4

Integrating with the Authority Service

This chapter covers

- Learning about the ManifoldCF security model
- Understanding the function of the Authority Service
- How to configure authorities using the Crawler UI
- How to use the Authority Service to add repository authorization to a search engine
- Demonstrating a security-conscious integration of ManifoldCF with Solr

In this chapter, we will introduce ManifoldCF's security model, and learn what the Authority Service does, and how it works. We'll define an authority, and use it to control access to content from a source repository which has been indexed by Solr.

Being able to enforce content security is essential, because no matter how good a search engine is, no one is going to willingly let one be used to index their content if they have no guarantee that their content's security is enforced. ManifoldCF can help provide that guarantee, as we will see in a moment.

4.1   Controlling access to documents

Your friend Bert seems to be avoiding you. He has not returned your calls, and his voice mailbox has filled up. But, one day, you bump into him in the grocery store. You can see right away that he's not happy, not happy at all.

When you ask him what the problem is, he confesses that he is on company-mandated leave. It turns out that his search engine worked a little too well. It seems that he

successfully indexed content from all sorts of places, including secure areas within his company's many repositories. Unfortunately, he did not realize that people searching his index might be able to see sensitive information in the document extracts he presented with his search results. How was he to know that some of the information that appeared would include sales strategy and projections for the next year?

   When you ask what he did to address this problem, Bert confesses that he rather quickly put together a system that was under his control which would enforce security for individual search results. He'd spent weeks at it, and then more time setting up individual accounts and permissions for his entire company's user base. When he brought this system on-line, everything seemed okay for a little while. But then, unbeknownst to him, a senior executive had been fired, and the executive's accounts had been locked – and Bert had not been told. The executive had used Bert's search engine to walk away with all sorts of sensitive information the very same afternoon.

4.1.1   *Repository security*

What Bert should have done was to provide software to make sure that any documents that are protected in any of his source repositories are provided with the same security at the time a search is performed. Proprietary content repositories almost always have powerful and complex authorization mechanisms built into them; this is one of the primary functions of such repositories. These native authorization mechanisms are often integrated with Active Directory, which is a ubiquitous product from Microsoft that provides authentication and authorization services for Windows systems. See figure 4.1 for a pictorial rendition of the security infrastructure of the canonical enterprise.
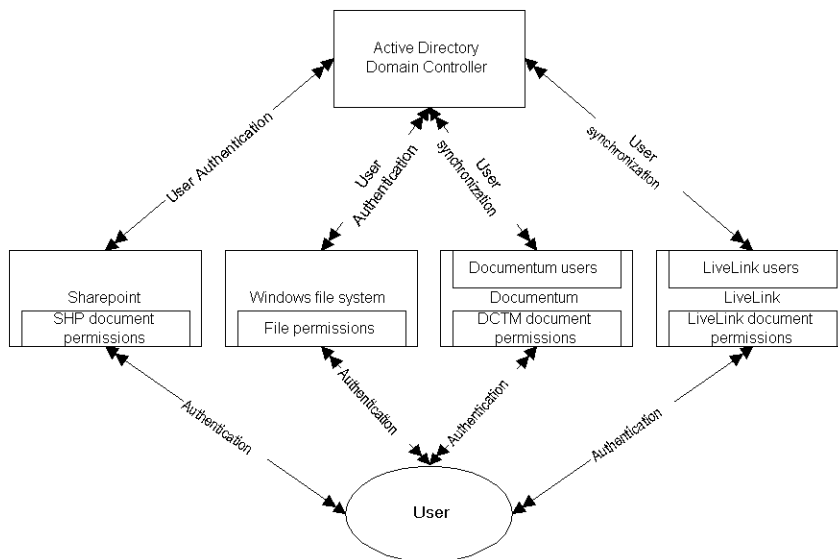
Figure 4.1 A user interacts with four kinds of repositories, and each one of them handles security differently. SharePoint uses both native and Active Directory security to manage its users. Windows uses Active Directory exclusively. Both Documentum and LiveLink keep their own set of users, which may or may not be synchronized with Active Directory users.

The typical components of repository security can be summarized as follows.

- An authentication mechanism, which allows the repository to positively identify each user
- Some way of grouping users together for the purposes of applying security
- Some way of grouping documents together for the purposes of applying security
- A means of associating users and user groups with documents and document groups, often with an explicit set of permissions, such as whether the user is allowed (or *not* allowed) to see a document, or is allowed to change it

Not all repositories have all of these components. SharePoint, for example, does not have native mechanisms for the first two, and instead uses Microsoft's Active Directory product exclusively for that purpose.   But on the whole, all content repositories have some mechanism for performing each of these functions.

MODELS FOR ENFORCING REPOSITORY SECURITY

We have already learned that a crawler's primary job is to synchronize content between a source repository and some target system. But it should be no surprise that an important part of that task is to help the target system secure those synchronized documents in a manner consistent with the source repository's security model.

This follows logically. After all, who is going to let their content be indexed, if they cannot assure themselves that the content will be properly protected, wherever it may wind up? It is in fact not possible to separate the two functions; synchronization without the ability to enforce a repository's security is of little use in the real world.

It turns out that there are two different ways to enforce a repository's security model in a search engine's results. Each of these methods has benefits and drawbacks.

RESULT FILTERING MODEL

The first model I'll call *result filtering*. Using this model, the search engine itself remains blissfully unaware of any security requirements, and returns results appropriate for all users. But before the results are presented to the user, the permission to display each result is obtained from the repository software. This arrangement is captured in figure 4.2.
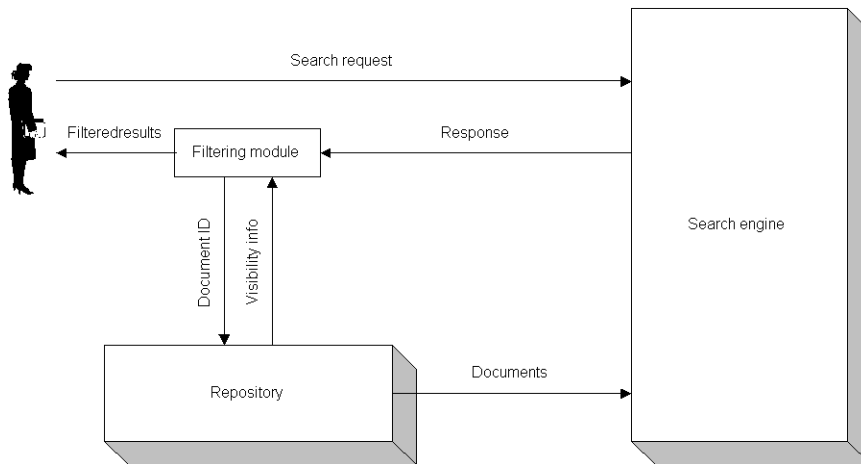
Figure 4.2 Result filtering. Each result is checked against the repository to be sure it can be viewed by the user.

The benefits of this approach are obvious. First, there is no need for the search engine to make any decisions whatsoever about security. Second, whatever convoluted logic the repository uses to figure out permissions remains opaque to the software involved in the security integration – there is no need to understand it at all.

But there are downsides to this very hands-off way of looking at the problem. The greatest downside is its impact on performance. Since every result must be independently checked, the amount of work that needs to be done goes up essentially linearly with the number of results. Even then, this may not be a big problem if most users can see most of the repository's documents. If, however, most users see only a small fraction of them, the amount of work that the repository needs to do per result increases even more dramatically. The limiting case is when the user can see no documents at all; in this case, the entire search engine document index may well need to be checked in order to provide that user with a "no documents match your search" answer.

