1 Meet ManifoldCF

This chapter covers

- Understanding the problem of enterprise data synchronization
- Understanding ManifoldCF's function and utility
- Getting ManifoldCF built and running
- Performing a simple "hello world" crawl
- Learning about scalability and sizing

One evening, you get a call from your friend Bert. Bert has been busy building the ultimate search engine, and he waxes euphoric about its power and capabilities. But after twenty minutes, he admits that he has a problem, a big problem. He has no way to get content into his search engine from the myriad different content repositories his company has somehow managed to acquire over the years. Somehow, in all the excitement of setting up the world's greatest search engine, he has completely overlooked the basic requirement of loading it up with real company data.

Face it, connection software is a boring topic. There do not appear to be any new principles or cool algorithms involved. If it gets thought of at all, it's treated like incontinence underwear - people want to pretend it doesn't exist. They want to set it up, and forget it. But no matter how little respect it gets, it is a critical part of any modern enterprise.

As Bert discovered, enterprise software today consists of a large number of independent systems and subsystems, which perform services such as user authentication, document storage, and providing search capabilities. The trick is to get these systems to cooperate in a manner that meets the overall goals of the organization. Usually that means getting every last bit of value from each system involved. So, for example, you would not seriously consider building your own authentication mechanism if, for instance, Windows already provides one.

ManifoldCF was designed to tackle the problem of integrating content repositories, where documents are stored, with search engines, where documents are indexed and can be searched for. In this chapter, you will learn about the basics of ManifoldCF – what it is, what it is good for, how to build it, how to get running, and how to use it. You will also learn some of the rudiments of how it is put together.

1.1 The big picture

Before we really begin, let's try to understand the big picture – what ManifoldCF is, why it was developed, and where it fits in the broad scheme of things. The story of ManifoldCF begins with the enterprise universe, or at least that part of it that deals with content.

1.1.1 Life, the universe, and enterprise software

Figure 1.1 is a diagram from approximately 40,000 feet in altitude of ManifoldCF's role in the enterprise world. In it, a business person interacts with many disparate systems. These systems are fundamentally all unrelated, although, as we shall see, they may cooperate with each other to a greater or lesser extent.

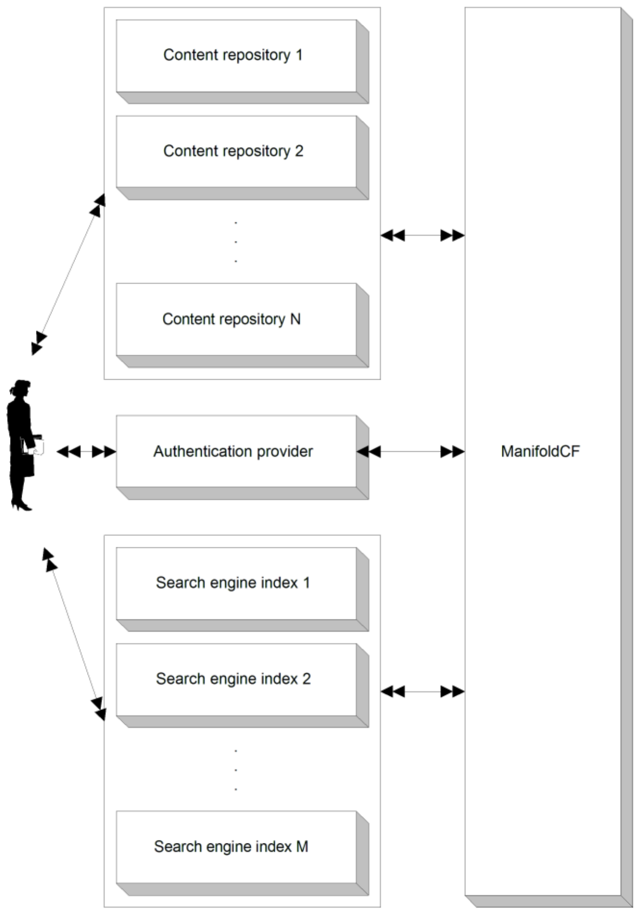


Figure 1.1: Some of the components of a typical modern enterprise, such as a medium-to-large-sized company or government agency. ManifoldCF solves the problem of keeping the search engine document indices up to date, and making sure an end user only sees search results for documents that they have access to.

In order to fully understand ManifoldCF's role in the enterprise, we will learn about each of these systems in turn – what they are for, what they do, and how they do it. By then it will be pretty obvious why ManifoldCF, or something similar, is utterly essential.

WHAT IS A CONTENT REPOSITORY, AND WHAT DOES IT DO?

