3

Integration using the API

This chapter covers

- Understanding JSON, and REST APIs in general
- Understanding typical use cases for the ManifoldCF API
- Using Apache Httpcomponents Httpclient to communicate with the API
- Using the ManifoldCF `Configuration` and `ConfigurationNode` classes
- Building an example application which relies on the API

Here we will explore in detail the uses and the usage of the ManifoldCF API. I introduce the general features of the API, and afterwards I will go on to develop an example based on the RSS connector. We'll make the example as interesting as time and space permit, in the hopes that you might find it genuinely useful as a basis for any ManifoldCF integration task that you might want to undertake.

3.1 Understanding the problem

The next time you talk, you learn that Bert has made great progress in his attempt to use ManifoldCF to get his content into his search engine. He has created connection definitions and job definitions all over the place, and things are working pretty well. However, he has now become a victim of his own success. His manager, although certainly impressed, does not trust Bert's setup to run without flaw indefinitely, and he wants to know immediately when something goes wrong.

Bert has wired up software that presents a big red blinking light on the manager's screen if Bert signals it to. But this requires Bert himself to keep track of the state of ManifoldCF, twenty-four hours a day and seven days a week. Bert is finding this tiring. Plus, his fingers

are getting mighty sore from clicking the `Refresh` link on the crawler UI's job status screen all the time. He wants to know if there is a good programmatic way of doing the same thing, so he can write some code, go home, and take a shower.

Indeed, as you tell Bert, there is a ManifoldCF API designed specifically for applications such as his. The API includes operations which pretty much would allow a programmer to write his or her very own user interface, if they so wished, including connection definition management, job definition creation, and job status. ManifoldCF's API has been rigorously designed to adhere to the unwritten but well-understood standards of REST. REST APIs are built upon HTTP, and all objects are uniquely described by a URL. This allows HTTP verbs such as GET, POST, PUT, and DELETE, to be used to work with objects covered by the API.

In this chapter, we will build an automated system that sets up and manages a continuous RSS news crawl using the ManifoldCF API. As you are no doubt aware, many news sites have public RSS feeds for the purpose of directing their users at timely content. We want our system to keep our search engine up to date with the latest news from some set of feeds that we have managed to gather. Along the way, we'll have opportunity to use almost every aspect of the ManifoldCF API. We'll learn how to create connection definitions, test them, create job definitions, and run them. We'll also learn how to check on a running job, and take corrective action should something go wrong – all using the API.

3.1.1 How to perform an RSS news feed crawl

Let's start by figuring out how we'd set up our crawl in the absence of any automation. Then, armed with that knowledge, we'll be much more able to design our application.

CRAWLING MODEL

The first obvious choice we need to make is what ManifoldCF crawling model to use. As we learned in Chapter 2, there are two models to choose from: repeated incremental crawling, or continuous crawling. The repeated crawling model has a problem if we want to keep our search engine up to date with news, namely that each cycle must complete before the next cycle begins. That means that some content must wait the entire length of a cycle before it will be fetched. Continuous crawling mode, where documents are refetched at intervals that depend on how frequently they have changed in the past, would seem like a more natural fit for crawling RSS feeds and their referenced documents, since each document, including the feeds themselves, runs on their own schedule during such crawls. So clearly our RSS job will need to use continuous crawling if it is going meet our requirements.

EXPIRATION VERSUS REFETCH

Pure continuous crawling involves a great deal of overhead, since the same document will be fetched multiple times. For news documents, it is also the case that they typically "change" each time they are fetched, but not because of actual content changes. Instead, it's the navigation and advertising clutter (or "chrome") that changes from fetch to fetch. Those changes nevertheless wreak havoc with ManifoldCF's change frequency statistics.

But since our use case involves only documents referenced by timely RSS feeds, there may be a better way. Instead of re-fetching documents, all documents (other than the feeds

themselves) can be expired some period of time since they were authored or fetched. This works fine because, for news, we are not interested in older content.

MINIMUM FEED REFETCH TIME

Many feeds include a *time to live* (TTL) value, which the RSS connector knows how to interpret, and will use to schedule the next fetch of the feed. The implication is that a crawler can safely re-fetch the feed based on the TTL value. So, what is the problem with that?

Unfortunately, a feed's TTL value often turns out to be too small. If just every feed's TTL value were used, the job could well spend all of its time fetching feeds, and no time fetching the actual documents. This is unnecessary, because feeds usually don't change that frequently. I have found that it is usually counterproductive to fetch them more than once every 15 minutes or so. But luckily, you can specify a minimum feed re-fetch interval in your RSS job, so as long as we set that to 15 minutes, we should be good to go.

NEWS CRAWL PARAMETER SUMMARY

A successful RSS news job therefore will have the following list of characteristics:

- A document re-fetch time set to infinity
- A document expire time set to some value that is meaningful for the level of currency we want, e.g. two weeks or a month
- An RSS feed minimum re-fetch interval of 15 minutes or more

Now that we understand how to perform a crawl of news RSS feeds, let's go ahead and write down what we expect the application to actually do.

3.1.2 Example application requirements

Our application has really needs to do three basic things: Set up the crawl, monitor it, and update it when needed. Let's clarify what each of these tasks really entails.

SETTING UP THE CRAWL

To set up the crawl, let's first stipulate that the output connection definition must be already set up by the user via the UI. This is not just a convenience; it may make our example broader by allowing you to use the same application with more than one kind of search engine.

On the other hand, our application will need to take care of setting up the RSS repository connection definition, with the appropriate throttling, as well as creating the job, and starting it.

MONITORING THE CRAWL

The monitoring part of the task needs to check the job for errors, report them if found, and restart the job if it has stopped. To add some spice to the application, it might be fun to have it raise the alarm if restarting the job a couple of times does not result in the job staying up for a reasonable period of time.

UPDATING THE JOB

From time to time the user of the application may discover new feeds, or remove discontinued ones. We will want to build in some mechanism that allows him to do this. Perhaps there should be a file that contains the urls of all the feeds, which the application periodically uses to update the running job.

REQUIREMENT SUMMARY

To summarize the application requirements, the application should:

- Tie into an existing output connection definition
- Create the RSS connection definition
- Create the RSS job definition, and start the job
- Update the job definition with feed changes
- Monitor the job, and restart it if there is a problem
- Raise the alarm if the job repeatedly aborts for the same reason

This is a pretty complete summary of the responsibilities of the proposed application. But we've forgotten something very important – namely, writing down how we intend to test this application. We'll do that next.

3.1.3 Planning for testing

An application is only going to be realistically usable if it is also testable. So, in general, it is good practice to think about how to write automated tests for our application before we get too far along in the planning process.

UNIT TESTS

Tests can come in two rather different flavors. The first kind of test is called a unit test, and it consists basically of test code that exercises each module within the application. The goal of a unit test is to verify that all of an application's modules do what they are each intended to do.

For the proposed RSS news feed application, we'll want unit tests that confirm that each basic API-based task actually works as designed. That is, we'll want unit tests that check out the code that creates the RSS connection definition, the code that creates the RSS job definition, and the code that updates the RSS job with new feeds. We'll also want tests that confirm that we can properly interrogate the status of the job.

END-TO-END TESTS

A second kind of test we should have is an end-to-end test. In this kind of test, we run the entire system (application, ManifoldCF, and the search engine), and perturb it in ways that are designed to exercise the application's logic. The test then notes the results, and verifies whether or not the application is performing as designed.

For this application, the biggest hurdle for an end-to-end test would probably be in simulating the application's error handling pathway. Somehow we will need to introduce errors into the crawl that cause the job to abort. Since the connector itself has a good

amount of logic built into it for handling errors, creating such a real-life error would not be straightforward at all. We find ourselves needing to write a test for a case that we hope will never occur, and that we are not certain actually **can** occur. Unfortunately, this kind of testing situation occurs with distressing frequency when complex systems are involved.

The only reliable solution is therefore likely to be a bit of a cheat. The trick is to realize that testing the pathway in the application can be approximated by having the test abort the job. In ManifoldCF, a manual abort and an abort due to an error are very similar; the job enters the same state after either. Let's therefore plan on using this technique as part of our testing strategy.

The remainder of the test would be quite straightforward. A basic output connection definition should be created by the test using the ManifoldCF API, and the application should be started. If all goes well, the job will shortly enter the "running" state, and stay there unless perturbed. From that point forward, we can make the test as elaborate was we feel is necessary.

With the test plan, we're now ready to begin designing our application in earnest.

3.2 Designing the application

The process of designing an application that uses ManifoldCF's API does not differ from good practices used to design any software application. The goal of the design process is fundamentally to think through the details of how the application will function, before the code is actually written. There are many schools of thought as to the best way to go about this task, but I'm going to take a minimalist approach; I'm going to start with what the API offers that seems to fit, and then think through the flow of the application in relation to those API methods. But first, we'll need to learn a little something about the structure of the ManifoldCF API.

3.2.1 API structure

The ManifoldCF API is a REST-style API. That means that every entity in the API is described by a URL, and the four HTTP verbs (GET, POST, PUT, and DELETE) are used in conjunction with these URLs to select the proper functionality. So, for example, to delete a connection definition, you would use the DELETE verb in conjunction with the connection definition's URL.

The form of the data transferred to and from the API for every operation is encoded using a format known as JSON. JSON, or "Javascript Simple Object Notation", has become a de-facto standard for describing hierarchically-structured objects. JSON's claim to fame is that it is truly simple, and people have not yet tried to overload its simple syntax with complex constructs.

Why the ManifoldCF API uses JSON and not XML

> Besides JSON, the only other standard way of describing hierarchical objects is by using XML. XML has been used as the fundamental language of SOAP and Web Services, and who-knows-how-many custom hierarchical specifications. This makes using XML for implementing APIs fraught with semantic minefields, because there is always the temptation to view an XML object as meaning more than what has been actually specified, and thus the question of "which flavor of XML is it" continually arises. So, for the moment, ManifoldCF does not use XML for its API.

Bert's been following along with all of this, and he's got some thoughts. First, he wants to know why the API isn't just a Web Service API using SOAP. He's really used to SOAP at this point, although he readily admits that SOAP is not actually very simple, despite the fact that the "S" in SOAP stands for "Simple".

Web services, you agree, are not simple. Even today, a complete implementation of all possible aspects of a web service is difficult to find in many languages. On the other hand, HTTP is very well supported everywhere, and JSON is easy enough that implementations abound.

Nevertheless, there's still a role in ManifoldCF for basic XML. It is pretty easy to map from one hierarchical representation to another. This turns out to be extremely useful as far as ManifoldCF's API is concerned. We'll talk about that further in a moment.