QUERY RESTRICTION MODEL

The second approach I call *query restriction*. The query restriction model uses the search engine itself to perform the result restriction. All queries presented to the search engine are directly modified to apply the necessary restrictions specific to the particular user performing the search. Of course, the data the search engine indexes must be augmented to provide enough information to make it possible to do this. See figure 4.3 for a pictorial representation of this method.
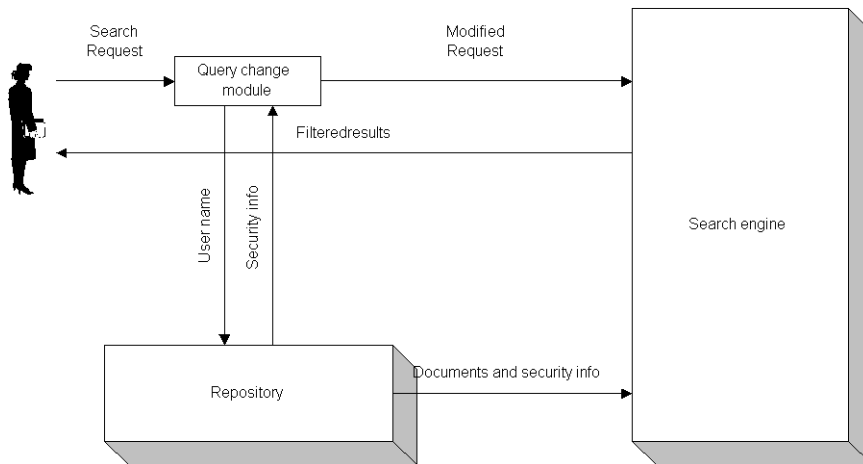
Figure 4.3 The query restriction model. Information about document security is distilled into metadata that is included in the document index, and the user's query is modified to include restrictions on the data returned based on the user's security information.

The theory behind this model is that modern search engines are well optimized to efficiently deal with search expressions, so we should let the search engine work its magic **before** thousands or millions of inappropriate results are returned. Furthermore, since the essence of every document's security has already been captured by the search engine at indexing time, the load on the content repository does not increase as the number of results increases. The repository work is limited at most to finding out information about the user doing the search, which needs to be done only once per search.

There are downsides of the query restriction model also. First, notice that this model implicitly replicates the repository's security. This implies a lag time between when the repository's security information changes, and when the search engine notices the change. While the essence of a user's authorization information is obtained for nearly every search, the document's authorization information is built into the index, so it will only be current as of the last time the index was built.

At first glance, this seems like a severe problem. If, as we agreed, crawlers are useless if they do not enforce security, then how can a crawler survive with such a delay? But it turns out that delays of this kind are usually acceptable, first because security policy changes to individuals are far more likely to be of immediate concern than security changes to documents, and second because changing security for an individual document version to a more restricted level just doesn't happen very often. "The horse has already left the barn", as they say.

The last, and most important, drawback with the query restriction model is that it requires you to find a way of converting the repository's security model into a search engine search expression. This problem also sounds like it might be difficult to solve but it turns out that the real-world solution is actually straightforward. We'll discuss how ManifoldCF performs this trick in the next section.

4.1.2   ManifoldCF security model

ManifoldCF defines a particular search model with the blind faith that it is possible to map whatever repository model you have onto that search model. Later on, I'll show why this blind faith is justified. First, though, I'll explain the model.

ACCESS TOKENS

We start with the ManifoldCF concept of *access token*. An access token is nothing more than a unit of authorization, which can come from a user, or be attached to a document in six different ways. It is actually represented as an unlimited-length arbitrary string, so neither the search engine nor the ManifoldCF framework needs to know precisely what's in it. Only the connectors that generate these access tokens need to know how they are put together.

Every authority defined within ManifoldCF accepts a user name and produces a corresponding set of user access tokens. It is these tokens that are used to qualify the user's search expression in order to limit the search results to just those answers the user has permission to see.

Every document handed to the search engine for indexing also contains access tokens. However, the tokens are classified in six standard ways. These ways correspond to the layered file permissions found in Active Directory.

Specifically, there are three different standard kinds of access tokens: *file*, *parent*, and *share*. Each kind of access token has the ability to allow or restrict access to the document independently. If the share tokens are taken to be the top level of token, and then the parent tokens are the next level, and finally the file tokens are the bottom level, then together there are three independent levels. For any given user, the document should only be presented if permission is granted by **all** of these levels.

For each individual level, the rules are simple. If there are no access tokens attached at that level at all, then all users are granted permission for the document to be viewed. If the user has any tokens of that kind attached using the "deny" flavor of attachment, then the user is denied permission. Barring all of that, the user is granted viewing permission for the level if the user has any tokens that are attached for the level using the "allow" flavor of attachment. Table 4.1 summarizes the standard ways you can attach access tokens to a ManifoldCF document, and what they mean.

Table 4.1 The standard ways you can attach access tokens to documents.

Attachment method	Meaning

File allow	An access token attached to a document in this way allows all users with said token to have "file" permission to view the document, unless the user also has tokens that are attached using the "file deny" method.
File deny	An access token attached to a document using this method prevents the users bearing this token from being granted "file" permission to see the document.
Parent allow	An access token attached to a document in this way allows all users with said token to have "parent" permission to view the document, unless the user also has tokens that are attached using the "parent deny" method. There is a single level of parent permission, which represents an amalgamation of all parent folders of the document.
Parent deny	An access token attached to a document using this method prevents the users bearing this token from being granted "parent" permission to see the document. There is a single level of parent deny permission, which represents an amalgamation of all parent folders of the document.
Share allow	An access token attached to a document in this way allows all users with said token to have "share" permission to view the document, unless the user also has tokens that are attached using the "share deny" method.
Share deny	An access token attached to a document using this method prevents the users bearing this token from being granted "share" permission to see the document.

In fact, the above permissions are arbitrary, and are considered "standard" only because that is what Windows does. ManifoldCF, under the covers, has a much more generic way of looking at security. For ManifoldCF, each set of allow and deny access tokens is simply given a unique key, e.g. "document" or "share", and all kinds of allow and deny token are treated exactly the same way. It is the output connector that determines whether documents which have access tokens of unsupported kinds will be rejected because their security cannot be guaranteed. For example, the Solr output connector rejects documents that have access token keys other than "document", "parent", or "share", because the Solr plugin that works with the connector doesn't support any other security token types.

Most third-party repositories do not need multiple levels of security tokens in any case, and thus they map just fine to a subset of the Active Directory functionality. Indeed, multiple levels of security exist in the model solely because of an historical accident. That accident traces its origins back to the early days of Windows, when the only security that could be applied was applied at the level of a Windows share. Later, when Active Directory was introduced, along with finer-grained per-file permissions, share security remained supported. Share security and file security operate completely independently in Windows, but you need permission to read from the share as well as permission to read a document in

order to be able to view the document. It is no coincidence that this is precisely how share permissions and file permissions interact in ManifoldCF.

MANIFOLDCF AUTHORITIES

Now it is finally possible to describe what a ManifoldCF authority actually does!

An authority's sole purpose is to convert a user identity into access tokens. The user identity consists of one or more tuples, where each tuple has an *authorization domain*, and a user name within that domain. For example, a domain might be "Active Directory", and the user name for that domain might be "bert@ad.widgetsRUs.com".   For your authority definition, you must always specify the authorization domain of the user name that the authority will use. The number and kinds of authorization domains used by your ManifoldCF setup will depend on how you've integrated ManifoldCF into your authorization environment. As we've discussed before, ManifoldCF does not perform authentication; it relies on external code to take care of that job. Authenticating against multiple systems, and thus coming up with multiple user names for a single user identity, is also considered to be the job of external code.

> **Note:** A ManifoldCF authorization domain is not the same thing as an Active Directory domain, although the two concepts fulfill a similar purpose.

The access tokens an authority produces presumably come from the repository it is configured to work with. Some kinds of repository have their own way of describing users and user groups, and these repositories need their own authority connectors.   Other repositories make use of Windows Active Directory for their user description; these repositories can use an Active Directory authority for their security needs. Table 4.2 lists the repositories supported as of this writing, and the authorities that support their security.

Table 4.2 Repositories and their associated authorities

Repository	Authorities
Local file system	None
HDFS (Hadoop)	None
Jira (Atlassian)	Jira
Google Drive	None
RSS	None
Web	None
Wiki	None
Database (JDBC)	Database (JDBC)

CIFS (Windows)	Active Directory
FileNet (IBM)	Active Directory
Documentum (EMC)	Documentum
LiveLink (OpenText)	LiveLink
Meridio	Meridio
SharePoint (Microsoft)	Active Directory, SharePoint Native + SharePoint Active Directory
CMIS	CMIS (limited in capability)
Alfresco	Currently unsupported

Your friend Bert, ever concrete, simply must know what an access token looks like for each kind of authority. He complains that it is hard for him to think about an abstract concept without a real example or two.