Figure 1.1 features a number of *content repositories*. A content repository stores documents, usually with some notion of user ownership and hierarchical organization, and some metadata associated with those documents. A repository often has some kind of authorization model associated with it – that is, there are rules which describe what content can be accessed and changed by which users.

A formal definition follows.

> **Definition** A *repository* is a program or system intended to manage a collection of documents and document-related data.

Content repositories are interesting within enterprises in part because there are so many different kinds. Each kind has its own strengths and weaknesses. But even so, that does not explain why enterprises accumulate different content repositories more or less to the same degree that your grandmother collects old food containers.

WHAT IS A SEARCH ENGINE, AND HOW DO SEARCH ENGINES WORK?

Another of the major components in figure 1.1 is a *search engine*. A search engine is a tool that allows people to find documents quickly, using some search phrase or other criteria. In order to do this quickly, every search engine must build a data structure called an *index*, which maps the search criteria to matching documents using a small number of steps.

Typical search engines define documents in terms of fields. The document itself may be considered one such field, while other data which is related to the document is called metadata. Usually, each item of metadata has a name, and often you can mention these metadata field names in the search criteria when you are searching for the document.

Here is a formal definition of a search engine.

> **Definition** A *search engine* is a program or system whose purpose is to permit users to locate documents within a repository (or multiple repositories), based on some search criteria. The data structure it keeps that allows it to do that is called an *index*.

In addition, a properly implemented search engine must take care to present to its users only those documents they are actually allowed to see. This is important because often the results of a search will contain snippets or extracts of each document found. Imagine the difficulties that might arise if the search engine routinely produced bits and pieces of, say, sensitive corporate strategy or unannounced quarterly forecasts. Luckily, ManifoldCF makes security control like this straightforward to implement, as we shall see.

AUTHENTICATION AND AUTHORIZATION

Figure 1.1 does not tell us much about the mechanisms or components that are used to control access to content. That's because much of the functionality involved in making security decisions in an enterprise is actually part of the other components of the system. Many content repositories, for example, have built-in security mechanisms, which do not appear explicitly in the figure, but are nevertheless essential.

There is, however, a box labeled *authentication provider*, which is a security related function. Indeed, authentication is half the work of providing security. Authorization is the other half. Let's formally define these terms.

> **Definition** *Authentication* is the mechanism by which an individual computer user demonstrates to a computer system that they are who they say they are.

> **Definition** *Authorization* is the mechanism by which a computer determines which authenticated user is allowed to perform which activities.

To take this a bit further, if authorization is the mechanism, then there must be an entity who does the authorizing, just like an "authenticator" or "authentication provider" might be what you call an entity that performs authentication. In this book we call a provider of authorization an *authority*.

> **Definition** An *authority* is a program or system whose job it is to control the ability for individual users to access specific resources, such as documents.

SYNCHRONIZING DATA

In order for all the systems in figure 1.1 to function optimally, they must share a reasonably consistent view of the documents within the enterprise. For instance, a search engine is not very helpful if its indexes refer to data that no longer exists, or do not contain appropriate references to data that does exist. Providing a mechanism for this to take place is the job of the box in the figure labeled ManifoldCF, which is, after all, the main topic of this book.

1.2 What is ManifoldCF?

ManifoldCF is a tool designed to synchronize documents and their metadata from multiple repositories, with some set of search engines or similar targets. It is designed to do this feat not merely once, but over an extended period of time. It supports multiple models of synchronization, and several varieties of each model.

This package also provides infrastructure for enforcing document security. Most repositories have some way of controlling which user can see what document, and good security policy demands that those rules be enforced, wherever the documents eventually wind up. ManifoldCF provides the means to do that.

ManifoldCF is meant to be extended to new kinds of repositories or indexes by the straightforward addition of new connectors. At the time of this writing, ManifoldCF comes with repository connector implementations for many kinds of interesting repositories, both open and proprietary: EMC's Documentum, IBM's FileNet, OpenText's LiveLink, Autonomy's Meridio, Microsoft's SharePoint, Microsoft's Active Directory shares, Oracle, Sybase, and

MSSQL databases, native file system, RSS feeds, and the general web. It also supports a number of different output connectors, to indexes such as Solr and MetaCarta's GTS.

In short, ManifoldCF is all about synchronizing data. We'll go into this topic in great depth in the next section.

1.2.1 Techniques for data synchronization

There are two basic techniques for synchronizing data across data repositories. One kind is the *push* model. In this model, changes to one data store get "pushed" to the other as part of the change itself. If I change a document in the first repository, not only is the document changed there, but as part of the same transaction the document is also changed in the second repository. The second model is called the *pull* model. In this model, changes to a repository take place without any side effects or notification. Later, a program asks the repository what changed, and transmits those changes to the second repository.