OBJECTS IN JSON

A JSON object is always wrapped in curly braces, and consists of a set of name and value pairs, as shown here.

```
{
  "name_1":"simple_value_1",
  "name_2":"simple_value_2",
  "name_3":{
    "nested_name_1":"nested_value_1",
    "nested_name_2":"nested_value_2"
    }
  "name_4":[
    "array_value_1",
    "array_value_2"
    ]
}
```

As you can see, values can be nested hierarchically. Each value may be a simple value, another JSON object, an array of simple values, or an array of JSON objects. Whitespace and newline characters are ignored (which allows me to present the pretty example above). You can read more about JSON at http://www.json.org .

CHARACTER ENCODING

The characters that form JSON strings for names and values above are limited only by the encoding in which the characters are presented. Special characters, such as quotation marks, may be included in each JSON string so long as they are encoded properly according to the JSON specification. For example, `she said "go left"` would be turned into the JSON string `"she said \"go left\""`.

The ManifoldCF API uses utf-8 exclusively as its character encoding, which ensures that all Unicode characters are properly represented and can therefore appear within the JSON objects that the API uses.

JSON CONSTRUCTION AND PARSING

If you find that you need to build the strings for JSON objects directly, or parse JSON object strings yourself, you can almost certainly find an open-source package that does an excellent job. For Java, ManifoldCF uses the sample code from http://www.json.org, which it packages into a jar called `json.jar`, which you are certainly entitled to use. For Python, a package called `SimpleJSON` is widely used. I highly recommend using one such package, rather than creating your own, since they are all pretty lightweight and their licenses are typically compatible with most commercial projects as well as other open-source projects.

Okay, now we understand the API's structure. Next, we will need to learn what we can do with it.

3.2.2 API functions

The API's verbs and objects correspond directly to most basic operations that you might have done yourself in the crawler UI. For example, there is an object that corresponds to each output connection definition, and a URL that describes it. The HTTP verbs GET and PUT can be used with this URL to load or save the connection definition.

Table 3.1 lists the subset of REST API resources and HTTP operations that seem applicable to our proposed application.

Table 3.1: Useful ManifoldCF API resources and HTTP operations

Resource	Operation	Function
`status/outputconnections/`*<encoded_connection_name>*	GET	Get the status of an output connection definition; output is a connection status or error object
`repositoryconnections/`*<encoded_connection_name>*	GET	Get a repository connection definition; output is a repository connection object
`repositoryconnections/`*<encoded_connection_name>*	PUT	Save or create a repository connection definition; input is a repository connection object, output is

		empty or an error object
`jobs`	GET	Get a list of all jobs definitions; output is an array of job objects, or an error object
`jobs`	POST	Create a job definition; input is a job object, output is job identifier object, or an error object
`jobs/`*<job_id>*	GET	Get a specific job definition; output is empty, or a job object, or an error object
`jobs/`*<job_id>*	PUT	Update a specific job definition; input is a job object, output is empty or an error object
`jobstatuses/`*<job_id>*	GET	Get the job status for a specific job; output is empty, or a job status object, or an error object
`start/`*<job_id>*	PUT	Start a specific job; input is empty, output is empty or an error object
`startminimal/`*<job_id>*	PUT	Start a minimum run of a specific job; input is empty, output is empty or an error object

JSON INPUT AND OUTPUT FORMAT

It may have occurred to you that a description of the above operations is not complete unless we understand what the input and output JSON for each operation looks like. I've therefore summarized the input and output JSON format for them in table 3.2.

Table 3.2: JSON formats for status, connection, and job objects

Object type	JSON format
Empty	`{ }`
Error	`{"error":<error_text>}`
Connection status	`{"check result":<message_text>}`
Repository connection	`{"repositoryconnection":<repository_connection_object>}`
Job	`{"job":<job_object>}`
Job status	`{"jobstatus":<job_status_object>}`
Job identifier	`{"jobid":<job_identifier>}`

As you can see, for each of the objects in table 3.2, there is also a subfield which needs description. It is set up this way because it needs to be possible for a client program to look at a response and determine at a glance whether the response is indeed the expected object, or is instead an error.

Many of the subfields are basic JSON strings. This is the case for *error_text*, *message_text*, and *job_identifier*. But that still leaves *repository_connection_object*, *job_object*, and *job_status_object*, which are more complex JSON objects that must be described in their own right.

JOB STATUS OBJECTS

Some of these complex JSON objects are straightforward to describe. Table 3.3 describes the JSON fields that comprise each *job_status_object* above.

Table 3.3: JSON fields in job status objects

Field	Meaning
`job_id`	The job identifier
`status`	The job status, having the possible values `not yet run`, `running`, `paused`, `done`, `waiting`, `starting up`, `cleaning up`, `error`, `aborting`, `restarting`, `running no connector`, or `terminating`

`error_text`	The error text, if the status is `error`
`start_time`	The job start time, in milliseconds since Jan 1, 1970
`end_time`	The job start time, in milliseconds since Jan 1, 1970
`documents_in_queue`	The total number of documents in the queue for the job
`documents_outstanding`	The number of documents for the job that are currently considered 'active'
`documents_processed`	The number of documents that in the queue for the job that have been processed at least once

There are no surprises here. As you can see, the Job Status object corresponds directly to the fields displayed in the Crawler UI's Job Status page, described back in Chapter 2.

REPOSITORY CONNECTION DEFINITION AND JOB DEFINITION OBJECTS

That leaves *repository_connection_object* and *job_object*. But, for these kinds of objects, there is a wrinkle: the form of the object depends on the kind of repository connector we are dealing with. RSS repository connection definition objects, therefore, have a different format than (say) web repository connection definition objects. Job definition objects also have a format that is dependent on connection type, and it is even more variable because a job definition object's format depends on **both** the job's output connection definition, as well as the job's repository connection definition.

Knowing this, I could nonetheless list all the JSON object forms, or better yet, refer you to the online documentation for the ManifoldCF API (which can be found at http://manifoldcf.apache.org/releases/trunk/en_US/programmatic-operation.html). But perhaps a better way to learn about the actual JSON for these objects would be to actually use the API to obtain them, and see what they look like. That is the approach we're going to take later in this chapter, when it comes time to write the application code. For now, though, we have accomplished our goal: we already know enough about the structure of the API to move on to thinking about the structure of the application as a whole.

3.2.3 Application structure

So, how should the application work? We've enumerated what it basically needs to do, but now we're going to want to dive into the details. Figure 3.1 shows an application flowchart that basically captures the requirements as we have stated them.

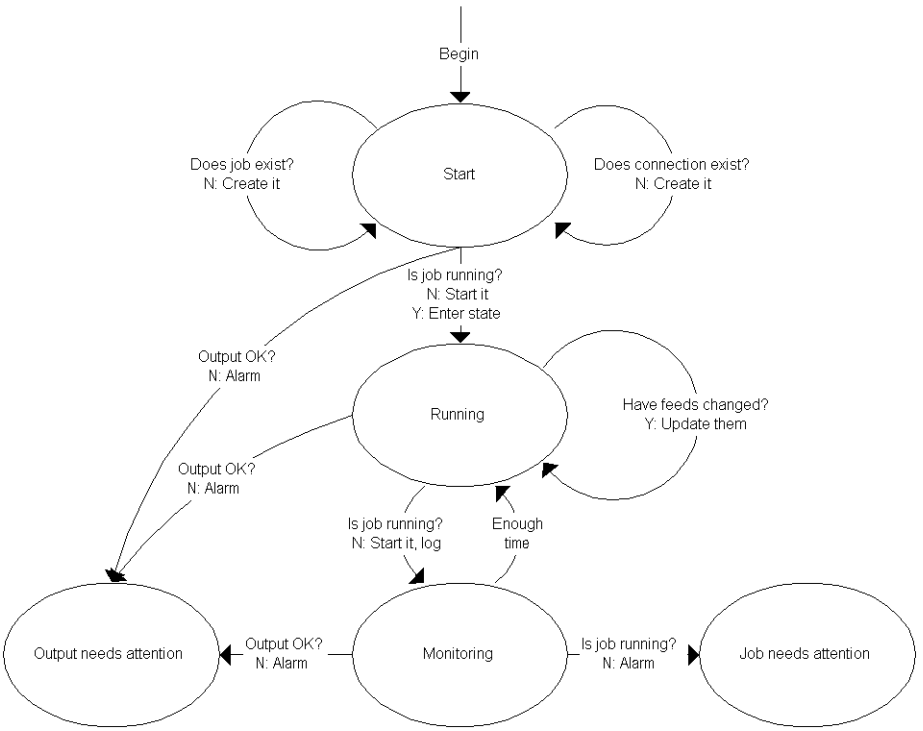

Figure 3.1: Flow chart for RSS crawl monitoring example application. There is one initial state and there are two final states. Errors that occur during communication with the API are not represented in this diagram, nor are reset links from needs-attention nodes back to Start state or Running state.

As far as the application code itself is concerned, we have two choices. One choice is to build in explicit conditional logic for each decision. This approach will produce code that is quite flexible, but also rather wordy and repetitive. Each state and each transition will look more or less the same, with changes in small details. Certain logic will wind up being duplicated for each – exception handling code, for instance, will be everywhere. Even more to the point, such an application cannot be readily repurposed if the flowchart were to change without major work.

The other path is to build a more general structure, namely a *finite state machine* (FSM), that we can "program" to represent the states and transitions implied by our requirements. Better yet, since our application flow chart already looks a whole lot like a FSM, if we adopt the FSM approach we've already pretty much designed the application!

Bert is thrilled that you have taken his problem seriously, and he wholeheartedly agrees with the requirements you've stated for the solution. But what he doesn't get is why you

want to implement it as a finite state machine. Bert's perfectly happy at this point to just throw together something that does what is needed in linear form. So why spend the time and effort to build something more?

What Bert doesn't see is that a finite state machine often has less code than the equivalent straight-line approach. This is because common code is more efficiently shared. It is also more flexible. Heaven forbid that Bert's manager decides he also needs a yellow light on his desk, in addition to the red! Since Bert is not entirely clear how things will evolve with over time, the extra flexibility that a finite state machine grants may well be more useful than Bert realizes.

Now that we've come up with a preliminary design, we are ready to start the process of putting together the code. In the next section, we'll begin by writing what we need to perform HTTP transactions against the API.

3.3 Communicating with the API

In this section, we'll learn about all the nuts-and-bolts concepts necessary to communicate effectively with ManifoldCF's API. We'll write code to use Httpcomponents Httpclient to perform the API transactions. We'll also learn about the classes that ManifoldCF itself uses to form and decode JSON objects.

