do we know that it worked? Well, go back and look at the server window, and you will now see the following:

```
run-server-process:
     [java] Server started and is awaiting connections.
     [java] The server process got a string of 'hello'
```

So, clearly, the server process received communication from the client process, and noted that communication!

To see how this is all put together, let's start with the interfaces that describe the remote objects. Listings 6.12 and 6.13 show the session factory and session interfaces that I've defined.

Listing 6.12 The ISessionFactory interface

```
public interface ISessionFactory extends Remote                    #1
{
  public ISession make()                                           #2
    throws RemoteException;                                        #2
}
```

#1 Declare the session factory interface
#2 Method to create a session object

At #1, we define the interface. Since this interface is designed to represent a server-side object, we must extend the `java.rmi.Remote` interface here as well. At #2, we provide one method that will create our individual session objects. The methods of interfaces extending `Remote` all may throw a `java.rmi.RemoteException`, so we must include that in the method definition.

Listing 6.13 The ISession interface

```
public interface ISession extends Remote                           #1
{
  public MyReturnObject doStuff(MyArgumentObject inputObject)      #2
    throws MyException, RemoteException;                           #2
}
```

#1 Declare the session interface
#2 Method that performs some session-specific activity

The code is similar to `ISessionFactory`. At #1, we once again define an interface designed to represent a server-side object. At #2, we define a method that uses object arguments. The rule is that all of these object arguments, **including the exception classes**, must be serializable. This condition is usually satisfied automatically by Java by having the object classes implement `java.io.Serializable`.

Next, let's have a look at the implementations for these interfaces. In a system where RMI is being used for isolation, the implementations would have access to the third party library or libraries that you were trying to isolate. This example, of course, has no way of demonstrating that, so you will need to use your imagination. Listings 6.14 and 6.15 show the implementation classes.

Listing 6.14 The session factory implementation class

```
public class SessionFactoryImpl extends UnicastRemoteObject                #1
  implements ISessionFactory                                               #1
{
  public SessionFactoryImpl()                                              #2
    throws RemoteException                                                 #2
  {
    super(0,new RMILocalClientSocketFactory(),                             #3
      new RMILocalSocketFactory());                                        #3
  }

  public ISession make()                                                   #4
    throws RemoteException                                                 #4
  {                                                                        #4
    return new SessionImpl();                                              #4
  }                                                                        #4
}
```

#1 Declare a class that implements ISessionFactory
#2 The class needs a constructor
#3 Control the socket binding to be localhost only
#4 Factory method instantiates the session

The class declaration at #1 must extend `java.rmi.server.UnicastRemoteObject`, as well as implementing the interface. At #2, we provide a constructor, which must be called only within the server process. At #3, we provide our own socket factories, which limit the socket bindings to localhost as a security measure. It is always good practice to not leave sockets available for external access if one can possibly avoid it. Finally, at #4, our actual method does nothing more than instantiate the session implementation class.

Listing 6.15 The session implementation class

```
public class SessionImpl extends UnicastRemoteObject                      #1
  implements ISession                                                      #1
{
  public SessionImpl()
    throws RemoteException
  {
    super(0,new RMILocalClientSocketFactory(),new RMILocalSocketFactory());
  }

  public MyReturnObject doStuff(MyArgumentObject inputObject)              #2
    throws MyException, RemoteException                                    #2
  {
    if (inputObject.getField2() == 0)                                      #3
      throw new MyException("Hey, you sent in a zero!");                   #3
                                                                           #3
```

```
    System.out.println("The server process got a string of '"+          #3
      inputObject.getField1()+"'");                                     #3
                                                                        #3
    return new MyReturnObject("This is my response");                   #3
  }
}
```

#1 Declare implementation class to implement ISession
#2 Our method can use any serializable paper object
#3 Implementation of the method is straightforward

At #1, we declare the class to implement `ISession`, and extend `UnicastRemoteObject`. At #2, the method signature requires all helper classes to be available in both the client and server processes, and all of them must be serializable. The method implementation itself, at #3, is very straightforward.

The processes themselves are worth looking at, because it is here that the infrastructure is created which allows RMI to do its job. Listing 6.16 shows the registry process main class.