The program that executes the second model is often called a *crawler*. The name is not a reference to its speed, but more a reference to how it discovers changes; often a document must be fetched in order to discover more documents that need to be fetched. What the crawler actually does is called *crawling*. For the rest of the book, whenever I use the word "crawling", I am referring to synchronization using a pull model.

See figure 1.2 for a depiction of pull and push models.

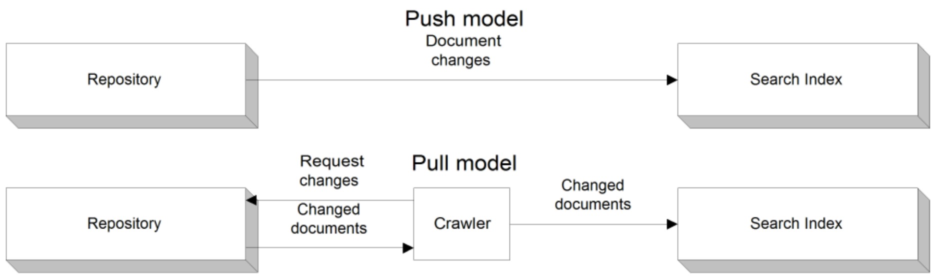


Figure 1.2: A pictorial description of the push model versus the pull model. Note that the pull model requires a crawler component.

We'll look at each of these models in more detail below.

PUSH MODEL

Why would anyone want to use a pull model, if a push model was a possibility? After all, it appears that there is a lot less work involved with push models – there's no need to discover changes after-the-fact, for instance. Indeed, there are many advantages to the push model – and some subtle disadvantages, which unfortunately prove to be its undoing in most real-world situations.

The most important disadvantage is that there's an implicit "perfection" requirement. The push model works well only insofar as every interesting change is noted and every change is actually written in a timely manner to both repositories. Whenever these conditions are not met, there is often no way to recover under the push model.

For example, suppose that the two repositories are connected by a network, and the network goes down just before a document is changed in the first repository. The second repository never sees subsequent notifications, and even after the connection is restored, there is no way to recover from this loss. It is possible to build into the first repository's notification feature the ability to save notifications, so that the second repository can be reminded after-the-fact about changes when the connection is restored, but that places an additional implementation requirement on the software that is managing the first repository.

Similarly, it's typical for repository software to overlook certain kinds of changes that an index would need to know about (such as changes to a document's security settings), and neglect to send a change notification in those cases. Once again, there is no way in a push model to recover from a missing feature without modifying the repository software itself.

Finally, the people who are in charge of the first repository may be unhappy about slowing down their transactions in order to allow both repositories a chance to save a document. Indeed, there may already be repository performance issues. Or (as is often the case) the people interested in indexing the documents do not have the same boss as the people responsible for running the repository, so the incentives are not aligned.

CRAWLING

If **any** of the above cases apply, then a push model won't work, and a pull model is the only option left. But once you agree that crawling is necessary, it turns out that there are a number of ways to go about actually doing it. Each of which may be appropriate in some circumstances, and inappropriate in others.

ManifoldCF synchronization models

ManifoldCF exclusively uses a pull model today, and is thus technically a crawler, although it is structured internally to support both pull and push components. During its development it was hypothesized that, just perhaps, a perfect repository would come along someday for which a push model would be appropriate, which led to certain architectural features. I'll talk more about that in Chapter 10, but suffice it to say that a commercial repository suitable for the push model has not yet been encountered by this author.

CRAWL ONCE

Sometimes it makes sense to just grab a snapshot of all the content you are interested in. Perhaps you are running an experiment on this content, or you intend to recrawl from scratch every time you want to refresh your search engine's document index. This model I'll call the *crawl-once* model. See figure 1.3.

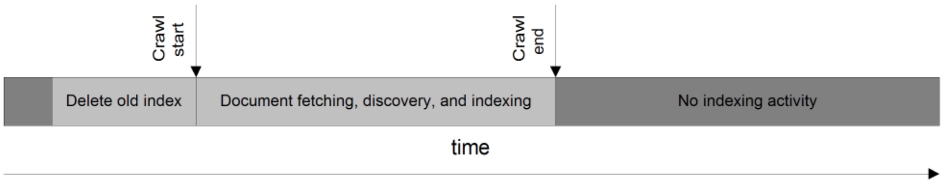


Figure 1.3: A pictorial representation of the crawl-once model. Note the pre-crawl removal of the old index.