3.3.1 Using httpcomponents httpclient

Any REST API, ManifoldCF's included, uses HTTP as its basic communication protocol. The Apache Httpcomponents Httpclient HTTP project works well for communicating with the ManifoldCF API. In this section, I will demonstrate one way to use this client library to set up such communication. I will show how to use the Apache Httpcomponents Httpclient to perform each of the four basic HTTP operations required by the ManifoldCF REST API.

HTTP CONNECTION COMPLEXITIES

There are, of course, many ways in which a straightforward HTTP communication can be made more complicated, such as by the use of SSL, or by the use of various authentication protocols (both open and proprietary forms). My example will not use any of these, but on the oft chance that your actual usage might, I've chosen to use the 4.3.x release of Httpcomponents Httpclient.

In addition to the possible need for authentication, there are many supported ways of using Httpcomponents Httpclient to set up HTTP connections and tear them down. The Httpcomponents Httpclient package includes support for various HTTP connection pools and other means of making connections more efficient. However, my example will use none of these. It is not the focus of this chapter to explore the capabilities of Httpcomponents Httpclient, but instead to demonstrate ManifoldCF API usage in a straightforward manner that will work for most people.

The actual setup of an HTTP connection, and its subsequent usage, involves creating or obtaining an HttpClient object, building a method, and executing that method. Code-wise, that would be something like the following:

```
import org.apache.http.impl.client.HttpClients;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.*;
import org.apache.http.HttpResponse;

...

  HttpClient client = HttpClients.createDefault();
  HttpXXX method = new HttpXXX(resourceURL);
...
  HttpResponse response = client.execute (method);
  int responseCode = response.getStatusLine().getStatusCode();
  InputStream responseData = response.getEntity().getContent();
...
```

The `execute` invocation is where the HTTP transaction actually takes place. Everything prior to that is setup, and everything afterwards is data interpretation. The "XXX" is a placeholder for one of the four HTTP method types – Get, Post, Delete, or Put. We'll look at the precise form of each of the four types next.

GET TRANSACTIONS

GET transactions in REST APIs are used for getting the current state of the object specified by the URL in question. The code for a GET transaction therefore needs to send nothing, but receive a response code and a response. The response itself, if it comes from the ManifoldCF API, will always be encoded using UTF-8, so it is safe to decode based on that assumption.

A Java method that will do what is needed follows in listing 3.1.

Listing 3.1 A method using HttpComponents HttpClient to perform a GET operation

```
public String performAPIRawGetOperation(String url)
  throws IOException
{
  HttpClient client = HttpClients.createDefault();
  HttpGet method = new HttpGet(url);                                    #1
  HttpResponse httpResponse = client.execute(method);                   #1
  int response = httpResponse.getStatusLine().getStatusCode();          #2
  String responseString = convertToString(httpResponse);                #2
  if (response != HttpStatus.SC_OK &&                                   #3
    response != HttpStatus.SC_NOT_FOUND)                                #3
    throw new IOException("API http GET error; expected "+              #3
      HttpStatus.SC_OK+" or "+HttpStatus.SC_NOT_FOUND+", "+             #3
      " saw "+Integer.toString(response)+": "+responseString);          #3
  return responseString;
}
```

#1 Use httpclient's GET method
#2 Assume the response is utf-8 encoded
#3 Response codes 200 (OK) and 404 (NOT_FOUND) are acceptable

At #1, we create an `HttpGet` object, which we then execute. At #2, we retrieve the response and convert it to a string, using a common method to do that. We check for valid response codes at #3. (In case you were concerned, for a REST GET transaction, an HTTP

NOT_FOUND is indeed a valid response with a specific meaning: it means that the object in question does not exist.)

This is almost too straightforward. Indeed, there are a few hidden issues with the code above that you might want to keep in the back of your mind. First, as we alluded to earlier, the connection is not being pooled. This can be a factor in performance whenever the setup time for the connection is significant. Second, the connection is not being authenticated in any way, which may or may not be reasonable for your setup. Finally, the entire response is being read into memory at once, which is reasonable only when you know you are communicating with a friendly service which has a limited amount of return data. If any of these assumptions turn out to be incorrect, then you would need to modify the code above significantly. These same caveats apply in varying degrees to the other HTTP methods used by REST also.

PUT TRANSACTIONS

In REST, PUT transactions are used to update, or in some cases create, an object that has the specified URL. The transaction therefore needs to send the object's data in the request, and the response contains only some indication of success or failure, as well as a numeric HTTP response code. A successful operation need not necessarily return response code SC_OK – it is also possible for SC_CREATED to be the return value, if the object is created as a result of the PUT. See listing 3.2.

Listing 3.2 A method using Httpcomponents Httpclient to perform a PUT operation

```
public static String performAPIRawPutOperation(String url,
  String input)
  throws IOException
{
  HttpClient client = HttpClients.createDefault();
  HttpPut method = new HttpPut(formURL(restPath));                    #1
  method.setEntity(new StringEntity(input,                            #2
    ContentType.create("text/plain","UTF-8")));                       #2
  HttpResponse httpResponse = client.execute(method);                 #3
  int response = httpResponse.getStatusLine().getStatusCode();        #3
  String responseString = convertToString(httpResponse);              #3
  if (response != HttpStatus.SC_OK &&                                 #4
    response != HttpStatus.SC_CREATED)                                #4
    throw new IOException("API http error; expected "+                #4
      HttpStatus.SC_OK+" or "+HttpStatus.SC_CREATED+", saw "+         #4
      Integer.toString(response)+": "+responseString);                #4
  return responseString;
}
```

#1 Use the httpclient PUT method
#2 Encode the input as utf-8
#3 Assume the response is utf-8 encoded
#4 Either 200 (OK) or 201 (CREATED) is fine

Here, at #1, we create a HttpPut object this time. Since this is a PUT, we transmit the input data at #2, before executing the method. At #3, we fetch and decode the response.

At #4, we make sure we got back either an HTTP `OK` response or an HTTP `CREATED` response.

Once again, this is very straightforward, but similar caveats apply as did for the GET method. No connection pooling is attempted, and both the request and response must fit entirely into memory.

POST TRANSACTIONS

POST transactions are used in REST to create objects that do not yet have a URL, but are going to be a part of some collection of similar objects. In other words, if you don't know what the URL for the created object is going to be, you execute a POST transaction instead of a PUT, and you do it to a URL that represents the entire collection of objects of the type you are trying to create. The response code we expect from a REST API POST is always `CREATED`, unless there's some kind of an error, because creation of objects is what the POST transaction exists for. It also logically follows that the response must contain information which makes it possible for the caller to construct the newly-created object's proper URL. The HttpClient code is in listing 3.3.

Listing 3.3 An Httpcomponents Httpclient-based POST method

```
public static String performAPIRawPostOperation(String url,
  String input)
  throws IOException
{
  HttpClient client = HttpClients.createDefault();
  HttpPost method = new HttpPost(formURL(restPath));                    #1
  method.setEntity(new StringEntity(input,                              #2
    ContentType.create("text/plain","UTF-8")));                         #2
  HttpResponse httpResponse = client.execute(method);                   #3
  int response = httpResponse.getStatusLine().getStatusCode();          #3
  String responseString = convertToString(httpResponse);                #3
  if (response != HttpStatus.SC_CREATED)                                #4
    throw new IOException("API http error; expected "+                  #4
      HttpStatus.SC_CREATED+", saw "+                                   #4
      Integer.toString(response)+": "+responseString);                  #4
  return responseString;
}
```

#1 Use the httpclient POST method
#2 Convert the request to utf-8
#3 Assume the response is also utf-8
#4 Only valid response code is 201 (CREATED)

If you are familiar now with the GET and PUT transactions, there are no additional surprises to be found with POST. At #1, we instantiate a `HttpPost` object. At #2, we set the input data, and execute the method. We decode the response at #3, and we make sure we got the right response code (only the HTTP `CREATED` response will do in this case). The caveats are similar as well.

DELETE TRANSACTIONS

DELETE transactions in REST are intended to allow you to delete the object referenced by the URL. No input is required, and the output signals success (`OK`) or failure. The HttpClient code is correspondingly quite simple. See listing 3.4.

Listing 3.4 An Httpcomponents Httpclient-based DELETE method

```
public static String performAPIRawDeleteOperation(String url)
  throws IOException
{
  HttpClient client = HttpClients.createDefault();
  HttpDelete method = new HttpDelete(formURL(restPath));                #1
  HttpResponse httpResponse = client.execute(method);                   #2
  int response = httpResponse.getStatusLine().getStatusCode();          #2
  String responseString = convertToString(httpResponse);                #2
  if (response != HttpStatus.SC_OK)                                     #3
    throw new IOException("API http error; expected "+                  #3
      HttpStatus.SC_OK+", saw "+                                        #3
      Integer.toString(response)+": "+responseString);                  #3
  return responseString;
}
```

#1 Use the httpclient Delete method
#2 Get the response, and presume it is utf-8
#3 Only legal response code is 200 (OK)

Once again, at #1 we create a `DeleteMethod` object. There is no input data for a delete, so we just execute the object. At #3 we decode the result, and insure that we saw an HTTP `OK` response code.

That's pretty much all you need to know to use Apache Httpcomponents Httpclient to communicate with the ManifoldCF REST API. The next question is how to form and interpret the JSON objects. We will start to cover that in the next section.

3.3.2 Using Configuration and ConfigurationNode classes

ManifoldCF has exactly the same problem that you do when it comes to dealing with JSON. It needs to interpret JSON objects, and it needs to form JSON objects and responses. So, one obvious way of forming and interpreting the JSON objects that the ManifoldCF API uses is to make use of the proper classes from ManifoldCF itself.

This approach has a lot to recommend it. For one thing, it reduces the chances that your code will turn out to be incompatible in some way with the API. As we will learn, ManifoldCF translates JSON to and from a Java object hierarchy, and that object hierarchy is more fundamental to ManifoldCF than the JSON representation. The object hierarchy has a deep relationship with ManifoldCF's connectors so, for example, a JSON object can be translated to or from an object hierarchy that represents the configuration information for a connection definition. Figure 3.2 shows this relationship pictorially.