You assure Bert that this is mostly not something he will need to worry about until he is ready to write his own connectors. But nevertheless you are happy to provide some examples. Let's start with Active Directory, since many connectors are designed to work with the Active Directory authority. The Active Directory authority uses Active Directory SIDs as its access tokens. These are strings that start with the letter "S" and have numbers after that, separated by hyphens, such as "S-192-123456-33422". The LiveLink authority, on the other hand, uses LiveLink group or user identifiers, which are simply raw numbers. There are also some special numbers with special meanings, such as -1 or -2. Documentum uses tokens that are the names of Documentum access control lists, which typically have the form *<string>:<hexadecimal_number>*. The takeaway should be that each authority is perfectly able to structure its access tokens in whatever wildly unique way it wishes, without regard to the others, as long as the repository connectors that work with those authorities know how to do the same thing.

AUTHORITY-GROUP-QUALIFIED ACCESS TOKENS

One complexity that ManifoldCF brings to the picture is that it must provide security assistance for multiple repositories, not for just one. Thus, the access tokens from one repository must not inadvertently collide with any of those from another, or the security model would develop a "hole".

ManifoldCF makes sure that a collision cannot happen by attaching the name of the governing authority group to the front of each access token before it is handed to the search engine for inclusion in the index. For example, an authority group named `MyLivelinkAuthorityGroup` will have access tokens associated with it that all have the form `MyLivelinkAuthorityGroup:xxx`. Indeed, this is the reason that you must declare what the governing authority group is at the time you set up a repository connection

Chapter author name: Wright

definition; ManifoldCF needs to understand this in order to be able to properly qualify the access tokens for documents that come from that connection. By and large, this qualification is taken care of by the ManifoldCF framework, so individual connectors do not need to worry about the details. But if you see tokens of this form, you will know what they mean.

MAPPING SECURITY FROM REPOSITORY TO MANIFOLDCF

Essentially, now, the problem of search engine document security devolves into coming up with a mapping between any repository's security model and ManifoldCF's security model. How do we know one exists, and how do we find it?

In the end, it is always possible to find a mapping, although it is possible that the mapping may have characteristics that make it unworkable in practice. For repositories that have access control lists (or *ACLs*) attached to their documents, the mapping is obvious; these lists consist of users and user groups, or their equivalents, so a corresponding ManifoldCF access token would be the user id or user group id. Every kind of repository known to this author uses ACL-based security at this time, so clearly it is a predominate technique. However, even for a more esoteric architecture, it is still theoretically possible to come up with a mapping, simply by enumerating all the users that the repository knows about and constructing a list of users which are allowed to see the document. This list would be the de-facto ACL for the document. Now, you really wouldn't want to do it this way, because it would probably be enormously expensive, but nevertheless it is possible to do so in theory. Starting from that point, you can readily imagine how one might incrementally improve on that naïve approach in order to arrive at something more workable.   For example, if the system supported user groups, you would use these in preference to individual users wherever possible, which would cut down on the number of users that needed enumeration by an enormous number. If there are rules, you can see whether the rules are categorized to improve things further, since those rule categories might well represent de-facto user groups, when you think about it.

All of this is highly theoretical, because as I've already stated, repositories that use security models other than ACLs are rare and unusual. So we can take it for granted that a mapping exists, and that we shouldn't have to work terribly hard to find it. Next, we'll move on to understanding the Authority Service, which is how ManifoldCF makes repository authorization information available. Then, we'll build an example query restriction module in Solr, to test out our understanding.

4.2   Obtaining user authorization information

Now that you understand the kinds of ways a repository's security model can be projected to include a search engine's results, we're ready to discuss how ManifoldCF helps in this process. As you may have gathered, ManifoldCF uses the query restriction model to limit target search engine results for a specific user. That implies, though, that a user's search query must be modified according to that user's capabilities with each repository. Specifically, there must be a way to obtain the user's access tokens for the repositories we care about.

4.2.1   *Creating ManifoldCF authorities*

For each repository with security, there needs to be a way for ManifoldCF to obtain user access tokens from that repository. We do this by creating an authority connection definition, either with the ManifoldCF Crawler UI, or through the ManifoldCF API.

> **Remember** Not all repositories with security require specific authority connections to a content repository. Many repositories delegate their user and user group management to other systems, such as Active Directory. For repositories that use Active Directory for their user and user group management, an Active Directory authority would be used instead.

You've already seen the mechanics of how an authority connection definition might be created in Chapters 2 and 3. But now we're going to actually want to define one, so that we can develop our chapter example. For this example, we're going to create a "null authority" connection definition. The "null authority" does nothing more than convert the incoming user identifier into an identical access token, but this is sufficient for our demonstration.

DEFINING AN AUTHORITY GROUP

Enter the ManifoldCF user interface, and click the `List Authority Groups` link. As before, you should see a screen similar to figure 2.2. Click the `Add a new authority group` link. Fill in the name and description on the `Name` tab, using a name like "Null" and a reasonable description. See figure 4.4.
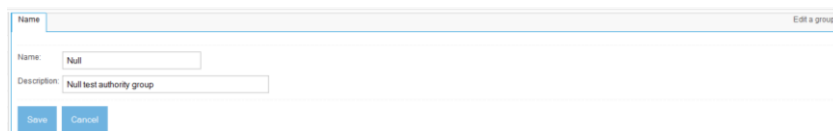

Figure 4.4 Creating an authority group for our null authority to live in.

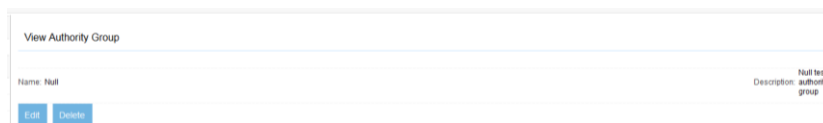Now, click the Save button. See figure 4.5.

Figure 4.5 The view screen of the "Null" authority group.

Congratulations! You've now created an authority group for our security example. Next, let's add an authority to this group.

Chapter author name: Wright

To create your authority, click on the `List Authority Connections` link. You should see a screen similar to figure 2.11. Click the `Add a new connection` link. Fill in the name and description on the `Name` tab, once again using a name like "Null" and reasonable description. See figure 4.6.
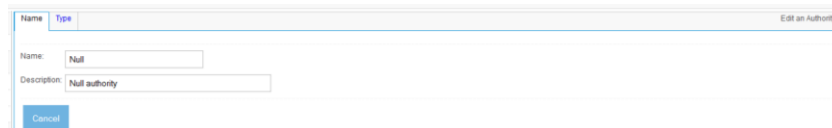

Figure 4.6 The Name tab for our example Null authority connection definition.

Then, click the `Type` tab. Select the `Null` authority type, and the `Null` authority group (which is the one we just created). Leave the selected authorization domain alone, and click the `Continue` button. See figure 4.7.
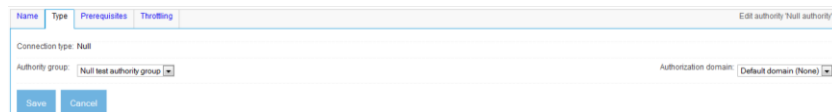

Figure 4.7 The tabs associated with the Null authority type. As you might expect, only the standard tabs are present.

Finally, click the `Save` button. That was easy! Now you have created a null authority connection definition that we can use for our example. See figure 4.8.
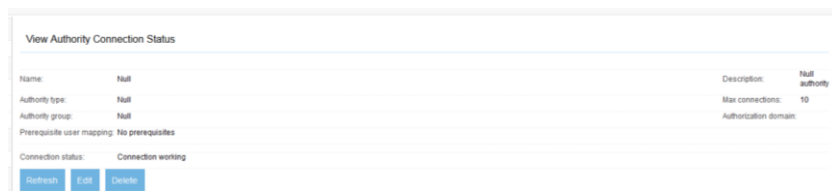

Figure 4.8 The view screen for our null authority example.

Now that we have a working null authority declared, let's spend a little time figuring out what it can do for us. As we'll learn, the main purpose of an authority is to provide access tokens through the ManifoldCF Authority Service. We will explore this next.

4.2.2   The Authority Service

The ManifoldCF Authority Service is a simple kind of API which supports only one kind of HTTP transaction. The transaction is always an HTTP GET, which accepts a user name (which is defined to be a domain-qualified name, e.g. `user@domain.com`) as an argument. The response is a list of access tokens, along with error information, if it turns out that the user was unknown or that the authority connection could not access the repository for any reason. This extra information may be helpful in that the user can be informed as to why they are not seeing results that they may expect.