The crawl-once model has some advantages. Specifically, it can be made very fast, since there are fewer data management requirements under this model. On the downside, every crawl places a tremendous burden on the repository that's being crawled, and absolutely no intelligence can be put towards reducing that burden. The burden is similar on the index, because in order to deal with deletions of data, you must delete the index entirely before each crawl, and rebuild it completely. The index must therefore be taken offline for the duration of the crawling process – or, at least, a complete copy must be retained.

Incremental Crawling

Incremental crawling is an alternative to the crawl-once model. The incremental crawling model presumes that you will be running the same crawl multiple times. It can therefore use that fact to make more intelligent decisions as to what data needs to be refetched and reindexed. See figure 1.4.

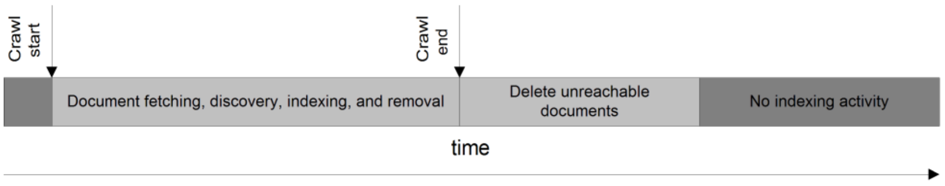

Crawl start
Crawl end
Document fetching, discovery, indexing, and removal
Delete unreachable documents
No indexing activity
time

Figure 1.4: A time-based picture of the events and periods needed to perform incremental crawling. There is no need to shut down the index during this crawl, since the index is never deleted.

Each crawl still takes place as a single, well-defined sequence of events, and at the end of each crawl the index is synchronized with the repository. The only difference between this model and the crawl-once model is in the relative amounts of work the crawler does versus the repository and index.

Incremental crawling has a much larger data management requirement than crawl-once crawling. More information must be kept around in order to be able to compare the current crawl with what happened on previous crawls. However, this model has the distinct advantage that it greatly reduces the burden on both content repositories and indexes – and, furthermore, there is no need to shut the index down while recrawling takes place.

CONTINUOUS CRAWLING

Incremental crawling relies on being able to tell if a document has changed from one crawl to the next. But sometimes it is difficult to determine this, at least without doing most of the work of actually crawling. This situation often occurs in a real-world web environment, where not only do documents often have "chrome" (the navigation and advertising cruft having nothing to do with the document itself), but also cannot be reliably compared to each other without the expensive operation of actually fetching the document from the server.

For situations like this, a better model is to give each document a schedule of its own. Once it has been fetched, it might be checked for changes at algorithmically determined intervals, or it may "expire" after some period of time. This document-centric kind of crawling is called *continuous crawling*. See figure 1.5.

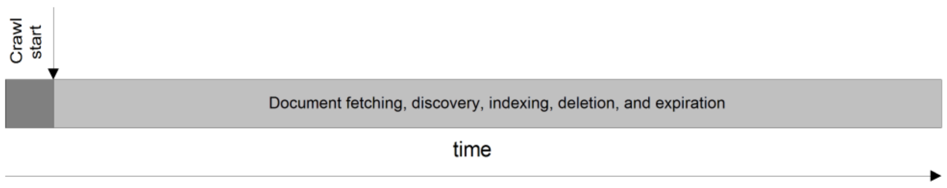

Crawl start
Document fetching, discovery, indexing, deletion, and expiration
time

Figure 1.5: The time-based pictorial of continuous crawling shows that it doesn't really have independent phases. It continues until manually stopped.

It is worth noting that continuous crawling seems to be missing a phase – the "delete unreachable documents" phase. This phase is intended to remove documents which are either no longer included in the document specification, or are no longer reachable because references to them within the crawled content no longer exist. Indeed, because continuous crawling runs forever, there is no point where it is possible to decide when a document is no longer reachable. Continuous crawling must therefore do without that functionality. Documents may still be deleted or expired, however.

1.2.2 Why ManifoldCF?

The next time you talk with Bert, he is thinking of writing his own crawler. After all, he reasons, he can capture some 75% of his documents by connecting to only some four or five different repositories. He now believes that his biggest problem will be how much he's going to have to learn about each kind of repository he needs to crawl before he can write the software.

Bert's concern about the required depth of domain knowledge is certainly justified. Many people make excellent livings integrating software with specific content repositories, because their experience level with API's that are often poorly documented is a valuable and marketable commodity. This is mercilessly multiplied because the number of different kinds of document repository that are at large in the world today is huge. Even more frighteningly, more are added every year.