Listing 6.16 The registry process main class

```
public class RegistryProcess
{
  private RegistryProcess()
  {
  }

  public static void main(String[] argv)
  {
    try
    {
      java.rmi.registry.Registry r =                                    #1
        java.rmi.registry.LocateRegistry.createRegistry(8401,           #1
        new RMILocalClientSocketFactory(),new RMILocalSocketFactory()); #1

      System.out.println("Registry started and is awaiting connections.");
      while (true)
      {
        Thread.sleep(600000L);
      }
    }
    catch (InterruptedException e)
    {
    }
    catch (RemoteException er)
    {
      System.err.println("Remote exception in registry main: " + er);
      er.printStackTrace(System.err);
    }
  }
}
```

#1 Create the registry

All of the important stuff happens at #1. Here, the registry is created and started. I've hardwired the registry port to a specific value (8401); you will need to do the same in order

for the registry to be located by the server and client processes. Once again, for security reasons we make sure to bind the registry process sockets only to localhost.

Listing 6.17 shows the server process. In this process, the factory implementation class instance is created and registered with the registry process.

Listing 6.17 The server process main class

```
public class ServerProcess
{
  private ServerProcess()
  {
  }

  public static void main(String[] argv)
  {
    try
    {
      SessionFactoryImpl factory = new SessionFactoryImpl();              #1

      Naming.rebind("//127.0.0.1:8401/session_factory", factory);         #2

      System.out.println("Server started and is awaiting connections.");
      while (true)
      {
        Thread.sleep(600000L);
      }
    }
    catch (InterruptedException e)
    {
    }
    catch (RemoteException er)
    {
      System.err.println("Remote exception in server process: " + er);
      er.printStackTrace(System.err);
    }
    catch (MalformedURLException er)
    {
      System.err.println("Exception in server process: " + er);
      er.printStackTrace(System.err);
    }
  }
}
```

#1 Create one instance of the factory
#2 Register the factory class instance

At #1, we create the instance of the factory implementation class that we intend to register with the registry service. Then, at #2, we register it, making sure to use the proper registry process port, and giving the factory class instance a well-known name that we'll use to look it up in the client process.

Listing 6.18 shows that client process, and how the lookup actually works.

Listing 6.18 The client process main class

```
public class ClientProcess
```

```
{
  private ClientProcess()
  {
  }

  public static void main(String[] argv)
  {
    try
    {
      System.out.println("Obtaining a factory handle...");

      ISessionFactory factory = (ISessionFactory)                          #1
        Naming.lookup("rmi://127.0.0.1:8401/session_factory");             #1

      System.out.println("Getting a session handle...");

      ISession newSession = factory.make();                                #2

      System.out.println("Calling remote method...");

      MyReturnObject result = newSession.doStuff(                          #3
        new MyArgumentObject("hello",123));                                #3

      System.out.println("Result: "+result.getValue());
      System.out.println("Done!");
    }
    catch (Exception e)
    {
      e.printStackTrace(System.err);
      System.exit(2);
    }
  }
}
```

#1 Use the well-known name to find the factory
#2 Use the factory to create the session
#3 Use the session to do something

At #1, we communicate with the registry process, which is listening on port 8401, and ask it to find the registered object called `session_factory`. Then, at #2, we use the factory object to create a session implementation object on the server, and a corresponding stub object on the client. We use the session object to perform some activity at #3.

> **Note** When you build an RMI-based connector, it will probably be necessary to separate the building of the isolation classes from the common classes and the connector classes. This is because the errant third-party library is likely to be in direct conflict with ManifoldCF classes, and should thus be compiled only against the isolation classes. The example's build.xml file demonstrates one way of doing this.

USING A CUSTOM CLASS LOADER FOR ISOLATION

A third-party-library isolation alternative to RMI is to use a class loader. Class loader-based isolation requires no socket communication, but it does require similar discipline in

separating implementation classes from interface classes. The fundamental requirement is that the third party library is never accessed from within the connector; the implementation class must be the only way to get at it. From within the main connector code, the isolated implementation classes must be instantiated by means of Java reflection. See figure 6.7.

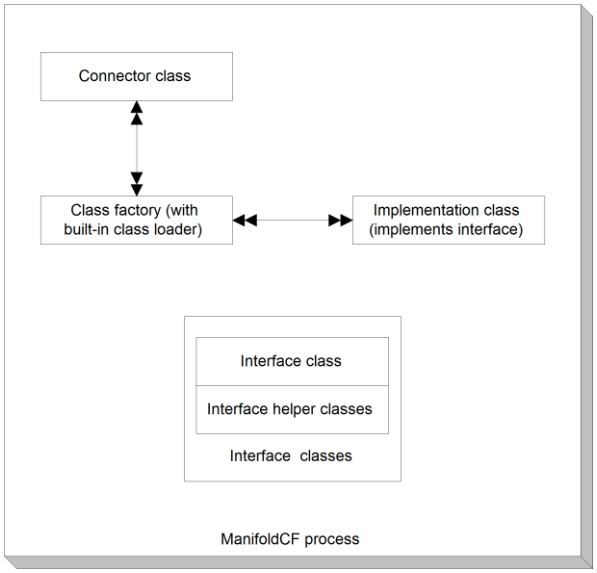

Figure 6.7 Using a class loader to isolate an implementation class, and indirectly the third-party library that the implementation class interacts with.