REQUEST FORMAT

The Authority Service requires a specific URL path, along with certain arguments. A typical example URL looks something like `http://localhost:8345/mcf-authority-service/UserACLs?username=someone@somewhere.com`. The simplest canonical form is:

```
http[s]://<server>:<port>/mcf-authority-
service/UserACLs?domain=<authdomain>&username=<username>
```

Since we already have created a `Null` authority, let's go ahead and try using `curl` to see what the Authority Service returns. Try the following command:

```
curl http://localhost:8345/mcf-authority-
service/UserACLs?username=foo@bar.com
```

The result is something like this:

```
AUTHORIZED:Null+authority
TOKEN:Null:foo%40bar.com
```

If you had earlier set up a broken Active Directory authority, like we did in Chapter 2, you might have seen this as a response instead:

```
UNREACHABLEAUTHORITY:The+Spanish+Inquisition
TOKEN:My+Authority+Group:DEAD_AUTHORITY
AUTHORIZED:Null+authority
TOKEN:Null:foo%40bar.com
```

So, it seems to have worked! But what does it mean?

RESPONSE FORMAT

You will notice that the output consists of multiple lines, each of which has a format of *specifier*:*content*. We'll go over what the specifiers mean in a moment, but first let's look at the big picture. Some of this output (namely, the lines beginning with the specifier `TOKEN`) are the actual access tokens from the various authorities you have defined. The other lines tell us about the status of individual authorities. For these statuses, there is exactly one line per authority. For status lines, the value part of the line (to the right of the colon) is always the description field of the authority in question. The description is returned because the status lines are meant to be used to provide user-readable feedback. Their purpose is to be used to clue the user in as to whether the results the user is seeing are complete, or are incomplete due to some sort of authority error.

In the above example output, the authority that goes by the display name "Null authority" was working, and says it recognized the user it was given. It returned an access

token `Null:foo%40bar.com`, which makes sense since the user we passed in was "foo@bar.com", and the name of the authority group was "Null". The "%40" you will probably recognize as a standard percent-encoded "@" symbol. The authority that goes by the display name "The Spanish Inquisition", on the other hand, apparently could not be reached for comment, since there is an `UNREACHABLEAUTHORITY` specifier associated with it. Even so, there seems to be an access token from that authority, `My+Authority+Group:DEAD_AUTHORITY`. It is qualified with the name of the authority group that the Active Directory authority is part of, so it clearly came from there. But what is this `DEAD_AUTHORITY` token?

The answer is that an authority is responsible for returning the proper access tokens for the user *regardless of any error conditions*. But in a situation where an error occurred and ManifoldCF could not actually verify even that the user was valid, it must in some way signal to the search engine that any documents which require a valid Active Directory user should not be returned in the search results. The `DEAD_AUTHORITY` token is a way to make that happen; the understanding is that all repository connectors that are designed to work with the Active Directory authority should attach this token as a "deny" token to each document that must not be visible if the user is not a valid one.

Table 4.3 lists all the response specifiers, and what they mean.

Table 4.3 Response specifiers from the Authority Service, and their meanings.

Specifier	Meaning
TOKEN	Designates an access token. The content is the actual token string.
AUTHORIZED	Status that designates a successful lookup of the user, and that the user is valid. The content is the display name of the authority.
UNAUTHORIZED	Status that designates a successful lookup of the user, but the user is locked out or otherwise not valid. The content is the display name of the authority.
UNREACHABLEAUTHORITY	Status that designates an authority that is not functioning. The content is the display name of the authority.
USERNOTFOUND	Status that designates a working authority but an unknown user. The content is the display name of the authority.

So now we are beginning to understand what the Authority Service does. Next, we'll learn how to work with it programmatically, with the ultimate goal of using it to add support for security into a search engine.

Since the Authority Service uses HTTP, it seems perfectly reasonable for us to use commons-httpclient to communicate with it. In Chapter 3, we did something similar to interact with the API. The only difference is that the Authority Service is simpler – it only uses the GET verb – and the transaction does not involve JSON. It does, however, need to interpret the response from the Authority Service, not as JSON, but along the lines described earlier.

  Listing 4.1 shows the Httpcomponents Httpclient-based method that does these things. You   can   find   the   complete   code   at http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/security_example/src /org/apache/manifoldcf/examples/ManifoldCFAuthorityServiceConnect.java.

Listing 4.1 A method which uses Httpcomponents Httpclient to request access tokens from the Authority Service, and parse the output

```
public List<ResponseComponent> performAuthorityRequest(
  String authorizationDomain, String userName)
  throws IOException
{
  HttpClient client = HttpClients.createDefault();
  HttpGet method = new HttpGet(formURL(authorizationDomain,userName));
  try
  {
    HttpResponse httpResponse = client.execute(method);
    int response = httpResponse.getStatusLine().getStatusCode();
    if (response != HttpStatus.SC_OK)                              #A
      throw new IOException("Authority Service http GET error;"+
        " expected "+HttpStatus.SC_OK+", "+
        " saw "+Integer.toString(response));
    List<ResponseComponent> rval = new ArrayList<ResponseComponent>();
    InputStream is = httpResponse.getEntity().getContent();
    try
    {
      BufferedReader br = new BufferedReader(new InputStreamReader(is, #B
        UTF_8));
      while (true)
      {
        String line = br.readLine();                              #C
        if (line == null)
          break;
        rval.add(new ResponseComponent(line));                    #D
      }
      return rval;
    }
    finally
    {
      is.close();
    }
  }
  finally
  {
    method.abort();
```

```
        }
    }
```
#A Only valid response code is 200-OK
#B Response is always utf-8
#C Break the response into lines
#D Parse the line

Now that we have coded a way of communicating with the Authority Service, we're ready to go ahead and set up our search engine. We'll use Apache Solr for this example, since it, too, is an open-source Apache project. But the techniques we will develop will be equally applicable to most general-purpose search engines in the world today.

4.3   A secure search-engine integration

You may have noticed that the authority we created earlier is essentially meaningless on its own. In order to give it meaning, the search engine which receives content supposedly protected by that authority must be configured to actually enforce that security.

In this section, we'll tie up all the loose ends, and finally achieve the goal of a search engine that enforces document security. What we're going to do is effectively create two users, Fred and George, and index content in such a way that only Fred can see some of it, and only George can see the rest. Each set of content will have its own specific job, one specifying the "Fred" user, and the other specifying the "George" user. "Fred" will be allowed to see world news, while "George" can only see sports articles.

4.3.1   Laying the groundwork

The notion of Fred seeing only news and George seeing only sports has excited Bert. "Wow," he says, "you could use this functionality in lots of different ways." He's clearly thinking once again of creating his own security system, and using ManifoldCF's security model to give him direct control over who sees what.

You explain to Bert that, while it is technically true that you can control security completely through ManifoldCF, this is not what the software is intended to do. Rather, its main goal is to extend a repository's security model into search engine results, not set up a whole new security model. The fact that most connectors allow you to set security manually may be helpful in rare circumstances, but should never be considered the principle way ManifoldCF security would be deployed. Nevertheless, this feature turns out to be very handy for learning how the ManifoldCF security system works.

To implement our example, we've got some work to do. We're going to need to create a secure repository connection definition in ManifoldCF, for a start. We'll also need to set up the search engine. We'll be using Solr for this purpose, since it is readily available and easy to set up. Of course, we'll have to create a Solr output connection definition as well, and jobs that provide proper access tokens that we can use to demonstrate security. Then, we will be ready to actually extend Solr to apply that security.

CREATING A SECURED REPOSITORY CONNECTION DEFINITION

We've had experience now with the file system connector, the web connector, and the RSS connector. It really doesn't matter which one we choose to use, as long as we pick one whose job definitions have a `Security` tab.

   The whole purpose of this tab is to allow the user, on a per-job basis, to attach access tokens to all documents indexed by that job. This feature was originally included to support demonstrations by MetaCarta sales engineers, because, as you might imagine, it can be rather difficult to obtain access to sensitive systems in a sales situation. In a real-world environment, using a proprietary repository such as SharePoint or LiveLink, it would not be necessary to set up security using this tab, because these repositories support document security natively: SharePoint leveraging both its native model as well as Active Directory, and LiveLink with its own home-grown user security model. But rather than burdening the reader with the cost of obtaining such a system, we'll make do with one of ManifoldCF's supported open repositories, and use the `Security` tab to simulate what will happen automatically for a typical proprietary system.