But what you politely tell Bert is that he's missing a bigger point. While it is easy to write a bad crawler, it's not so easy to write a good one. Complex issues arise that will not even occur to a novice - for example, the interaction between throttling and document scheduling, or how to deal with document security across disparate kinds of repository. In fact, these problems are extensive. A small subset is listed below.

- *Repeated synchronization* – how to keep systems up to date with each other over an extended period of time
- *Throttling* – how to prevent your crawler from overwhelming the target systems
- *Scheduling* – when to do all the work
- *Performance* – how to keep things running at high efficiency, especially in the context of throttling
- *Handling diversity* – how to deal with very different systems in a unified manner
- *Security* – making sure each repository’s own security model is enforced
- *Dealing with errors* – doing the right thing with errors that may arise

In a real sense, ManifoldCF represents the years of effort required to turn a bad crawler into one that is "good enough" to deal with all of the issues listed above. A solid framework deals with the synchronization issues, and a wide variety of existing connectors allows most people to dodge the domain knowledge issues. A geographic search company called MetaCarta, Inc. originally wrote it. In the process, it produced at least a half-dozen different crawlers since its founding, but only one of them finally evolved to the point where it had all the right resilience characteristics, crawled incrementally, could be set up and largely forgotten, and performed well enough. That crawler evolved into ManifoldCF.

1.3 Setting things up

Let’s get started working with ManifoldCF in earnest. By the end of this book, we’re going to want to be able to develop connectors, and even before that, we’re going to be learning how to interact with ManifoldCF through many mechanisms. In order to do that, we’re going to need to build it and run it, in preparation for to a simple “hello world” crawl.

1.3.1 Building the software

We will need to have a working copy of ManifoldCF that we can run. You can just download the latest binary release and run it. But upon occasion you may want to inspect the source code for yourself, or perhaps you might want to take advantage of ManifoldCF's build infrastructure, so being able to build the software can come in handy.

> **Note** Releases prior to 0.5-incubating required you to build the software in order to support connectors for proprietary repositories such as Documentum. As of the 0.5-incubating release, this is no longer true.

ManifoldCF releases include three artifacts:

- an `src` artifact, which contains just the source code
- a `lib` artifact, which consists solely of binary dependencies needed to build the source code
- a `bin` artifact, which contains an already-built ManifoldCF, without any proprietary binary materials

If you want to build a released version of ManifoldCF yourself, you will need to download and unpack the `src` artifact. But since this artifact does not include any of the required binary dependencies (in accordance with Apache release policies), you will also need to download and unpack the `lib` artifact, and copy its contents into a `lib` folder within the unpacked source tree before you can build. The `lib` artifact's `README.txt` file describes exactly how to do this in more detail.

Another way to build ManifoldCF is by checking it out from the Apache repository using the Subversion source-control tool. If you are familiar with Subversion and you think you might be contributing patches or needing fixes and enhancements, this is the best way to proceed. This is what you will need:

- An installed version of the Sun or Open JDK, 1.7.x or above;
- An installed version of Apache Ant, 1.7.x or above;
- An installed version of a Subversion client, 1.7 or above.

> **Note** Some readers may wish to use Apache Maven in order to build ManifoldCF. ManifoldCF does support Maven builds in a limited way, but includes just those connectors which have only open-source dependencies. Since the primary build system of ManifoldCF is Apache Ant, in this book I will exclusively use Ant. Instructions for building ManifoldCF with Maven can be found at http://manifoldcf.apache.org/releases/trunk/connectors/en_US/how-to-build-and-deploy.html.

For Debian Linux, or for Ubuntu, these packages are available via the standard package mirrors, so you can trivially install them using the Debian `apt-get install` commands. For Windows, getting the prerequisites installed is a little more involved. You can download an appropriate JDK from the Oracle site (formerly Sun), after filling in some contact details. A Windows installer for Apache Ant can be downloaded from the Apache Ant site. If you decide to pursue the source checkout route, you will need Subversion also. For a good prebuilt `svn` client for Windows, I'd recommend SlikSvn, which provides a nice Windows installer.