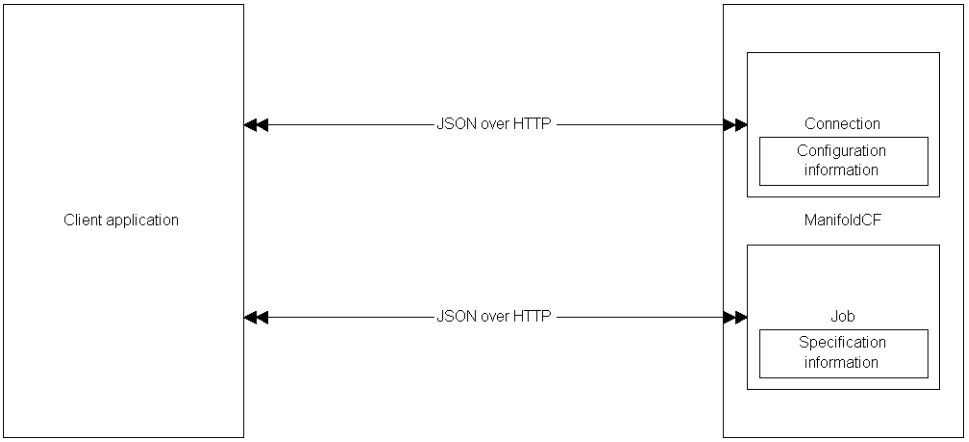

Figure 3.2: JSON is used to transfer connection definition configuration information, and job definition document and output specification information.

The classes you will be interested in are called `Configuration` and `ConfigurationNode`, which alludes to their basic function of being a hierarchy used for the purposes of representing configurations. They can be found in `mcf-core.jar`. Their functions are described in table 3.4.

Table 3.4: Configuration object hierarchy classes

Class	Function
org.apache.manifoldcf.core.Configuration	The root of an ordered hierarchy of nodes, which has some number of child nodes.
org.apache.manifoldcf.core.ConfigurationNode	An individual node, which can contain multiple child nodes, multiple attributes, and a single value.

If your purpose is to create some JSON using these classes, then only certain methods are likely to be interesting to you. On the other hand, maybe you are interested in interpreting a JSON response instead, in which case a different set of methods come into play. We'll introduce examples for each next.

CREATING JSON

Let's try to use these ManifoldCF classes to create some JSON. Please examine the code section in listing 3.5.

Listing 3.5 Creating a simple hierarchy using Configuration and ConfigurationNode

```
Configuration root = new Configuration();

ConfigurationNode helloNode = new ConfigurationNode("hello");            #A
helloNode.setAttribute("helloAttribute","helloAttributeValue");          #A
helloNode.setValue("helloNodeValue");                                    #A
root.addChild(root.getChildCount(),helloNode);                           #A

ConfigurationNode worldNode = new ConfigurationNode("world");            #B
worldNode.setAttribute("worldAttribute","worldAttributeValue");          #B
worldNode.setValue("worldNodeValue");                                    #B

ConfigurationNode nestedWorldNode =                                      #C
  new ConfigurationNode("nestedunderworld");                             #C
worldNode.addChild(worldNode.getChildCount(),nestedWorldNode);           #C

root.addChild(root.getChildCount(),worldNode);
System.out.println("JSON="+root.toJSON());
```

#A Creates "hello" node
#B Creates "world" node
#C Creates "nestedunderworld" node, under "world"

This code snippet is meant to construct a simple explanatory hierarchy of objects, with attributes and values as portrayed in figure 3.3.

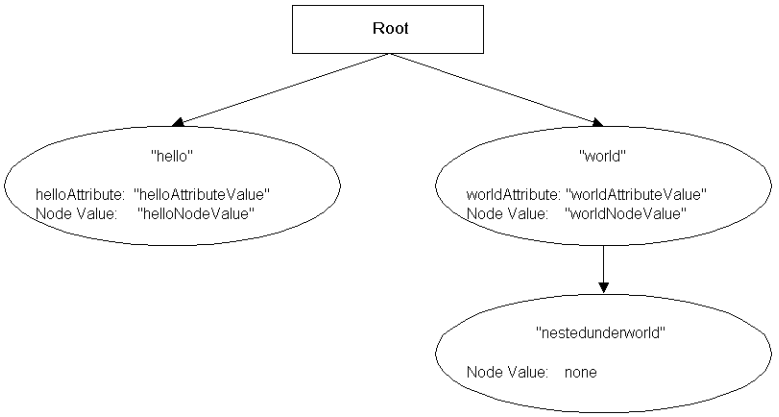

Figure 3.3: An example hierarchical structure, showing node attributes, values, and children.

The section uses certain class methods in conjunction with the `Configuration` and `ConfigurationNode` objects, which deserve a full explanation. They are described in table 3.5.

Table 3.5: Methods for constructing configuration hierarchies

Method	Function
`ConfigurationNode` constructor	Builds a node with the specified name
`getChildCount()`	Returns the current number of child nodes for either a `Configuration` or `ConfigurationNode` object
`addChild()`	Adds a child node to either a `Configuration` or `ConfigurationNode` object, at the specified place
`setAttribute()`	Sets a specific `ConfigurationNode` attribute value
`setValue()`	Sets the `ConfigurationNode` value
`toJSON()`	Convert a `Configuration` object to a JSON string

Can you guess what the resulting JSON output might be for this example? You may find it difficult to do so, because we haven't yet discussed the mapping between the internal `Configuration` object hierarchy and JSON. For the record, here is the resulting JSON, formatted in a way that's easy to read.

```
{"hello":
  {"_value_":"helloNodeValue",
   "_attribute_helloAttribute":"helloAttributeValue"
  },
 "world":
  {"_value_":"worldNodeValue",
   "_attribute_worldAttribute":"worldAttributeValue",
   "nestedunderworld":
    {
    }
  }
}
```

It is clear that values and attributes have their own special JSON labels within a JSON object, and nested `ConfigurationNode` objects do the expected thing and generated a nested object in JSON. Table 3.6 describes the mapping between a `ConfigurationNode` hierarchy and a JSON object.

Table 3.6: ConfigurationNode structures to JSON objects mapping

ConfigurationNode concept	JSON equivalent
Node name, e.g. `node = new ConfigurationNode("xxx")`	Object label, e.g. "xxx" : {..}
Attribute, e.g.	Special label, e.g.

`node.setAttribute("x","y")`	"_attribute_x" : "y"
Value, e.g. `node.setValue("z")`	Special label, e.g. "_value_" : "z"

The only JSON structure we don't see in this example is a JSON array. No doubt you have already guessed that a JSON array will be produced when a node has multiple child nodes of the same name. For example,

```
Configuration root = new Configuration();
ConfigurationNode child = new ConfigurationNode("array");
root.addChild(0,child);
child = new ConfigurationNode("array");
root.addChild(1,child);
```

This will produce JSON as follows:

```
{"array":[ {}, {} ] }
```

We'll talk more about canonical forms and their simplified equivalents in the next section. But, first, let's reverse the process and try to parse some JSON using the same ManifoldCF classes.

INTERPRETING JSON

To interpret JSON using the `Configuration` and `ConfigurationNode` classes, all you need to do is the following.

```
Configuration root = new Configuration();
root.fromJSON(jsonString);
```

The object root will then be the parent of a hierarchy corresponding to the JSON string you passed in, using the same canonical rules as described in table 3.6. In short, the operations are meant to be reversible.

Traversing the `Configuration` hierarchy is straightforward. Please examine the code in listing 3.6.

Listing 3.6 A set of methods to write a nested hierarchy to System.out

```
public static void dumpConfiguration(Configuration root)
{
  int i = 0;                                                    #A
  while (i < root.getChildCount())                              #A
  {                                                             #A
    ConfigurationNode child = root.findChild(i++);              #A
    dumpConfigurationNode(child,0);                             #A
  }                                                             #A
}

public static void dumpConfigurationNode(ConfigurationNode node,
  int level)
{
  printSpaces(level);                                           #B
  System.out.println("Node '"+node.getType()+"'; "+             #B
    "value '"+node.getValue()+"'; attributes:");                #B
  Iterator iter = node.getAttributes();
```

```
    while (iter.hasNext())                                              #C
    {                                                                   #C
      String attributeName = (String)iter.next();                       #C
      String attributeValue =                                           #C
        node.getAttributeValue(attributeName);                          #C
      printSpaces(level+1);                                             #C
      System.out.println("Attribute '"+attributeName+"' "+              #C
        "value '"+attributeValue+"'");                                  #C
    }                                                                   #C
    printSpaces(level);                                                 #D
    System.out.println("Children:");                                    #D
    int i = 0;                                                          #D
    while (i < node.getChildCount())                                    #D
    {                                                                   #D
      ConfigurationNode child = node.findChild(i++);                    #D
      dumpConfigurationNode(child,level+1);                             #D
    }                                                                   #D
  }

  public static void printSpaces(int i)
  {
    int j = 0;
    while (j < i)
    {
      System.out.print("  ");
      j++;
    }
  }
```

#A Go through all Configuration children
#B Print the node type and value
#C Print the node attributes
#D Print the node children, properly indented

This code is intended to print a `Configuration` hierarchy to the standard output stream. In the process it demonstrates all the methods you will need to decode such a hierarchy for your own purposes. To reinforce the point, table 3.7 describes all the pertinent methods.

Table 3.7: Methods used to deconstruct JSON objects using Configuration and ConfigurationNode classes

Method	Function
`fromJSON()`	Reads a JSON string into an already constructed `Configuration` object
`getChildCount()`	Gets the number of child nodes of a `Configuration` or `ConfigurationNode` object
`findChild()`	Gets the specified child node of a `Configuration` or `ConfigurationNode` object
`getAttributes()`	Returns an iterator over the names of all attributes of a

	`ConfigurationNode` object
`getAttributeValue()`	Gets the value of a specified attribute of a `ConfigurationNode` object
`getValue()`	Gets the value of a `ConfigurationNode` object

If this seems pretty easy, that's good, because it is about to get a little harder. In the next section, we are going to show just how flexible ManifoldCF is about how it maps the `Configuration` object hierarchy to its JSON equivalent.

SHORTCUTS AND SYNTACTIC SUGAR

You may already have begun to gather that the mapping from a `Configuration` object hierarchy to JSON has certain rules. It may no longer surprise you to learn that ManifoldCF treats certain JSON expressions as equivalent, even though at first glance they look quite different. For example, the same `Configuration` hierarchy will result from parsing the following two rather different JSON objects:

```
{"array": {} }
```

Or:

```
{"array": [ {} ] }
```

If you think about it, you will realize that this is because `Configuration` objects represent arrays by having more than one of the same kind of node.

There are other cases of equivalence as well. Another example is an object that has no attributes and no children, but just a value:

```
{"object": {"_value_" : "myvalue"} }
```

is equivalent to:

```
{"object": "myvalue"}
```

The reasons for this conversion are more aesthetic than representational. Nevertheless, such a substitution might surprise you if you weren't looking for it. This can be especially vexing if you happen not to be using the `Configuration` and `ConfigurationNode` classes. Table 3.8 summarizes these cases.