   We have already set up an RSS connection definition once before, in Chapter 3, but we did this mostly by using the API. Let's do it again, but use the UI this time, and make sure we specify our newly-created authority connection definition this time. In the ManifoldCF user interface, click the `List repository connections` link. Then, a s you have done before, click the `Add new connection` link at the bottom of the page. For the `Name` tab, provide a meaningful name, maybe "Secured RSS". Then, click the `Type` button. See figure 4.9.
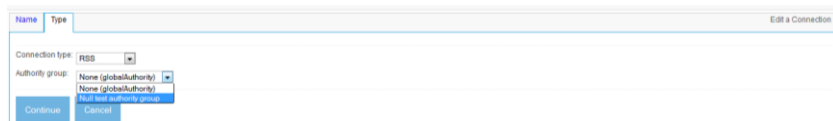

Figure 4.9 Select the RSS connection type, and an authority connection definition to secure the repository connection.

Once you've selected the RSS connection type, and the `Null` authority you created earlier, you may click the `Continue` button. Many additional tabs appear at this point, as captured in figure 4.10.
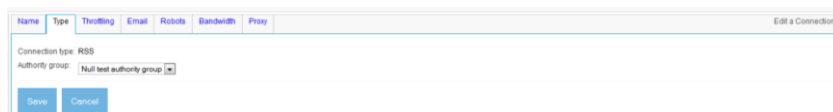

Figure 4.10 An RSS connection definition's tabs. The default settings are likely adequate for all except for the Throttling and Email tabs.

Chapter author name: Wright

The tabs you should actually visit are the `Throttling` and `Email` tabs. Set the maximum connections on the `Throttling` tab to 35. Set an appropriate email address on the `Email` tab. You can leave the defaults in place for all the others. Then, click the Save button. The connection definition view screen should look something like figure 4.11.
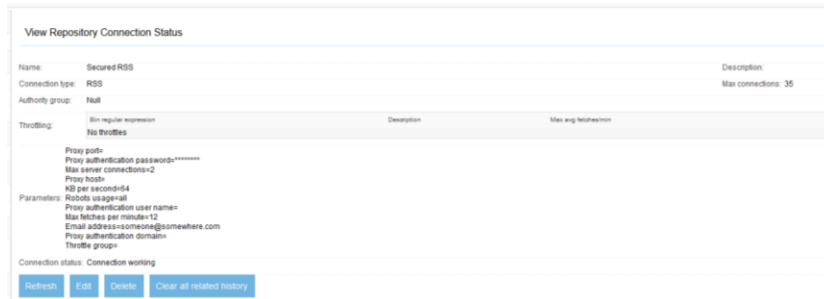

Figure 4.11 A screenshot of a properly configured, secured RSS connection definition. Note that the authority is listed as Null, which happens to be the name we chose for the null authority connection definition we set up earlier.

So now, we now have an authority connection definition, and a connection definition configured to use that authority. What else do we need to make our example complete?

The answer is that we need to set up a search engine, and use ManifoldCF to push secured content into that search engine. We'll tackle that next.

INSTALLING SOLR

Solr consists of two parts at the moment – the Solr framework, and the Lucene search engine. Lucene has been around for over a decade, while Solr is a somewhat newer invention. Together, Solr and Lucene represent a highly configurable and extendible search engine, bundled up in a convenient form.

To get started, we're going to need to set up Solr and Lucene as our search engine. Luckily this is very easy to do. Start by downloading the tar or zip file with the latest released version of Solr, and unpack it. At the time of this writing, the current latest release of Solr is 3.5.0, which includes Lucene 3.5.0.

Once you have unpacked Solr, open a shell in Unix, or a command prompt in Windows, and change to the Solr example directory. The directory contents should look something like this:

```
C:\solr\apache-solr-3.5.0\example>dir
 Volume in drive C has no label.
 Volume Serial Number is 00EF-E2CD

 Directory of C:\solr\apache-solr-3.5.0\example

06/16/2011  08:21 AM    <DIR>          .
06/16/2011  08:21 AM    <DIR>          ..
```

```
06/16/2011  08:21 AM    <DIR>          etc
06/16/2011  08:21 AM    <DIR>          example-DIH
06/16/2011  08:21 AM    <DIR>          exampledocs
06/16/2011  08:21 AM    <DIR>          lib
05/30/2011  10:51 PM    <DIR>          logs
06/16/2011  08:21 AM    <DIR>          multicore
06/16/2011  08:21 AM             2,120 README.txt
06/16/2011  08:21 AM    <DIR>          solr
06/16/2011  08:21 AM            17,241 start.jar
06/16/2011  08:21 AM    <DIR>          webapps
05/30/2011  10:51 PM    <DIR>          work
              2 File(s)         19,361 bytes
             11 Dir(s)  44,931,178,496 bytes free
```

From here, you can start Solr with the following simple command:

```
<path_to_java_binary> -jar start.jar
```

This will start the Solr web application under Apache Jetty, much like the ManifoldCF web applications use Jetty for the ManifoldCF Quick Start. Once started, the Solr example is configured to listen on port 8983. The example is further configured with several Solr request handlers active, which are designed to convert incoming web requests into actions. The ones we are going to be interested in are the standard search handler, the extracting update request handler, and the ping request handler. The first we'll use to query Solr for answers, while the latter two we'll use to get content into the search engine.

So let's go ahead and start Solr now, and use a browser to attempt to get results out of it. Since we haven't put in any documents yet, we should not expect any results, but this is still a useful exercise, to work out the kinks in our example process. So, once Solr is done starting, open a browser, and send it to the url `http://localhost:8983/solr/admin`. If you've done everything right, you should see a screen similar to figure 4.12.
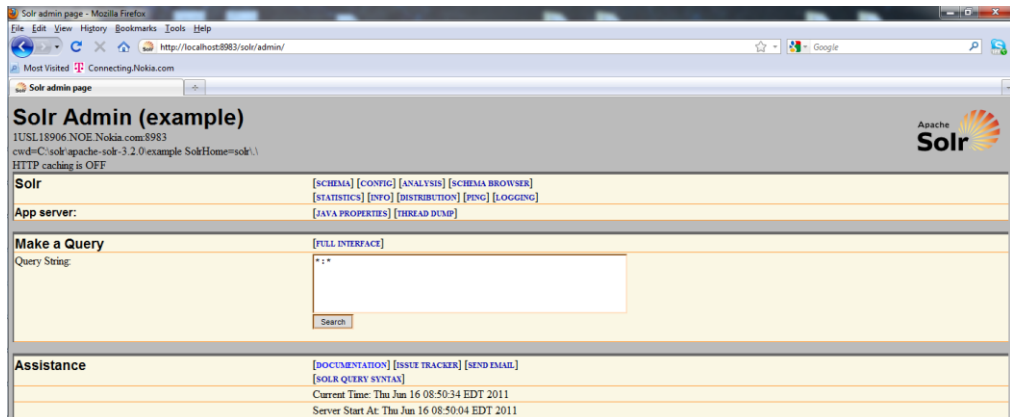

Figure 4.12 Solr admin page. Here we can make queries and inspect the contents of the Solr index.

Chapter author name: Wright

To execute a search, enter a query string and click the Query button. You should see a response that looks something like Figure 4.13.

If you got this far, don't be discouraged that you have zero results. We'll be fixing that in a moment. It looks like we are ready for content! Luckily, that's exactly what ManifoldCF is designed for.

CREATING A SOLR OUTPUT CONNECTION DEFINITION

To create a Solr output connection definition in ManifoldCF, open another browser instance and point it at the ManifoldCF Crawler UI. Click on the `List output connections` link, and then on the `Add a new output connection` link on the resulting page. Fill in the output connection definition `Name` tab with something reasonable, e.g. "Solr", and give your connection definition a decent verbal description as well. Then, click the `Type` tab, and select the `Solr` option from the connection type pulldown. Finally, click the `Continue` button. You will see a screen similar to figure 4.14.
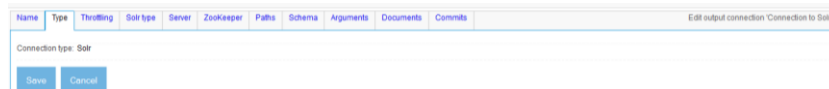

Figure 4.14 The tabs for a Solr output connection definition.

Feel free to visit all the tabs, if you so desire. But there's actually nothing you need to change! The default values are all acceptable, in this case. Click the `Save` button, and the view screen you should see should be similar to figure 4.15.

Figure 4.15 The view page of a successfully created Solr output connection definition. "Connection working" indicates that a connection using these parameters was successfully able to interact via the Solr Ping handler.