Before you check anything out, you will probably want to go to the ManifoldCF website to verify what the `svn` path for the ManifoldCF tree actually is, and what the current release is labeled. You can list the released versions of ManifoldCF using this command (which is valid both in a Windows command prompt, and on Linux):

```
svn list http://svn.apache.org/repos/asf/manifoldcf/tags
```

If everything is installed properly, you should see a list of releases. You probably will want the latest one, so the next step is to check that release out. For example:

```
svn co https://svn.apache.org/repos/asf/manifoldcf/tags/release-1.5.1 mcf-
1.5.1
```

This will create a directory underneath the current directory called `mcf-1.5.1`, which will contain everything from the ManifoldCF 1.5.1 release, including everything needed to build it. Most source files are located under the `mcf-1.5.1/framework` and `mcf-1.5.1/connectors` directories. The root of the `ant` build is the main directory `mcf-1.5.1`, so before we begin you will want to change to that directory:

```
cd mcf-1.5.1
```

If you are using a released version of ManifoldCF, you should download and unpack the corresponding `lib` artifact. If you choose to use the trunk version of ManifoldCF, you will need to obtain the required binary dependencies. To do that, execute the following command from the shell or command line:

```
ant make-core-deps
```

Note If you've already downloaded, unpacked, and properly installed the lib artifact, you will not need to do this step; the lib artifact contains all the binary dependencies you need.

You will probably also want to download the binary dependencies that cannot be redistributed under the Apache license. This is optional, but without doing it, the documentation and some connectors will not be built. Here's the ant command for that step:

```
ant make-deps
```

Now that dependencies are present, you can build and/or test as many times as you need to with the following command:

```
ant build test
```

That's it! In a few minutes, your build and test run should complete. If it is successful, the ManifoldCF version you have built will have also passed a number of end-to-end tests and is ready to run.

> **Warning** If you have multiple versions of a Java JDK installed, you will need to tell `ant` which one to use. To do that, you must set an environment variable called `JAVA_HOME` to point to your JDK instance before running `ant`. On Linux, make especially sure that you using a Sun or Open JDK, because there are a number of open-source java implementations that might otherwise be installed. Some of these open-source implementations are not particularly compatible with the Sun JDK, and the build may fail as a result.

1.3.2 Running the software

Once the ant build is complete, or when you unpack the `bin` artifact, you will have a working single-process example version of ManifoldCF. This will be located under the `dist/example` directory if you built ManifoldCF yourself, or under just `example` if you unpacked the binary artifact rather than building. The example uses Apache Derby for a database, and runs Apache Jetty for its web applications on port 8345, and is called the *Quick Start example*, or sometimes the *Single-Process example*.

> **Note** If you build ManifoldCF yourself, you might notice a `dist/example-proprietary` directory also. This directory contains a version of the Quick Start example that includes connectors and binaries that rely on any proprietary or non-Apache-licensable binaries that you have included. If you need to use any such connectors, you will need to use this version of the example, rather than the non-proprietary one.

After running ant, all you will need to do to run ManifoldCF is the following, on Windows:

```
cd dist\example
"%JAVA_HOME%\bin\java" -jar start.jar
```

Or, on Linux:

```
cd dist/example
"$JAVA_HOME/bin/java" -jar start.jar
```

If all went well, you will see statements indicating that the connectors you built have been registered:

```
Successfully unregistered all output connectors
Successfully unregistered all authority connectors
Successfully unregistered all repository connectors
Successfully registered output connector
'org.apache.manifoldcf.agents.output.solr.SolrConnector'
Successfully registered output connector
'org.apache.manifoldcf.agents.output.nullconnector.NullConnector'
Successfully registered output connector
'org.apache.manifoldcf.agents.output.gts.GTSConnector'
Successfully registered authority connector
'org.apache.manifoldcf.authorities.authorities.activedirectory.ActiveDirect
oryAuthority'
Successfully registered repository connector
'org.apache.manifoldcf.crawler.connectors.webcrawler.WebcrawlerConnector'
Successfully registered repository connector
'org.apache.manifoldcf.crawler.connectors.rss.RSSConnector'
Successfully registered repository connector
'org.apache.manifoldcf.crawler.connectors.jdbc.JDBCConnector'
Successfully registered repository connector
'org.apache.manifoldcf.crawler.connectors.filesystem.FileConnector'
```

The connectors for the example are all declared in the xml configuration file `connectors.xml`. The `connectors.xml` file can be found in the `dist` directory, right above the one you run the Quick Start from. The ManifoldCF build automatically constructs this file based on which connectors were built. So, if you want to add your own connector to the build, you will not need to directly edit `connectors.xml` yourself – provided you add your connector to the root-level `build.xml` file properly. We'll go into that in more detail in Chapter 6.