In the class loader isolation model, we once again have two kinds of classes – the common, simple, paper-object classes, and the bifurcated classes and interfaces which isolate the problematic library. But instead of stubs and skeletons, the ManifoldCF process deals with direct references to instances of the implementation classes.

The very same model that we used for RMI, with a single session factory and multiple sessions, works quite well when a custom class loader is involved. If this model is used, it is only necessary to instantiate the session factory class using the custom class loader by means of reflection. As in RMI, all subsequent dependencies of the factory class will automatically be instantiated using the correct class loader. We'll see exactly how this works in our example.

A CUSTOM CLASS LOADER EXAMPLE

Since there are currently no known ManifoldCF connectors that use a custom class loader for isolation, I've put together a short example of such an architecture, which is identical in every way to the RMI example except for the isolation technology.

Indeed, because of the similarity of the two architectures, I've placed the sources side-by-side in the same area of svn, and allowed them to share the helper classes and ant build file. Only the ant target is different: run-classloader-example. If you try this ant target, you should see the following:

```
run-classloader-example:
    [mkdir] Created dir:
C:\wip\mcfia\examples\edition_2_revised\ground_rules_example\build\connecto
rarea
     [copy] Copying 1 file to
C:\wip\mcfia\examples\edition_2_revised\ground_rules_example\build\connecto
rarea
     [java] Configuration file successfully read
     [java] Obtaining a factory handle...
     [java] Getting an isolated session handle...
     [java] Calling isolated method...
     [java] The isolated class got a string of 'hello'
     [java] Result: This is my response
     [java] Done!
```

We know that the example is working because the implementation classes are not present in the original class-path. The example uses a `properties.xml` configuration parameter to provide a connector-specific area for the jars that run in the isolation area, and the implementation classes are only included there. Thus, without the proper class loader being used, the program would not be able to function. See listing 6.19.

Listing 6.19 Class loader example main class

```
public class ClassLoaderExample
{
  private ClassLoaderExample()
  {
  }

  public static void main(String[] argv)
  {
    try
    {
      ManifoldCF.initializeEnvironment();                                  #1

      File connectorLibraryFolder = ManifoldCF.getFileProperty(            #2
        "org.apache.manifoldcf.examples.connectorjarpath");                #2
      if (connectorLibraryFolder == null)                                  #2
        throw new ManifoldCFException(                                     #2
          "Must supply the "+                                              #2
          "org.apache.manifoldcf.examples.connectorjarpath "+              #2
          "property!");                                                    #2

      ManifoldCFResourceLoader connectorLoader =                           #3
        ManifoldCF.createResourceLoader();                                 #3

      ArrayList libList = new ArrayList();                                 #4
```

```
        libList.add(connectorLibraryFolder);                                #4
        connectorLoader.setClassPath(libList);                              #4

        System.out.println("Obtaining a factory handle...");

        Class factoryClass = connectorLoader.findClass(                     #5
          "org.apache.manifoldcf.examples.CLSessionFactoryImpl");           #5
        ICLSessionFactory factory = (ICLSessionFactory)                     #5
          factoryClass.newInstance();                                       #5

        System.out.println("Getting an isolated session handle...");

        ICLSession newSession = factory.make();                             #6

        System.out.println("Calling isolated method...");
        MyReturnObject result = newSession.doStuff(                         #7
          new MyArgumentObject("hello",123));                               #7
        System.out.println("Result: "+result.getValue());
        System.out.println("Done!");
      }
      catch (Exception e)
      {
        e.printStackTrace(System.err);
        System.exit(2);
      }
    }
  }
```

#1 Initialize the environment
#2 Get the file property containing the isolation directory
#3 Create a ManifoldCF resource loader
#4 Initialize the resource loader with the isolation directory
#5 Use the resource loader to instantiate factory class
#6 Use the factory class to instantiate session
#7 Use the session class normally

Since we're using the ManifoldCF core framework in this example, we must be sure to initialize the environment at #1. We obtain a file/directory parameter describing the directory from which the isolation jars are loaded from `properties.xml` at #2. This directory is the location where the jar containing the isolation classes has been placed. At #3, we use the ManifoldCF infrastructure to build a resource loader that is derived from the ManifoldCF core resource loader, so that the class loader hierarchy is complete. At #4, we add the isolation directory to the new resource loader instance. We use the new instance at #5 to instantiate the session factory implementation class. From there on, it's easy; at #6 we use the session factory to instantiate a session instance, and we use the session instance at #7 to actually do something.