CREATING SECURED RSS JOB DEFINITIONS

Next, we will need to create a couple of RSS job definitions. We will need more than one, because the documents from each job will be visible by a different set of users! The choice of feeds is entirely up to you – however, most news sites have many such feeds. You have your choice of dozens from CNN alone.

To create the first RSS job with the proper security, point your browser again at the ManifoldCF Crawler UI. Click the `List all jobs` link, and then the `Add a new job` link. Given your job a name, such as "RSS documents seen only by Fred", and click the `Connection` tab. As you might expect, you will want to select the `Solr` output connection definition we just created, and the `Secured RSS` repository connection definition we created earlier. We're not going to be performing a continuous crawl, this time, because we're basically only interested in a snapshot. So it is okay to leave the rest of the `Connection` tab parameter in their default state. Click the `Continue` button. You should see a screen similar to figure 4.16.

Figure 4.16 The tabs on a secured RSS job. We'll need to visit the URLs tab and the Security tab this time.

Chapter author name: Wright

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=727

The tabs we need to work on are the `URLs` tab, where we need to add the feeds we decide that Fred can see, and the `Security` tab, where we need to make sure that all documents indexed from this job have the "Fred" access token. Let's tackle the URLs tab first. When you click on it, you see a blank field into which you can put all the newline-separated feed URLs you want Fred to be able to see, as in figure 4.17.
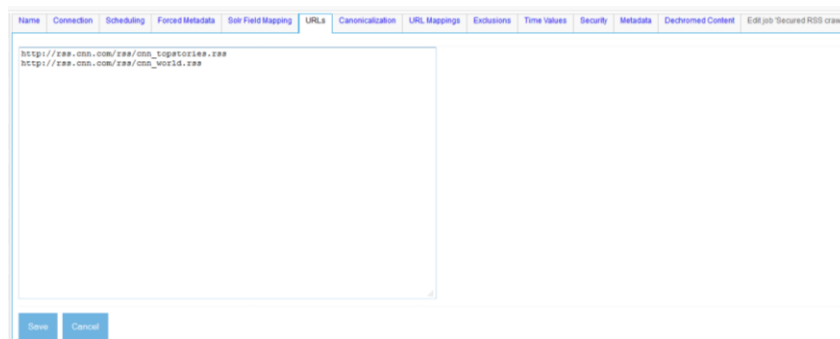
Figure 4.17 The URLs tab of an RSS job. Fred apparently can see top stories and world news.

Next, click the `Security` tab. Here we're going to list the access tokens that all documents belonging to this job should get. Type "Fred" into the box, and click the `Add` button. See figure 4.18.
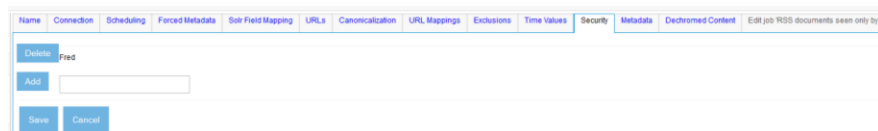
Figure 4.18 An RSS job Security tab, with the token "Fred" already added.

We are done with this job, so click the `Save` button.

We will also need to create another job with different access tokens, so that we have some documents with one set of tokens, and other documents with a different set. To achieve this, all you need to do is to retrace our steps and create a second RSS job. Call it "RSS documents seen only by George", and make the same selections for the `Connection` tab. You will need different feed URLs for the `URLs` tab. I've decided that George should see only sports documents. See figure 4.19.
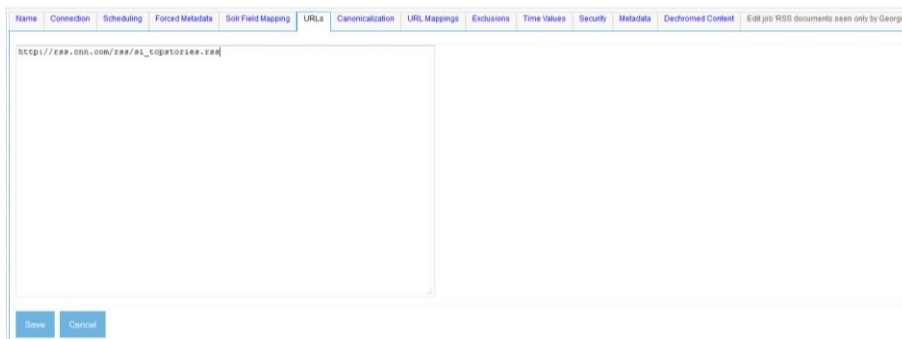
Figure 4.19 George's RSS sports feeds. Hopefully George's feeds do not share any documents in common with Fred's feeds. Since each job operates independently, the common documents will adopt the security of the last job they were crawled with.

This raises an interesting point. Each of the jobs we are creating has a different set of feeds. But there is no guarantee that the documents the feeds refer to are completely distinct from one another. What happens if, say, a story about a Super Bowl victory is included both under sports news and top stories? What does ManifoldCF do in cases like that?

The answer is that ManifoldCF treats the document as if it is owned by **both** jobs. But when the document is indexed, that indexing activity must happen under the direction of either one job or the other. The parameters of the job that were in effect at the time the document was last indexed are what will count.

Of course, there is nothing that stops the other job from coming along afterwards, and re-indexing the same document, using *completely different* job parameters. So we can make no apriori claims about such documents, because the last job to index the document "wins". But it will not be a fatal problem if a document turns out to be shared between the two jobs – it may wind up being visible by George one moment, and by Fred the next, but nothing really pathological will occur.

So let's go ahead and click the `Security` tab for this second job. Fill in the access token field with "George", and click the `Add` button. Then, click the `Save` button, because we are done with this job now, also.

A TRIAL RUN

We are now ready to take the steps to apply security to the Solr search results. But before we start messing around with the Solr configuration, let's run these two jobs with the default configuration intact, to see whether we get searchable results.

Browse to the Job Status page, and click the `Start` link for both jobs. After a short delay, the jobs should start off by reading the feeds and parsing them. Soon afterwards,

your Solr shell or command window should start to spit out many lines of text that looks something like this:

```
Nov 14, 2010 3:29:45 PM org.apache.solr.core.SolrCore execute
INFO: [] webapp=/solr path=/update/extract
params={literal.deny_token_document=Null:DEAD_AUTHORITY&literal.source=http
://rss.cnn.com/rss/cnn_world.rss&literal.id=http://rss.cnn.com/~r/rss/cnn_w
orld/~3/_8SnLLWc4i8/index.html&literal.allow_token_document=Null:Fred&liter
al.title=Sarkozy+reappoints+French+PM+one+day+after+resignation&literal.pub
date=1289730560000} status=0 QTime=15
Nov 14, 2010 3:29:50 PM org.apache.solr.update.processor.LogUpdateProcessor
finish
INFO: {add=[http://rss.cnn.com/~r/rss/cnn_world/~3/Em8W7lbQURs/index.html]}
0 14
```

This is Solr's way of logging the fact that it received a document for indexing. Note also that there are two parameters that seem suspiciously like they might be access tokens: `literal.deny_token_document`, and `literal.allow_token_document`. Indeed, for the allow argument, you can see a qualified access token that you recognize as being actually correct: `Null:Fred`. So it looks like things are working!

Let's go back now to the Solr admin page and try to do a search on some likely keywords. Remember that there is no actual search security in place yet; we're just doing a sanity check. Type in some likely word, such as "Afghanistan", into the search box and click the `Search` button. If you picked a reasonable word, you should get results, similar to those as shown in figure 4.20.
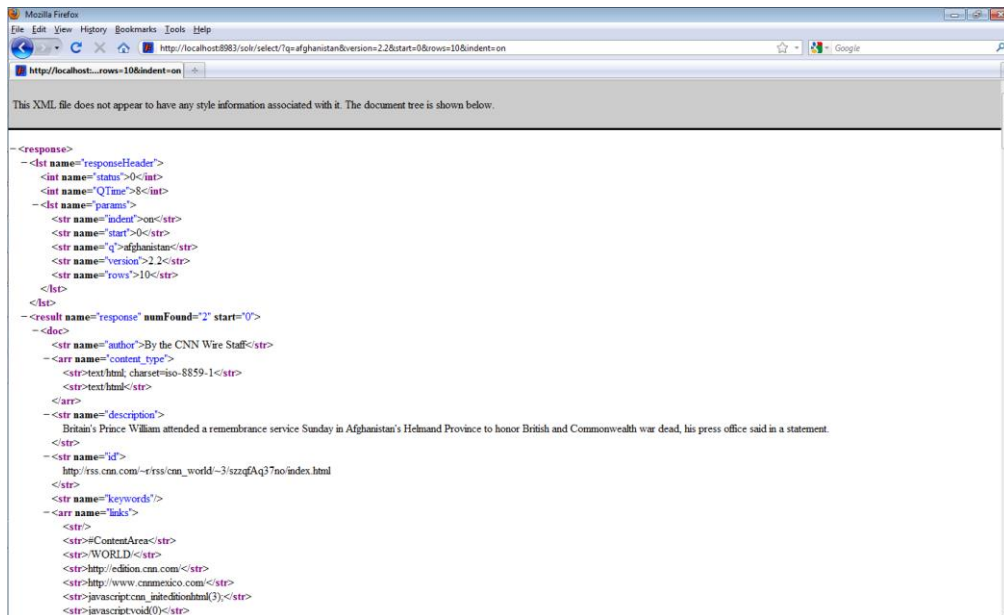

Figure 4.20 Unfiltered results from a search for "Afghanistan", after both of my secured jobs were run.