The example will then unpack the ManifoldCF web applications, start Jetty, and then start the crawler itself:

```
Starting jetty...
2010-09-04 09:01:41.250:INFO::Logging to STDERR via
org.mortbay.log.StdErrLog
2010-09-04 09:01:41.406:INFO::jetty-6.1.22
2010-09-04 09:01:41.921:INFO::Extract war/mcf-crawler-ui.war to
C:\DOCUME~1\kawright\LOCALS~1\Temp\Jetty_0_0_0_0_8345_mcf.crawler.ui.war__m
cf.crawler.ui__3gfw37\webapp
2010-09-04 09:01:46.546:INFO::Extract war/mcf-authority-service.war to
C:\DOCUME~1\kawright\LOCALS~1\Temp\Jetty_0_0_0_0_8345_mcf.authority.service
.war__mcf.authority.service__s8m3kj\webapp
2010-09-04 09:01:48.906:INFO::Extract war/mcf-api-service.war to
C:\DOCUME~1\kawright\LOCALS~1\Temp\Jetty_0_0_0_0_8345_mcf.api.service.war__
mcf.api.service__.q3qt3h\webapp
2010-09-04 09:01:52.953:INFO::Started SocketConnector@0.0.0.0:8345
Jetty started.
Starting crawler...
```

By now you may be realizing that there seem to be quite a number of bits and pieces that make up ManifoldCF. Before we proceed any further, let's introduce them.

1.3.3 Software components

ManifoldCF consists of both software dependencies, such as the underlying database, and pieces that belong wholly to ManifoldCF. The latter includes the crawler process itself, and several web applications which provide the means of interaction between ManifoldCF and users or software programs.

Database

ManifoldCF is built upon a relational SQL database. Many of its desirable operating characteristics stem from this one design decision. The requirements of the database are covered in more detail in Chapter 11.

ManifoldCF includes a database abstraction layer, and can in theory work with a wide variety of database solutions. However, supported databases are currently limited to PostgreSQL, Hsqldb, MySQL, and Apache Derby. The Quick Start example by default uses an embedded Apache Derby database. This has some limitations, however. While Derby makes the Quick Start example very easy to work with, Derby has a number of problems and limitations which make it insufficiently powerful to support serious ManifoldCF crawling. I therefore highly recommend using Quick Start with PostgreSQL, if you like the convenience but don't want to rely on Derby.

A note about choosing an underlying database for ManifoldCF

There are many considerations that arise when selecting which database to run ManifoldCF on top of. While it is simple in practice to get a database to function with ManifoldCF, not all databases are created equally. The differences are most profound in two areas: ability, and performance. A database's abilities (or lack thereof) show up mainly in whether it can support all of ManifoldCF's reports fully. A database's performance characteristics will directly affect ManifoldCF's scalability.

At the time of this writing, the databases that have adequate abilities are Apache Derby, PostgreSQL, and MySQL. The databases that have good performance are PostgreSQL, MySQL, and Hsqldb.

Setting up ManifoldCF to use PostgreSQL is simple enough.

1. Install PostgreSQL. This is available from http://www.postgresql.org/download.
2. Change the `properties.xml` file property `org.apache.manifoldcf.databaseimplementationclass`, to have a value of `org.apache.manifoldcf.core.database.DBInterfacePostgreSQL`. The `properties.xml` file can be found in the `dist/example` folder, where you go to start Quick Start.
3. Add the property `org.apache.manifoldcf.dbsuperusername` and set it to the name of your PostgreSQL super user, usually `postgres`.
4. Add the property `org.apache.manifoldcf.dbsuperuserpassword` and set it to your PostgreSQL super user's password.
5. Run the Quick Start in the normal way

Note While PostgreSQL is the preferred database to use with ManifoldCF at this time, it does require some periodic maintenance, especially when used heavily for long periods of time. The maintenance procedures are available online at http://manifoldcf.apache.org/releases/trunk/connectors/en_US/how-to-build-and-deploy.html#A+note+about+maintenance.

AGENTS PROCESS

The Agents Process is the piece of ManifoldCF that actually does the crawling. It runs many threads, some which process individual documents, some which find documents to be processed, and some that do other things entirely, like start and stop jobs. It's called the *Agents* process because it is, in theory, capable of handling more than one kind of repository synchronization agent – the crawler, or "pull-agent", being only one such possibility. We will discuss this in greater depth in chapters 10 and 11.

CRAWLER UI WEB APPLICATION

The ManifoldCF Crawler UI is structured as a standard java web application, built on JSP technology, and it can be readily deployed on other web servers besides the Jetty server that the Quick Start example uses, such as Apache Tomcat. The Crawler UI allows you to create connection definitions, define and run jobs, and examine what is currently going on, or what happened in the past. Shortly we will learn how to use the Crawler UI to set up a basic crawl.

AUTHORITY SERVICE WEB APPLICATION

ManifoldCF includes a servlet web application designed to help people integrate document security into their applications. Because many repositories have strong notions of security, ManifoldCF has significant functionality in place which is designed to help search applications enforce multiple repository notions of security. The Authority Service web application is one part of that security functionality. The Authority Service will locate a user's access tokens, given the user name, for the purpose of a search engine enforcing security.