Table 3.8: Equivalent ManifoldCF forms of JSON

Returned form	Equivalent
Single object, e.g. { … }	Array of objects with one object value, e.g. [{ … }]
Object name and value, e.g. "foo" : "bar"	Named object having no attributes, no children, and only a value, e.g. "foo" : { "_value_" : "bar" }
Similar child objects, e.g. "foo" : [{ … }, { … }]	Individually enumerated child objects, e.g. "_children_" : [foo:{ … }, foo:{ … }]

Note These equivalences may become clearer when you realize that there is a standard canonical JSON form corresponding to every `ConfigurationNode` object. This form is what is used when the `Configuration` class cannot find a simpler equivalent way in JSON to present the same node. That canonical form can be summarized as:

name : { "_attribute_*attribute1*" : *attribute1_value*,

"_attribute_*attribute2*" : *attribute2_value*, …,

"_value_" : *value*,

"_children_": [*childname1* : *child_object_1*,

childname2 : *child_object_2*],

… }

Now that we understand how ManifoldCF looks at JSON, we are ready to put everything together in a single class. We will use this class whenever we need to communicate with ManifoldCF's API.

3.3.3 Putting it all together

These building blocks naturally form a class that will enable us to effectively and easily communicate with the ManifoldCF API. This will include the Httpcomponents Httpclient interfacing code we introduced earlier, extended by JSON formation and interpretation using ManifoldCF's `Configuration` and `ConfigurationNode` classes. When we are done, we will have a single class that implements all four HTTP REST verbs, while accepting and returning ManifoldCF `Configuration` objects. I've also included the hierarchy dumper we wrote earlier that permits us to see what is in a `Configuration` hierarchy, for debugging purposes. You can examine the resulting class for yourself at http://manifoldcfinaction.googlecode.com/svn/examples/edition_2_revised/api_example/src/org/apache/manifoldcf/examples/ManifoldCFAPIConnect.java.

This helper class contains everything you might need to perform any ManifoldCF API transaction. In the next section, we'll use this class as a foundation for constructing our application.

3.4 Developing the application

The application itself will need code that implements a finite state machine, as well as code that communicates with the ManifoldCF API, and code that forms the requests and interprets the responses from that API.

For a first step, we need to complete our exploration of the ManifoldCF API JSON format. Then, we'll be in a position to finally write the application and its supporting methods.

3.4.1 Using the API to examine connection definitions

It is now time to explore what the JSON for RSS connection definitions looks like, so we can write the code that creates them using the API. To do that, we can use the API to obtain an example of what we want, and then just print it out. Indeed, we can use the `ManifoldCFAPIConnect` class to help us to do just that.

DEFINING AN EXAMPLE RSS CONNECTION DEFINITION WITH THE UI

First, we'll need to create the RSS connection definition that we want in the UI. We'll be using this connection definition to learn what the JSON form of a corresponding JSON repository connection object looks like. This connection definition should therefore have all the desired throttling characteristics, as well as the proper maximum number of connections, as we discussed when we went into the details of how to do an RSS news crawl earlier in the chapter. Figure 3.4 shows a view of such a connection definition.

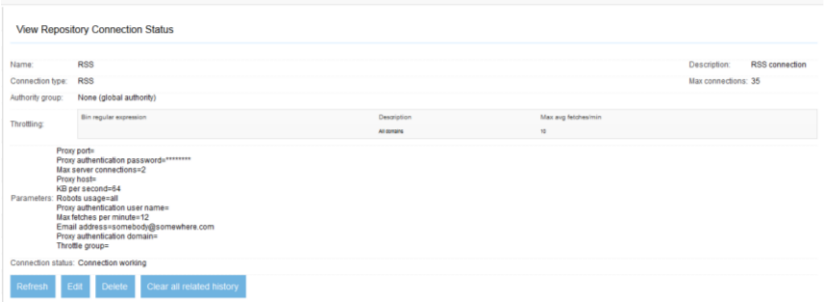

Figure 3.4: Screen shot of the properly set-up RSS repository connection definition. Note that the average fetches per minute is 10, and the maximum fetches per minute is 12. This helps insure that we don't use up all the worker threads waiting on throttles.

DUMPING JSON AND CONFIGURATION OBJECTS

Next, we will create a class to obtain the JSON repository connection definition object for the connection definition we just created and print it. It will also be handy to include a `Configuration` hierarchy dump as well, so if we need to we will know how to put one of these repository connection definition objects together using the `Configuration` class. The appropriate example code can be found at

http://manifoldcfinaction.googlecode.com/svn/examples/edition_2_revised/api_example/src/org/apache/manifoldcf/examples/RepositoryConnectionDumper.java.

If you run this program, and point it to the repository connection definition we just created, you should see the output in listing 3.7.

Listing 3.7: Dump output of the model RSS connection definition we want to produce, with hand-formatted JSON

```
--NODE HIERARCHY FOLLOWS--
Node 'repositoryconnection'; value 'null'; attributes:                    #1
Children:                                                                 #1
  Node 'isnew'; value 'false'; attributes:                                #2
  Children:                                                               #2
  Node 'throttle'; value 'null'; attributes:                              #3
  Children:                                                               #3
    Node 'match'; value ''; attributes:                                   #3
    Children:                                                             #3
    Node 'match_description'; value 'All domains'; attributes:            #3
    Children:                                                             #3
    Node 'rate'; value '1.6666666E-4'; attributes:                        #3
    Children:                                                             #3
  Node 'description'; value 'RSS connection'; attributes:
  Children:
  Node 'configuration'; value 'null'; attributes:                         #4
  Children:                                                               #4
    Node '_PARAMETER_'; value 'somebody@somewhere.com'; attributes:
      Attribute 'name' value 'Email address'
    Children:
    Node '_PARAMETER_'; value 'all'; attributes:
      Attribute 'name' value 'Robots usage'
    Children:
    Node '_PARAMETER_'; value '64'; attributes:                           #5
      Attribute 'name' value 'KB per second'                              #5
    Children:                                                             #5
    Node '_PARAMETER_'; value '2'; attributes:
      Attribute 'name' value 'Max server connections'
    Children:
    Node '_PARAMETER_'; value '12'; attributes:
      Attribute 'name' value 'Max fetches per minute'
    Children:
    Node '_PARAMETER_'; value ''; attributes:
      Attribute 'name' value 'Throttle group'
    Children:
    Node '_PARAMETER_'; value ''; attributes:                             #6
      Attribute 'name' value 'Proxy host'                                 #6
    Children:                                                             #6
    Node '_PARAMETER_'; value ''; attributes:                             #6
      Attribute 'name' value 'Proxy port'                                 #6
    Children:                                                             #6
    Node '_PARAMETER_'; value ''; attributes:                             #6
      Attribute 'name' value 'Proxy authentication domain'                #6
    Children:                                                             #6
    Node '_PARAMETER_'; value ''; attributes:                             #6
```

```
      Attribute 'name' value 'Proxy authentication user name'            #6
    Children:                                                            #6
    Node '_PARAMETER_'; value ''; attributes:                            #6
      Attribute 'name' value 'Proxy authentication password'             #6
    Children:                                                            #6
  Node 'class_name'; value                                               #7
    'org.apache.manifoldcf.crawler.connectors.rss.RSSConnector';         #7
    attributes:                                                          #7
  Children:                                                              #7
  Node 'name'; value 'RSS'; attributes:
  Children:
  Node 'max_connections'; value '35'; attributes:
  Children:
--JSON FOLLOWS--
{"repositoryconnection":                                                 #8
  { "isnew":"false",                                                     #9
    "throttle":                                                         #10
      { "match":"",                                                    #10
        "match_description":"All domains",                             #10
        "rate":"1.6666666E-4"},                                        #10
    "description":"RSS connection",
    "configuration":                                                    #11
      {"_PARAMETER_":
        [
          {"_value_":"somebody@somewhere.com",
           "_attribute_name":"Email address"},
          {"_value_":"all",
           "_attribute_name":"Robots usage"},
          {"_value_":"64",                                             #12
           "_attribute_name":"KB per second"},                         #12
          {"_value_":"2",
           "_attribute_name":"Max server connections"},
          {"_value_":"12",
           "_attribute_name":"Max fetches per minute"},
          {"_value_":"",
           "_attribute_name":"Throttle group"},
          {"_value_":"",
           "_attribute_name":"Proxy host"},
          {"_value_":"",
           "_attribute_name":"Proxy port"},
          {"_value_":"",
           "_attribute_name":"Proxy authentication domain"},
          {"_value_":"",
           "_attribute_name":"Proxy authentication user name"},
          {"_value_":"",
           "_attribute_name":"Proxy authentication password"}
        ]
      },
    "class_name":                                                       #13
      "org.apache.manifoldcf.crawler.connectors.rss.RSSConnector",      #13
    "name":"RSS",
    "max_connections":"35"
  }
}
--END—
```

#1 Outer-level repository connection node

#2 true for new object
#3 Throttle
#4 Connector-specific info
#5 Typical connection definition parameter
#6 Parameters for proxies, unused here
#7 Connector class name
#8 JSON outer-level repository node
#9 JSON isnew field
#10 JSON throttle
#11 JSON connector-specific info
#12 Typical connection definition parameter in JSON
#13 JSON connector class name

The outer node at #1 corresponds to the JSON outer node at #8. The `isnew` field at #2 should be `true` for new objects, `false` otherwise, and corresponds to the JSON at #9. At #3, you see the structure corresponding to the default throttle, which is also in the JSON at #10. At #4, connector-specific information begins, just as it does in the JSON at #11. A typical connection definition parameter looks like #5, or like #12 in the JSON. The nodes at #6 all correspond to parameters we are not using in this example, having to do with fetching RSS content through a proxy. The node at #7, and at #13 in the JSON, contains the repository connector class, which should be a registered repository connector.

This output shows us much of what we need to know to create the hierarchical structure that represents a repository connection definition. It also seems to prove something: that it is easier and less error-prone to use the `Configuration` and `ConfigurationNode` classes than it is to try to produce the correct corresponding JSON directly. So, from this point forward, we'll only use the former approach.

3.4.2 Examining job definitions using the API

We will need to do something similar to what we have just done for the connection definition for the RSS job definition that our application will need to create. Then, once again, we'll use the API to request the corresponding job definition object, and print it out.

CREATING AN EXAMPLE RSS JOB DEFINITION

A screen shot of the ideal job is presented in figure 3.5. This is based entirely on the continuous RSS job characteristics we discussed earlier in the chapter.