I am sure that you find this extremely exciting. For the first time, we have actually combined ManifoldCF with Solr to produce an artifact that returns useful results! We aren't done yet, but now we are finally ready to customize Solr to have it enforce ManifoldCF security.

4.3.2   Setting Solr up to enforce security

One of the reasons I've chosen Solr for this integration exercise is because it is quite easy to configure and extend. We're going to need to extend Solr in two ways: first, adding fields to the Solr document schema to keep ManifoldCF access tokens around, and second, adding a search component designed to filter documents using query restriction.

Of course, other search engines may or may not be as easy to deal with. But that, I'm afraid, is going to wind up being your problem, not mine.

MODIFYING THE SOLR SCHEMA

To keep the access tokens from ManifoldCF around so they can be used in search expressions, we will need to augment Solr's example schema in order to add the appropriate fields to it. The schema is determined by a file called schema.xml, which can be found in the Solr example under the solr/conf subdirectory, as seen below.

```
C:\solr\apache-solr-3.5.0\example>dir solr\conf
 Volume in drive C has no label.
 Volume Serial Number is 00EF-E2CD

 Directory of C:\solr\apache-solr-3.5.0\example\solr\conf

06/16/2011  08:21 AM    <DIR>          .
06/16/2011  08:21 AM    <DIR>          ..
06/16/2011  08:21 AM             1,125 admin-extra.html
06/16/2011  08:21 AM             1,310 elevate.xml
06/16/2011  08:21 AM            82,327 mapping-FoldToASCII.txt
06/16/2011  08:21 AM             3,114 mapping-ISOLatin1Accent.txt
06/16/2011  08:21 AM               894 protwords.txt
06/16/2011  08:21 AM            32,580 schema.xml
06/16/2011  08:21 AM               921 scripts.conf
06/16/2011  08:21 AM            57,575 solrconfig.xml
06/16/2011  08:21 AM                16 spellings.txt
06/16/2011  08:21 AM             1,229 stopwords.txt
06/16/2011  08:21 AM             1,148 synonyms.txt
06/16/2011  08:21 AM    <DIR>          velocity
06/16/2011  08:21 AM    <DIR>          xslt
              11 File(s)        182,239 bytes
               4 Dir(s)  44,891,688,960 bytes free
```

As we have seen, the Solr output connector uses preset field names for the access tokens it sends to Solr. The field names it uses are listed in table 4.4.

Table 4.4 Field names the Solr output connector uses for access tokens.

Field name	Meaning

`allow_token_document`	A multivalued set of allow tokens, at the document level
`deny_token_document`	A multivalued set of deny tokens, at the document level
`allow_token_parent`	A multivalued set of allow tokens, at the document parent level
`deny_token_parent`	A multivalued set of deny tokens, at the document parent level
`allow_token_share`	A multivalued set of allow tokens, at the share level
`deny_token_share`	A multivalued set of deny tokens, at the share level

So we need to add these fields to the Solr schema – somehow. But that turns out to be quite easy. If you examine the contents of the example `schema.xml`, you quickly can see how the file is laid out. All you need to do to add these six fields is add the following, pretty near anywhere:

```
<!-- Security fields -->
<field name="allow_token_document" type="string" indexed="true"
stored="false" multiValued="true" default="__no_security__"/>
<field name="deny_token_document" type="string" indexed="true"
stored="false" multiValued="true" default="__no_security__"/>
<field name="allow_token_parent" type="string" indexed="true"
stored="false" multiValued="true" default="__no_security__"/>
<field name="deny_token_parent" type="string" indexed="true" stored="false"
multiValued="true" default="__no_security__"/>
<field name="allow_token_share" type="string" indexed="true" stored="false"
multiValued="true" default="__no_security__"/>
<field name="deny_token_share" type="string" indexed="true" stored="false"
multiValued="true" default="__no_security__"/>
```

The purpose for the default values for the fields will be clearer in a moment. For now, type ^C in the Solr window to stop Solr, and then restart it. The changed `schema.xml` will go into effect at that time. The necessary schema changes are almost done! There's only one more thing you need to do to add the access tokens to the index – you will need to make ManifoldCF send them in again, so that Solr can record them.

To do this, you cannot just rerun the jobs. Remember that ManifoldCF is an incremental crawler; it will remember what the documents looked like and may well choose to not reindex anything. How should we get around this?

We could delete both RSS jobs, recreate them, and run them again. But that is a lot of work. There must be a better way!

Luckily, there **is** a better way. Way back in Chapter 1 we discussed a link that appears on the view page of every output connection definition, which is titled `Re-ingest all associated documents`. That is the link we need to click to signal to ManifoldCF that it needs to redo all documents for that connection definition, regardless of whether or not the

document itself has changed since the last indexing operation. So, navigate now to the view page for the Solr output connection definition you created earlier, and click that link. You will get a nice popup warning you about the implications of what you are about to do. Agree, and you are all set to run the jobs again.

Go to the Job Status page, and click the `Start` link for both jobs once again. The Solr log should again show numerous indexing requests for all the documents the feeds have found. When the jobs are done, the re-indexing is complete, and all the indexed documents should also have their proper access tokens within Solr.

CREATING A SECURITY SEARCH COMPONENT

Solr searching can be extended by adding specific components which modify or perform the search. These components are called Search Components, in Solr lingo. We're going to create one that takes a user name argument and uses it to add clauses to the incoming query. We'll use the `ManifoldCFAuthorityServiceConnect` class to perform the actual communication, and filter out all response components except for token values. It is left as an exercise for the student to modify this component to add hints to the response that would allow a client web UI to tell the user the reason their results are less than they expect.

> **Note** ManifoldCF has a set of integration packages prebuilt and available for people to use to integrate with Solr or with other systems. You can download packages for both Solr 3.x and Solr 4.x that do what this example does, or (starting in ManifoldCF 0.4-incubating) the integration jars and instructions are included in the dist/integration directory.

Our example creates a nested Solr structure called a *Filter*, which it then applies to the search results. The code recognizes the special meaning of having no access tokens whatsoever at a given level, and treats that in a separate conditional clause. See listing 4.2 for some of the highlights from the `ManifoldCFSecurityFilter` class. You can see the entire class at http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/security_example/src /org/apache/manifoldcf/examples/ManifoldCFSecurityFilter.java.