There are a few differences between the interfaces and implementations for the class loader example, and for the RMI example. The interfaces in the class loader example do not need to extend `java.rmi.Remote`, for instance, nor do the implementation classes need to be derived from `java.rmi.server.UnicastRemoteObject`. Otherwise, they are essentially identical. Similarly, separate targets in the build process are also needed, and the considerations identical as well.

WHEN YOU SHOULD USE ISOLATION

Isolation is clearly not something you would want to undertake lightly. So, when do you need to do it? What are the rules?

One general rule is that you **must** use RMI for your connector if your connector requires a native library. This is because native code can potentially cause a process to fail with a fatal native exception, e.g. a SEGFAULT, or possibly trounce Java's working memory, or any number of other nasty possibilities. A badly behaved native library can also leak resources, and has myriads of ways of bringing down the Java process it is running within. Finally, it is not possible in Java to use a class loader to isolate native library loads; such loads are global for the process, so there could in theory be conflicts that cannot be resolved.

If no native code is involved, then you still must use some kind of isolation technology if your connector requires a specific version of a Java jar which differs from one that is included in ManifoldCF's main `lib` directory. But in this case you have a choice; you may choose either RMI or a custom class loader for your isolation technology. As we have seen, the requirements for how you would code your connector for either isolation method are pretty similar.

The advantage to the custom class loader approach is that there is no need to start up sidecar processes. Also since there are no sockets used, there is no need for transient thread creation. But this is offset because, very typically, the third-party library uses sockets itself, so transient threads may be needed for that reason, regardless.

The advantage for RMI is that it's easy to specify the precise environment your third-party library requires. This may include loading of native libraries, precise control of the class path, and also the ability to guarantee the proper settings for various environment variables. But it's also more work to code, and because of the class serialization required, it may function more slowly as well.

It would be wonderful to be able to say that, with third-party-library isolation techniques, we were done with learning connector ground rules. But not quite – there is still one rule we need to talk about, and that is making sure your connector's memory consumption is bounded. That's next.

6.3.5 Bounding memory consumption

I'm sure you've run into a Java program at one point or another which seems to never be happy when it comes to the memory it is given. You might set it up one day, and run it the next on a slightly more involved task, only to discover that it runs out of memory. You can fix this by giving the Java process more memory, but sooner or later you know that it will happen again.

What's going on in such cases is that the memory usage of the process is not *bounded*. Any process where the memory consumption depends sensitively on the size of the task, or the largest document, or any such external operational parameter, falls into this class. While a program that has unbounded memory consumption may be an inconvenience when run

occasionally, for a project like ManifoldCF, which is meant to run continuously, such a flaw is deadly, and must be avoided at all costs. Getting calls in the middle of the night may happen once and a while if your log messages are obscure, but if your connector causes ManifoldCF to run out of memory, you can count on a not-very-happy call every time that it happens.

This book cannot possibly cover all ways that it is possible to do bad things where memory consumption is concerned. However, I'm certainly willing to drop hints. Table 6.8 describes some common situations, and the recommended way to avoid them.

Table 6.8 Common memory over-use situations, and how to avoid them

Situation	Recommendation
I need to access content more than once	Stream the content to a temporary file first
I need to parse an arbitrarily large body of XML	Use the Java SAX XML parsing API, never the DOM API
My third-party library provides a method to get document content as a byte array	Don't use it; find a way to access the content as a stream instead
Want to cache stuff	Use the ManifoldCF cache manager, and make sure your cached stuff is LRU limited
I have an open-ended data structure	Consider using a connector-specific database table instead

As you can see, there's almost always a way to structure things so that you can accomplish your goals without writing a connector that is unbounded in memory. That's a good thing, because your sleep is valuable.

6.4 Summary

In this chapter, we've learned a great deal. We've learned about how connector class instance pooling works, how to write connector UI components, how to use a local trust store, and how to deal with many common connector problems, such as the need for isolation of third-party libraries. Finally, we learned a little bit about how to be sure a connector could exist in a bounded amount of memory.

In the next chapter, we finally begin exploring how to write actual connectors. We begin will begin with writing a repository connector, using an example repository engine as our example.