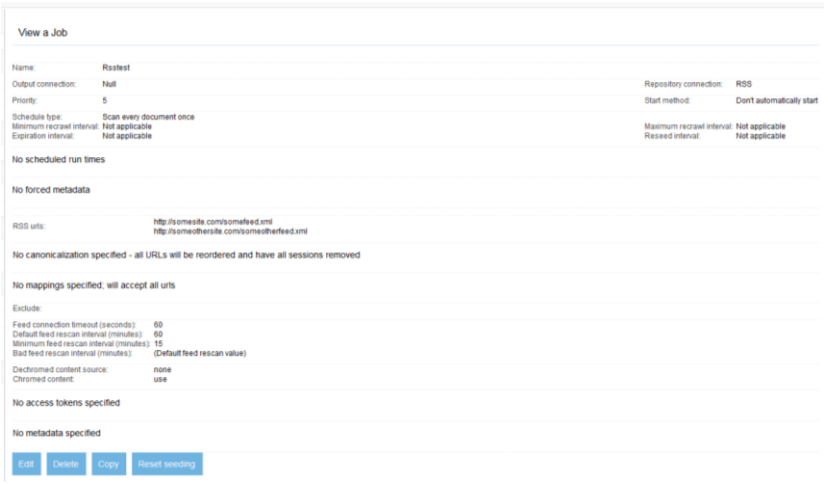

Figure 3.5: A view of an ideal RSS continuous job definition. The job definition was created using an output connection definition that did not have any output specification information, hence there is none on this display. A job that used a different output connection definition might well have had additional information.

DUMPING THE RSS JOB OBJECT

It is now pretty obvious how to obtain a node and JSON output for a properly configured job, using the classes we've already developed. I will therefore leave that as an exercise for the reader. If you want to cheat and look at my code, it's in the standard place: http://manifoldcfinaction.googlecode.com/svn/examples/edition_2_revised/api_example/src/org/apache/manifoldcf/examples/JobDumper.java.

The node output for the job displayed in figure 3.5 is provided in listing 3.8.

Listing 3.8 Configuration node output corresponding to the job in figure 3.4

```
--NODE HIERARCHY FOLLOWS--
Node 'job'; value 'null'; attributes:                                    #1
Children:                                                                #1
  Node 'start_mode'; value 'manual'; attributes:
  Children:
  Node 'reseed_interval'; value '3600000'; attributes:
  Children:
  Node 'schedule'; value 'null'; attributes:                             #2
  Children:                                                              #2
  Node 'id'; value '1287343465833'; attributes:
  Children:
  Node 'run_mode'; value 'continuous'; attributes:
  Children:
  Node 'hopcount_mode'; value 'accurate'; attributes:
  Children:
  Node 'output_specification'; value 'null'; attributes:                 #3
  Children:                                                              #3
```

```
    Node 'description'; value 'rsstest'; attributes:
    Children:
    Node 'document_specification'; value 'null'; attributes:                #4
    Children:                                                               #4
      Node 'chromedmode'; value ''; attributes:
        Attribute 'mode' value 'none'
      Children:
      Node 'feedtimeout'; value ''; attributes:
        Attribute 'value' value '60'
      Children:
      Node 'badfeedrescan'; value ''; attributes:
        Attribute 'value' value '1440'
      Children:
      Node 'feed'; value ''; attributes:
        Attribute 'url' value 'http://somesite.com/somefeed.xml'
      Children:
      Node 'feed'; value ''; attributes:
        Attribute 'url' value 'http://someothersite.com/someotherfeed.xml'
      Children:
      Node 'dechromedmode'; value ''; attributes:
        Attribute 'mode' value 'none'
      Children:
      Node 'minfeedrescan'; value ''; attributes:
        Attribute 'value' value '15'
      Children:
      Node 'feedrescan'; value ''; attributes:
        Attribute 'value' value '60'
      Children:
    Node 'output_connection'; value 'null'; attributes:
    Children:
    Node 'repository_connection'; value 'RSS'; attributes:
    Children:
    Node 'priority'; value '5'; attributes:
    Children:
    Node 'expiration_interval'; value '2592000000'; attributes:
    Children:
    Node 'recrawl_interval'; value 'infinite'; attributes:
    Children:
  --END—
```

#1 Outer-level wrapper node
#2 Schedule node
#3 Output connector dependent contents
#4 Repository connector dependent contents

Here, the outer level wrapper node at #1 wraps the entire job object. The schedule node at #2 is degenerate in this case, because we have no child schedule records. Structure that is dependent on the actual type of output connector involved begins at #3; here it is degenerate because the null output connector requires none. Repository connector-specific information begins at #4.

3.4.3 Writing the finite state machine

To build a general finite state machine, we need to generalize on the kinds of things found in any FSM, such as states and transitions. Then we can write an engine in terms of these pieces. Implementing the pieces in the context of our specific problem is the final step.

STATE ABSTRACTION

Let's start with a state abstraction. This is straightforward; all that we care about for each state are the transitions that leave it. The following interface class seems both necessary and sufficient, if indeed very simple.

```
public interface State
{
  public Transition[] getTransitions();
}
```

TRANSITION ABSTRACTION

Next, we'll need an abstraction for transitions. But for us to describe a transition, we will need the transition to be able to interrogate ManifoldCF, and also take actions that might affect its state. Where do we put that?

The answer is **not** to put any of the nuts and bolts of the ManifoldCF API into the transition abstraction itself. Remember that any class implementing the abstraction can have references of its own to underlying support classes, so there is no need to mention any of the details here.

```
public interface Transition
{
  public boolean isTransitionReady()
    throws InterruptedException;

  public State executeTransition()
    throws InterruptedException;
}
```

Here we have two methods, one of which checks to see whether we should take this transition, and the second which performs that transition and returns the new state we should enter.

THE ENGINE

Finally, we are ready to supply the engine itself. The engine will start with an initial state, and will then run until it is done, at which time the execution method returns. It can be interrupted, of course, as should any well-written Java program, so we'll allow it to throw `InterruptedException`. Logically, it must look for pertinent state transitions and execute them when it finds them. If it can't find any, it must wait for a period of time before looking again. See listing 3.9.

Listing 3.9: The Engine class

```
public class Engine
{
  protected State currentState;
```

```
  public Engine(State initialState)
  {
    this.currentState = initialState;                                    #A
  }

  public void execute()
    throws InterruptedException
  {
    while (true)
    {
      if (currentState == null)                                          #B
        break;                                                           #B
      Transition[] transitions =                                         #C
        currentState.getTransitions();                                   #C
      boolean newStateFound = false;                                     #C
      int i = 0;                                                         #C
      while (i < transitions.length)                                     #C
      {                                                                  #C
        Transition t = transitions[i++];                                 #C
        if (t.isTransitionReady())                                       #C
        {                                                                #C
          currentState = t.executeTransition();                          #C
          newStateFound = true;                                          #C
          break;                                                         #C
        }                                                                #C
      }                                                                  #C
      if (newStateFound)
        continue;
      Thread.sleep(10000L);                                              #D
    }
  }
}
```

#A Set the current state in the constructor
#B Exit if we no longer have a state
#C Examine the transitions in order
#D If no transition found, sleep for 10 seconds

As always, the complete code for all of these interfaces and classes can be found at http://manifoldcfinaction.googlecode.com/svn/examples/edition_2_revised/api_example/src/org/apache/manifoldcf/examples.

There is one case this class glosses over, which is the very real possibility that more than one transition may be possible at any given moment. However, since we do allow each state the ability to control the order of the transitions that are evaluated, there is already an implicit mechanism for controlling transition priority, so no explicit mechanism needs to be created in the `Engine` class to deal with this possibility.

So now it looks like we are almost ready to implement the states and state transitions that match our earlier diagram. But not so fast – before we can write these, we'll want to create methods somewhere that use the class we developed in section 3.1 in order to perform those basic API transactions that we will need. That will make it easier for us to create the state and transition classes to model our diagram.

3.4.4 Methods that wrap the API

The list the methods that we need that will use the API operations we identified earlier are listed below.

- Check the status of an output connection definition
- Check whether a repository connection definition exists
- Create a new RSS repository connection definition
- Check whether a job definition exists
- Create a new RSS job definition
- Start a job
- Check a job's status
- Get an RSS job definition's feed urls
- Set an RSS job definition's feed urls

Some of these methods are clearly generic, and can be used with any kind of underlying connector. Others are specific to connection definitions and job definitions that are based on the RSS connector. To reflect this difference, and to allow for maximum reuse of the example code, I've chosen to extend the `ManifoldCFAPIConnect` class to add methods that are generic for all kinds of connectors, and then extend that new class in order to add still more methods that are specific to the RSS connector.

At this point you may be starting to regret the fact that each connector has its own idea of configuration and specification information, since this means that all of our integration efforts must necessarily be connector specific too. Unfortunately, this cannot be helped. ManifoldCF does an excellent job of wrapping up details of repositories and authorities into the bundles of code called connectors, even to the point of giving each connector code the ability to supply its own UI components. But when you look under that wrapper, as you do when you use the API, you are no longer insulated from individual connector details.

NON-CONNECTOR-SPECIFIC API WRAPPING METHODS

Now we are ready to implement the class which provides the non-connector-specific API methods. I've taken care to create as many reusable building blocks as possible, so that it will hopefully be easy to implement future methods. Listing 3.10 contains demonstrative portions of the class I am describing. The complete code can be found at http://manifoldcfinaction.googlecode.com/svn/examples/edition_2_revised/api_example/src/org/apache/manifoldcf/examples/GenericMCFAPISupport.java.

Listing 3.10 A portion of class GenericMCFAPISupport

```
public String checkOutputConnection(String connectionName)
  throws IOException, ManifoldCFException
{
  String connectionURI = "/outputconnections/"+                    #1
    connectionNameEncode(connectionName);                          #1
```

```
    return getConnectionStatus(performAPIGetOperation(connectionURI));
  }

...

  protected static String getConnectionStatus(Configuration response)
    throws ManifoldCFException
  {
    int i = 0;
    while (i < response.getChildCount())
    {
      ConfigurationNode node = response.findChild(i++);
      if (node.getType().equals(NODETYPE_ERROR))                          #2
        throw new ManifoldCFException(                                    #2
          "Server error: "+node.getValue());                              #2
      else if (node.getType().equals(NODETYPE_CHECKRESULT))               #3
        return node.getValue();                                           #3
    }
    return null;
  }
```

#1 Connection definition names specially encoded
#2 Always check for error node in response
#3 Specific to the kind of response

At #1, we build the REST resource URL for the connection definition object. The connection definition name itself is specially encoded, because not only is it Unicode, but it may contain "/" characters, which would interfere with URL parsing. At #2, we look for an error node in the response, and throw an exception if we find it. The result check at #3 is specific to a status request; other kinds of API requests will correspondingly have a different check.