Listing 4.2 Portions of the ManifoldCFSecurityFilter class, demonstrating how access tokens are converted to Filters

```
public void prepare(ResponseBuilder rb) throws IOException
{
  SolrParams params = rb.req.getParams();                        #1

  String qry = (String)params.get(CommonParams.Q);
  if (qry != null)                                               #2
  {                                                              #2
    for (String ga : globalAllowed)                              #2
    {
```

Chapter author name: Wright

```
      if (qry.equalsIgnoreCase(ga.trim()))                          #2
        return;                                                     #2
    }                                                               #2
  }

  String authorizationDomain =
    params.get(AUTHORIZATION_DOMAIN_NAME);
  if (authorizationDomain == null)
    authorizationDomain = "";
  String authenticatedUserName =
    params.get(AUTHENTICATED_USER_NAME);

  if (authenticatedUserName == null)                                #3
    return;                                                         #3

  if (connection == null)
  {
    throw new SolrException(
      SolrException.ErrorCode.SERVER_ERROR,
      "Error initializing ManifoldCFSecurityFilter "+
      "component: 'AuthorityServiceBaseURL' init "+
      "parameter required");
  }

  List<String> userAccessTokens =                                  #4
    getAccessTokens(authorizationDomain,authenticatedUserName);    #4

  BooleanQuery bq = new BooleanQuery();


  Query allowShareOpen = new TermQuery(new Term(fieldAllowShare,
    EMPTY_FIELD_VALUE));
  Query denyShareOpen = new TermQuery(new Term(fieldDenyShare,
    EMPTY_FIELD_VALUE));

  Query allowParentOpen = new TermQuery(new Term(fieldAllowParent,
    EMPTY_FIELD_VALUE));
  Query denyParentOpen = new TermQuery(new Term(fieldDenyParent,
    EMPTY_FIELD_VALUE));

  Query allowDocumentOpen = new TermQuery(new Term(fieldAllowDocument,
    EMPTY_FIELD_VALUE));
  Query denyDocumentOpen = new TermQuery(new Term(fieldDenyDocument,
    EMPTY_FIELD_VALUE));

  if (userAccessTokens.size() == 0)                                #5
  {                                                                #5
    bq.add(allowShareOpen,BooleanClause.Occur.MUST);              #5
    bq.add(denyShareOpen,BooleanClause.Occur.MUST);               #5
    bq.add(allowParentOpen,BooleanClause.Occur.MUST);             #5
    bq.add(denyParentOpen,BooleanClause.Occur.MUST);              #5
    bq.add(allowDocumentOpen,BooleanClause.Occur.MUST);           #5
    bq.add(denyDocumentOpen,BooleanClause.Occur.MUST);            #5
  }
  else
  {
```

```
       bq.add(calculateCompleteSubquery(fieldAllowShare,fieldDenyShare,    #6
         allowShareOpen,denyShareOpen,userAccessTokens),                   #6
         BooleanClause.Occur.MUST);                                        #6
       bq.add(calculateCompleteSubquery(fieldAllowParent,fieldDenyParent,  #6
         allowParentOpen,denyParentOpen,userAccessTokens),                 #6
         BooleanClause.Occur.MUST);                                        #6
       bq.add(calculateCompleteSubquery(fieldAllowDocument,                #6
         fieldDenyDocument,allowDocumentOpen,                              #6
         denyDocumentOpen,userAccessTokens),                               #6
         BooleanClause.Occur.MUST);                                        #6
     }

     List<Query> list = rb.getFilters();
     if (list == null)
     {
       list = new ArrayList<Query>();
       rb.setFilters(list);
     }
     list.add(new ConstantScoreQuery(bf));
   }
```

   #1 Get the request parameters
   #2 Some kinds of queries go through without modification
   #3 If there was no authenticated user, don't modify the query
   #4 Use ManifoldCFAuthorityServiceConnect to get the access tokens
   #5 If no access tokens, only public documents must be returned
   #6 If access tokens, build filters for all six ways they can be present

At #1, this code gets the Solr request parameters, checking at #2 to see whether this is a kind of request that should not be modified. Similarly, if there is no authenticated user (at #3), by convention we choose to not modify the query either. (This prevents warmup queries from throwing errors.) We get the user's access tokens in the method call at #4, and treat the case of no access tokens (#5) differently from the standard case, at #6.

   Now we see why the schema needs a default value for each security field. The query we must build needs a fast way to identify documents that have no values for each security field. While we could have used Wildcard queries in Solr for this purpose, in practice that would be much slower than a direct Term query.

BUILDING THE EXAMPLE

Our search component class will, of course, need to compile against various Solr jars, and Httpcomponents Httpclient as well. If you like, you can use `svn` to check out the entire example, except for the configuration file changes, from `http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised`, as follows:

```
svn co
http://manifoldcfinaction.googlecode.com/svn/trunk/edition_2_revised/securi
ty_example
```

   Here you will find not only the sources and dependent libraries for the example, but also an ant build file. You can build the whole thing into a jar by entering the `security_example` directory, and typing `ant`, similar to the way you did back in Chapter 1

Chapter author name: Wright

when you built ManifoldCF. A jar called `authority-example.jar` will appear in the subdirectory `build/jar` when you do this. We'll need to place the jar containing the class in a directory that Solr will know to load it from. The directories that Solr searches for classes within are specified in another configuration file, this one called `solrconfig.xml`. It is in the same directory as the `schema.xml` file we already altered. If we examine this configuration file, we will find descriptions of the library paths that are included by default, using the `<lib dir="…"/>` XML tag. These paths are all relative to the Solr home path, which for the example is `example/solr`. We'll need to create a place for our new jar, so let's create the directory `example/solr/lib` for this purpose. Go ahead now and create this directory, and copy `authority-example.jar` into it. Then, add a line to the `solrconfig.xml` file that declares this directory as a place to look for jars:

```
<lib dir="./lib"/>
```

CONFIGURING THE SEARCH COMPONENT

The final step to enabling the component is to declare it in the `solrconfig.xml` file we have already seen. We will need to add two pieces: a declaration, and the use of that declaration. The declaration of the new search component should look like this:

```
<!-- ManifoldCF document security enforcement component -->
<searchComponent name="manifoldCFSecurity"
  class="org.apache.manifoldcf.examples.ManifoldCFSecurityFilter">
  <str name="AuthorityServiceBaseURL">http://localhost:8345/mcf-authority-
service</str>
</searchComponent>
```

To include this component in the search itself, we need to modify an existing section of `solrconfig.xml`, specifically the one that sets up the standard search handler. In Solr 3.x, this is called `search`, but in older versions of Solr this was called `standard`. The handler appears in the file in an XML clause like this:

```
<requestHandler name="search" class="solr.SearchHandler" default="true">
  ...
</requestHandler>
```

To this we will need to add our new component:

```
<requestHandler name="search" class="solr.SearchHandler" default="true">
  <arr name="last-components">
    <str>manifoldCFSecurity</str>
  </arr>
  ...
</requestHandler>
```

Here the new component is set to run after all the others. It will modify the search query at the end, after all the other search components have a crack at it.

After you've made all these changes, ^C and restart Solr, so that they take effect. When Solr comes back up, we're finally ready to try everything out! We won't be able to use the Solr admin UI for this test, since the query will require an additional parameter which the admin UI does not know how to send. That parameter is the `AuthenticatedUserName` argument. If you recall, the search component we wrote will not attempt to perform any query restriction if the argument isn't present. But when it is present, it will modify the

query to limit the responses according to the access tokens it finds for the user that is provided.

AUTHENTICATION

Bert ruefully accepts the fact that he messed up, and is suitably grateful that you've helped him find a decent technical solution. But he's unsure that your solution is complete. He's concerned that there is a "hole", in that anyone can simply tell Solr that they are a certain person, and Solr will believe them.

Bert's concern is absolutely justified, because our solution to the problem so far only includes authorization. It is missing the authentication component - the piece that positively identifies the user as being who she says she is. In a real system, how would Solr obtain the user name? I am afraid that that is, to a large extent, beyond the scope of this book. The user clearly must be authenticated, since there is no real security without that step.

Java architectures such as JAAS are designed to support that authentication process. A potential way that the user authentication might be done is by running Solr under an application server such as Apache Tomcat. Tomcat has some support for JAAS, so it would only be necessary to write or find the correct JAAS plugin capable of authenticating the user against the authority of your choice, which would typically be Active Directory.

For now, we're just going to pretend we've solved this problem, and make do with `curl`. We'll pass the faux "authenticated" username directly as a parameter. Try the following `curl` request:

```
curl
"http://localhost:8983/solr/select?q=Afghanistan&AuthenticatedUserName=Fred
"
```

If all went well, you should see an XML response similar to the one you received when you performed the same query (but without the security) under Solr's admin UI. (I'm making a presumption here, that Afghanistan will match at least some world affairs documents, which Fred can see.)

If that worked for you, then you may also try a similar query, but with the user name of George:

```
curl
"http://localhost:8983/solr/select?q=Afghanistan&AuthenticatedUserName=Geor
ge"
```

You should now see only sports-related documents that have something to do with Afghanistan.

Congratulations! You've successfully integrated ManifoldCF with Solr, and enforced repository security on the results!

4.4   Summary

In this chapter, we've explored document security in considerable depth. We've learned about the models that can potentially be used to project repository security to a search

engine, and we've discovered how ManifoldCF maps this security to search results using a query restriction model.

   This concludes the section of the book that deals with how ManifoldCF integrates with the world at large.   The next section begins a discussion of the internal architecture of ManifoldCF, with the ultimate goal in mind of teaching you how to write your own connectors.