RSS-CONNECTOR-SPECIFIC API WRAPPING METHODS

As I mentioned before, I have placed the methods that perform RSS-connector-specific activities in a separate class, which extends the generic class. These methods include support for RSS connection definition creation, RSS job definition creation, and the updating of an RSS job definition's feeds. A portion of this class is described in listing 3.11 below. The complete class can be found at http://manifoldcfinaction.googlecode.com/svn/examples/edition_2_revised/api_example/src/org/apache/manifoldcf/examples/RSSMCFAPISupport.java.

Listing 3.11 A portion of class RSSMCFAPISupport

```
  public void createRSSRepositoryConnection(String connectionName,
    String connectionDescription,
    Double maxAverageFetchRate,
    String emailAddress,
    Integer maxKBperSecondPerConnection,
    Integer maxConnectionsPerServer,
    Integer maxFetchesPerMinutePerServer)
    throws IOException, ManifoldCFException
  {
    Configuration connectionConfiguration = new Configuration();
    addParameterNode(connectionConfiguration,                             #1
```

```
      "Email address",emailAddress);                                    #1
    addParameterNode(connectionConfiguration,"Robots usage","all");
    if (maxKBperSecondPerConnection != null)
      addParameterNode(connectionConfiguration,
        "KB per second",maxKBperSecondPerConnection.toString());
    if (maxConnectionsPerServer != null)
      addParameterNode(connectionConfiguration,
        "Max server connections",maxConnectionsPerServer.toString());
    if (maxFetchesPerMinutePerServer != null)
      addParameterNode(connectionConfiguration,
        "Max fetches per minute",maxFetchesPerMinutePerServer.toString());
    createRepositoryConnection(connectionName,                          #2
      connectionDescription,                                            #2
      "org.apache.manifoldcf.crawler.connectors.rss.RSSConnector",      #2
      100,                                                              #2
      maxAverageFetchRate,                                              #2
      connectionConfiguration);                                         #2
  }
```

#1 Convenience method for adding a parameter node
#2 Use the generic method to do the rest

At #1, we use the convenience method we wrote earlier to set the RSS-specific email address parameter in the node structure. We then use the generic connection definition creation method at #2 to finish the job of creating the connection definition.

With the completion of these two supporting classes, we are now ready to write code for our states and transitions.

3.4.5 States and transitions

The code for each state and transition is quite simple now. The only problem that one can foresee is the potential number of states and transitions. So it would seem wise to share such code as much as possible.

Luckily, as is often the case with FSM implementations, many actual transitions can often be reused unchanged from state to state. All we really will need to do, then, is make sure that we set things up so that we can actually share transition implementations in this way.

Listing 3.12 consists of portions of the main class, which I've called `RSSCrawlMonitor`, which has some number of subclasses that represent the states and transitions that correspond to figure 3.1. The initialization of the states and transitions, and an implementation of one of the transitions, is described within. The url for the complete class is
http://manifoldcfinaction.googlecode.com/svn/examples/edition_2/api_example/src/org/apache/manifoldcf/examples/RSSCrawlMonitor.java.

Listing 3.12 Portions of class RSSCrawlMonitor class

```
public class RSSCrawlMonitor
{
  protected BasicState startState = new BasicState();                   #1
  protected BasicState notYetRunningState = new BasicState();           #1

...
```

```
  public RSSCrawlMonitor(String baseURL, String outputConnectionName,
    File feedFile)
  {
    this.apiAccess = new RSSMCFAPISupport(baseURL);
    this.outputConnectionName = outputConnectionName;
    this.feedFile = feedFile;
...
    startState.addTransition(                                          #2
      new JobNotRunningCheck(                                          #2
        notYetRunningState));                                          #2
...
  }

  protected abstract class BaseTransition                              #3
    implements Transition                                              #3
  {
    protected State targetState;
    protected Exception theException = null;

    public BaseTransition(State targetState)
    {
      this.targetState = targetState;
    }

    public boolean isTransitionReady()
      throws InterruptedException
    {
      try
      {
        return checkTransition();
      }
      catch (InterruptedIOException e)
      {
        throw new InterruptedException(e.getMessage());
      }
      catch (Exception e)                                              #4
      {                                                                #4
        theException = e;                                              #4
        return true;                                                   #4
      }                                                                #4
    }

    public State executeTransition()
      throws InterruptedException
    {
      if (theException != null)                                        #5
      {                                                                #5
        theException.printStackTrace(System.err);                      #5
        soundTheAlarm("Lost communication with ManifoldCF");           #5
        return null;                                                   #5
      }                                                                #5
      try
      {
        doTransition();
```

```
      }
      catch (InterruptedIOException e)
      {
        throw new InterruptedException(e.getMessage());
      }
      catch (Exception e)                                                    #6
      {                                                                      #6
        e.printStackTrace(System.err);                                       #6
        soundTheAlarm("Lost communication with ManifoldCF");                 #6
        return null;                                                         #6
      }                                                                      #6
      return targetState;
    }

    protected boolean checkTransition()
      throws IOException, ManifoldCFException
    {
      return true;
    }

    protected void doTransition()
      throws IOException, ManifoldCFException
    {
    }
  }

...

  protected class JobNotRunningCheck                                         #7
    extends BaseTransition                                                   #7
  {
    protected String jobID;

    public JobNotRunningCheck(State targetState)
    {
      super(targetState);
    }

    public boolean checkTransition()
      throws IOException, ManifoldCFException
    {
      jobID = apiAccess.findJobID(JOB_NAME);                                 #8
      if (jobID == null)
        return false;
      String jobStatus = apiAccess.getJobStatus(jobID);
      if (jobStatus == null)
        return false;
      return !jobStatus.equals("running") &&                                 #9
        !jobStatus.equals("starting up");                                    #9
    }

    public void doTransition()
      throws IOException, ManifoldCFException
    {
      lastStateTime = System.currentTimeMillis();                           #10
    }
```

```
    }

  ...

  }
```

#1 State declarations
#2 Add a transition to a state
#3 Base transition class, for handling exceptions
#4 Record the exception, agree to the transition
#5 If recorded exception, terminate by going to null state
#6 Do the same thing if exception during state transition
#7 One of the transition classes
#8 Looking up job by name is not necessarily safe
#9 Only two status values that mean everything is okay
#10 Record the time, for use by other transitions

At #1, we see two state declarations. The `startState` state is meant to be the state we begin on, while the `notYetRunningState` represents the state where the job has been created but it has not yet been started. There are, of course, many other states, but you will have to look at the complete listing to see these. At #2, we add a transition to a `BasicState` object, in this case to `startState`. We define a base transition class at #3, because otherwise every transition class would need to deal with exceptions. At #4, we see some exception handling. An exception causes the transition to be taken, but the exception is recorded for future use. If we see that an exception has been recorded at #5, we transition to the null state, which terminates the FSM. We do the same thing if an exception occurs during state transition, at #6.

An example transition class can be found at #7. This transition is intended to fire only if the job is not running. The transition check method looks for the job ID given the name at #8. This may fail to find a job, so we must deal with a potential null return. At #9, we check the job status for the two values that indicate that the job is running, and perform the transition if all is not well. The actual state transition execution at #10, on the other hand, does nothing more than record the current time, for future reference.

That's it! With the completion of the main class, that is all the code that we need to implement the RSS monitor we set out to create! Well, at least, we think we have. Before we can be sure, we will want to test it.

3.5 Testing the application

The application is done. Now the application must be tested.

Testing the application requires that we run it. You may have noticed that the api-example area has a `lib` folder. (See http://manifoldcfinaction.googlecode.com/svn/examples/edition_2_revised/api_example/lib.) This folder contains the jars needed to build and run the example code. Included are the necessary jars that supply Httpcomponents Httpclient functionality, as well as the ManifoldCF components that include the `Configuration` and `ConfigurationNode` classes.

Running the example is as simple as supplying the proper class path to Java and specifying the main `RSSCrawlMonitor` class. The parameters the class requires will be spelled out in the usage message it prints.

3.5.1 Automated tests

The best way to test software is to write automated testing code. Typically these tests are developed alongside the application, and are changed as the application changes. To facilitate such an approach, we proposed a testing plan for the application as part of the design process earlier in the chapter. We considered both unit tests and end-to-end tests at that time.

It is not, however, our focus to develop such tests for all of the book's examples. That does not mean that developing such tests would not be a valuable exercise in itself. The reader is thus encouraged to implement the test plan as an exercise. If you do, you will learn much more about how to use the ManifoldCF API, as well as learning how to write testing code for integration software in general. If you want advice, I will supply only one hint: my experience in this area leads me to believe that end-to-end tests are more valuable than self-contained unit tests when the application is intended for integration.

If we aren't going to write these tests now, then how are we going to make sure our application works? Well, we're going to need to do it the old-fashioned way: trying it out by hand.

3.5.2 Manual testing

To try out our example program manually, let's start by picking out a couple of RSS feeds. You can find reasonable feeds from most news sites, such as http://www.cnn.com. Put these feeds in a file one line at a time, and then invoke the example class as follows:

```
<java> -cp <classpath> RSSCrawlMonitor <base_URL> <output_connection_name>
<feed_file>
```

Or, if you created your output connection definition with the name `Null`, you can simply use the supplied ant execution target to run the program:

```
ant run-api-example
```

The only caveat of our implementation is that the predefined output connection definition cannot require output specification information, because the monitor as it stands does not provide that when it creates the job. But if all is well, the output will look something like this:

```
Created repository connection 'RSS Connection'
Created job 1287935932032
Starting job 1287935932032
```

Clearly, the monitor seems to have created a connection definition and a job definition, and the job was started and does not seem to have aborted. If you enter the UI at this point, you can see that this is indeed the case.

For fun, let's click the `Abort` link on the job status screen, and wait to see what happens. If we wait long enough, the output on the console should now grow one more line to include the following:

```
Starting job 1287935932032
```

The monitor seems to have correctly picked up on the fact that the job was no longer running, and has started it up again. Congratulations, you've built a genuinely useful tool on top of ManifoldCF's API!

A note about the presentation of debugging techniques

In the author's experience, every platform requires a markedly different set of tools and approaches which will help a programmer figure out coding errors. For ManifoldCF, the problem is compounded because debugging integrations is quite different than debugging extensions, such as new connectors. Also, the debugging techniques required for each connector are unique, to a considerable extent. This means that with ManifoldCF there are a very large number of different potential debugging contexts, so an in-depth treatment of every one of them is out of the question.

I've thus had to make some compromises. First, in this book I will presume that the reader is familiar with basic Java code debugging techniques in whatever environment they choose to work in, and not attempt to teach this. Second, ManifoldCF-specific debugging techniques will not be localized in one chapter, but will be strewn throughout the book, presented where I believe they will do the most good. This allows me to tailor the discussion more appropriately for the context. For example, Chapter 7, which primarily describes how to write repository connectors, includes a section on debugging techniques that can be used effectively for a broad set of repository connectors.

Finally, while very basic debugging techniques such as how to use logging and how to interpret history reports will be presented early on in an example-based fashion, more advanced techniques such as packet capture and interpretation will be described without concrete examples. Indeed, I believe that a whole book of its own could readily be written to cover the more advanced techniques.

As far as debugging our API example is concerned, basic code debugging techniques and use of logging should easily suffice to diagnose most client bugs. And don't be afraid to augment your client code with debugging logic, if you need to!

3.6 Using the Script Interpreter

Since version 0.4-incubating, ManifoldCF has included its own Python-like scripting engine. This engine interacts with the API Service in just the same way we've been learning how to do in Java. The purpose of this engine is to allow users to write programs that create and manage connections and jobs without needing any of the boilerplate code one would need to do this in Java. Thus, using the Script Interpreter might be a good alternative, depending on the details of your application.

3.6.1 Running a script

The Script Interpreter has two modes of operation. The first is to execute a script from a specified file. For example, on Windows (being sure to first set the environment variable ENGINE_HOME to point to the `dist/script-engine` directory):

```
cd dist\script-engine
run-script my_script_file.mcf
…
```

The above will execute the contents of `my_script_file.mcf` as a ManifoldCF script. The other way to use the Script Interpreter is interactively. Executing the `run-script` command without a script file will start it in this mode. Anything you type at that point will be interpreted as script. For example:

```
run-script
print "hello";
hello
```

3.6.2 Objects in the scripting language

In the ManifoldCF scripting language, objects are completely described by their properties. This allows for a great deal of flexibility because as long as an object provides a property that you need you really don't need to care about what kind of object it actually is. This is the sense in which it is very similar to Python.

Indeed, this even applies to expression operations, such as '+' or '*'. The language interpreter parses the expression using standard Java-like rules of operator precedence, but it leaves it up to each object to figure out how to apply the operations. Thus, "7" + 4 will yield a string object with a value of "74", while 7 + 4 will yield an integer object with a value of 11. Thus:

```
print "7" + 4;
74
print 7 + 4;
11
```

The scripting language includes native objects representing integers, strings, floating point numbers, Boolean values, and arrays. But it also includes native objects representing URLs, HTTP responses, Connection Name objects, and Configuration and ConfigurationNode objects. The latter objects make interaction with the ManifoldCF API Service much more straightforward than interacting in Java. See table 3.9.

Table 3.9 Script-language object types and their uses

Object type	Utility
String	Standard basic string, e.g. "`hello world`"
Integer	Standard basic integer, e.g. `12345`
Float	Standard basic floating point value, e.g. `1.2345`

Boolean	Standard basic Boolean value, e.g. `true` or `false`
Array	Single-dimensioned arrays of objects, e.g. `[ 1, 2, 3 ]`
Dictionary	Unordered set of key/value pairs
Configuration	Equivalent of a JSON expression used to communicate with ManifoldCF API
ConfigurationNode	Hierarchical section or sub-section of a Configuration object
Connection name	A special kind of string representing the name of a ManifoldCF connection, which understands how to escape connection names when building API REST URLs
URL	A object representing a URL, with operations that allow easy path assembly
Result	The result code and payload from an API HTTP operation

A complete list of the objects and their attributes and operations can be found online at http://incubator.apache.org/connectors/en_US/script.html.

3.6.3 Commands in the scripting language

In addition to objects, the scripting language has a number of commands, which describe collectively what you can do within the language. We've already seen one of these demonstrated (the `print` command). Commands include the operations needed to interact with the API, such as HTTP GET, POST, PUT, and DELETE, as well as control flow and book-keeping operations, such as setting variables, loops, and conditionals. Table 3.10 lists the control-flow and book-keeping commands and their functions.

Table 3.10 Script language control-flow commands and their functions

Command	Function
`set` expression = expression `;`	Sets a variable, array element, or field to a value
`print` expression `;`	Prints the value of an expression
`if` expression `then` statements [`else` statements] `;`	Conditional control flow
`while` expression `do` statements `;`	Loop
`break ;`	Break out of loop
`error` expression `;`	Exit script with an error message, as described by the expression

`wait` expression`;`	Wait for the specified number of milliseconds
`insert` expression `into` expression [`at` expression] `;`	Insert an object into an array or configuration node object at the specified location
`delete` expression `from` expression `;`	Delete an object from the specified location within an array

As you can see, the scripting language does not at this time have any means of defining functions, methods, or classes. This may change someday, but for the moment, the scripting language is useful mainly for coding repetitive tasks, such as creating specific connection definitions. You may also notice that the language has very little it can actually do, but that's because I have not yet listed the HTTP commands that actually interact with the REST API. Those are in Table 3.11.

Table 3.11 Script language commands for interacting with the ManifoldCF API

Command	Function
`GET` expression = expression `;`	Perform a GET operation on the URL described by the second expression, putting the result in the first expression
`DELETE` expression = expression `;`	Perform a DELETE operation on the URL described by the second expression, putting the result into the first expression
`PUT` expression = expression `to` expression `;`	Perform a PUT operation of the second expression to the URL described by the third expression, putting the result into the first expression
`POST` expression = expression `to` expression `;`	Perform a POST operation of the second expression to the URL described by the third expression, putting the result into the first expression

As you can see, there are no real surprises here. The scripting language simply provides a more convenient way to manipulate the ManifoldCF API than coding in Java might, but doesn't otherwise alter the problem much. Still, the interactive ability of the script interpreter makes it an extremely convenient way of inspecting the structures that the API needs for various different types of connection. In the following example, that capability is exactly what we're going to demonstrate.

3.6.4 Script Interpreter example

Suppose we needed to learn a bit about the `Configuration`/`ConfigurationNode` structures that were used by a specific connection type. Earlier in this chapter we explored how to do that in Java code, but that meant multiple coding and compilation cycles. With

the script interpreter, all of that is no longer needed. See listing 3.13 for the output of an interactive session that examines the structures associated with a specified connection.

Listing 3.13 Using the interactive script engine to query for a web connection's parameters

```
set mybaseurl =                                                          #1
  (new url "http://localhost:8345/mcf-api-service") + "json";            #1
print mybaseurl;                                                         #2
http://localhost:8345/mcf-api-service/json                               #2
GET result = mybaseurl + "repositoryconnections" +                       #3
  (new connectionname "web");                                            #3
print result.__script__;                                                 #4
(200) {                                                                  #4
<< "repositoryconnection" : "" :  :
  << "description" : "" :  :  >>,
  << "name" : "web" :  :  >>,
  << "configuration" : "" :  :
    << "bindesc" : "" : "binregexp"="", "caseinsensitive"="false" :
      << "maxfetchesperminute" : "" : "value"="12" :  >>,
      << "maxkbpersecond" : "" : "value"="64" :  >>,
      << "maxconnections" : "" : "value"="2" :  >>
    >>,
    << "accesscredential" : "" : "domain"="",
      "password"="7A6A2222DAE2", "type"="basic",
      "urlregexp"="foo\\.com", "username"="foouser" :  >>,
    << "_PARAMETER_" : "someone@somewhere.com" :
      "name"="Email address" :  >>,
    << "_PARAMETER_" : "all" : "name"="Robots usage" :  >>
  >>,
  << "class_name" :
  "org.apache.manifoldcf.crawler.connectors.webcrawler.WebcrawlerConnector"
    :  :  >>,
  << "isnew" : "false" :  :  >>,
  << "max_connections" : "10" :  :  >>
 >> }
print result.__OK__.__script__;                                          #5
true                                                                     #5
print result.__value__.__script__;                                       #6
{                                                                        #6
<< "repositoryconnection" : "" :  :
  << "description" : "" :  :  >>,
  << "name": "web" :  :  >>,
  << "configuration" : "" :  :
    << "bindesc" : "" : "binregexp"="", "caseinsensitive"="false" :
      << "maxfetchesperminute" : "" : "value"="12" :  >>,
      << "maxkbpersecond" : "" : "value"="64" :  >>,
      << "maxconnections" : "" : "value"="2" :  >>
    >>,
    << "accesscredential" : "" : "domain"="", "password"="7A6A2222DAE2",
      "type"="basic", "urlregexp"="foo\\.com", "username"="foouser" :  >>,
    <<"_PARAMETER_" : "someone@somewhere.com" :
      "name"="Email address" :  >>,
    << "_PARAMETER_" : "all" : "name"="Robots usage" :  >>
```

```
    >>,
    << "class_name" :
  "org.apache.manifoldcf.crawler.connectors.webcrawler.WebcrawlerConnector"
    :  :  >>,
    << "isnew" : "false" :  :  >>,
    << "max_connections" : "10" :  :  >>
  >> }
```

#1 Create the base URL for talking with the API
#2 Print it, to see if we have it right
#3 Do an HTTP GET for a specific repository connection
#4 Print the complete result; note the __script__ attribute
#5 Print out whether result is HTTP "OK"
#6 Print the result payload; note the __value__ attribute

In this example, we first create a base URL object at #1. Printing the URL shows that it has the contents expected, at #2. At #3, we perform an HTTP GET operation on the repository connection we are interested in. We print the complete response, which consists of an HTTP response code plus a Configuration/ConfigurationNode payload, at #4. Note that we print the `__script__` attribute, since there is no string equivalent of the result object. At #5, we print the Boolean value for whether the response was an HTTP OK value or not, once again using the `__script__` attribute. Finally, at #6, we obtain the `__value__` attribute of the response, and print that alone.

3.7 Summary

In this chapter, we've learned how the ManifoldCF API works, and how to use it. This has required an introduction to JSON and the Apache Httpcomponents Httpclient library, which we built upon to provide a base class for communicating with the API. For an example, we then built an RSS crawl monitor, which creates and maintains a continuous RSS job, signaling when user intervention is required.

In the next chapter, we will conclude this section of the book by examining how to use the ManifoldCF authority service to maintain document security in the context of a search-engine integration.