You can ask the Authority Service for a user's access tokens by doing the right HTTP GET request, e.g.:

```
curl "http://localhost:8345/mcf-authority-
service/UserACLs?username=user@domain.name"
```

We will discuss ManifoldCF's security model at great length in subsequent chapters. You may want to consider installing the `curl` utility on your computer at this time. You will likely find it to be a very useful tool for working ManifoldCF's HTTP services.

API SERVICE WEB APPLICATION

ManifoldCF includes a RESTful API service, which uses JSON for its object composition. The service is structured as a servlet-based web application. You can use `curl` to easily access this service as well:

```
curl http://localhost:8345/mcf-api-service/json/repositoryconnectors
{"repositoryconnector":[
  {"description":"Filesystem",
  "class_name":
      "org.apache.ManifoldCF.crawler.connectors.filesystem.FileConnector"},
    {"description":"JDBC",
    "class_name":
      "org.apache.ManifoldCF.crawler.connectors.jdbc.JDBCConnector"},
    {"description":"RSS",
    "class_name":
      "org.apache.ManifoldCF.crawler.connectors.rss.RSSConnector"},
    {"description":"Web",
    "class_name":

  "org.apache.ManifoldCF.crawler.connectors.webcrawler.WebcrawlerConnector"}
  ]}
```

The response above is a JSON document that lists all the currently-registered repository connectors. Complete ManifoldCF API documentation can be found online at:

http://manifoldcf.apache.org/releases/trunk/en_US/programmatic-operation.html

THE QUICK START EXAMPLE

The Quick Start example combines all the components of ManifoldCF into one single process, combining the database, Agents Process, and all three web applications together, for convenience.

> **Note** Running all of ManifoldCF in a single process is not the only way to deploy ManifoldCF. The following online materials describe two ways to deploy ManifoldCF in a multiprocess environment: http://manifoldcf.apache.org/releases/trunk/en_US/how-to-build-and-deploy.html. There you can learn about the Multiprocess example, which is located in the directory `dist/multiprocess-example`, and the Multiprocess ZooKeeper example, located in the directory `dist/multiprocess-zk-example`. There is also a combined war deployment which includes all of ManifoldCF in a single war that can be deployed under an application server such as Apache Tomcat.

Now that we know what the pieces are, let's make sure that everything seems to be in good order after the build.

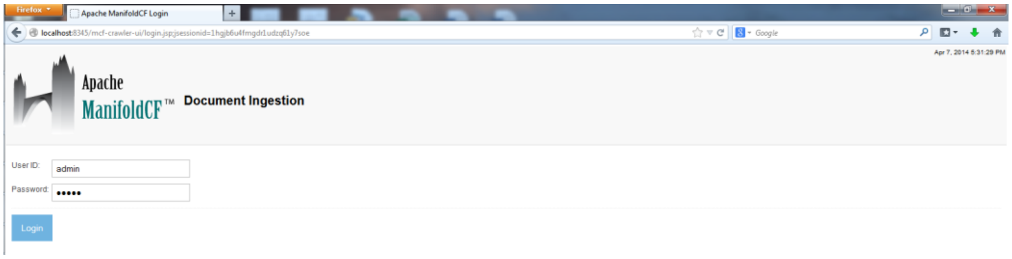
1.3.4 Verifying the build

Let's make sure everything seems to be working, by visiting the Crawler UI and setting up a simple crawl. You have the option of crawling the web, of course, but for the purposes of this exercise, let's pick the most basic possible crawl we can, so that very little can go wrong: the local file system.

In order to have something to crawl, you'll want to create a directory and some files. You can do that wherever in the file system you like, but you will obviously need to have write access to the place you choose. Then, you'll need a few files you can add to, modify, or delete. This is how you might create these on Windows:

```
cd <the place you want to put the crawl area>
mkdir testcrawlarea
cd testcrawlarea
mkdir subdir1
mkdir subdir2
echo "this is the time" >this_time.txt
echo "for all good men" >for_all_good_men.txt
echo "to come to the aid of their country" >2_come_2_their_country.txt
echo "four score" >subdir1\four_score.txt
echo "and ten" >subdir1\and_ten.txt
echo "years ago" >subdir2\years_ago.txt
echo "IBM formed a more perfect computer" >subdir2\more_perfect.txt
```

Now that we have a sample crawl area, point your browser at `http://localhost:8345/mcf-crawler-ui`. You should see the screen in figure 1.6.

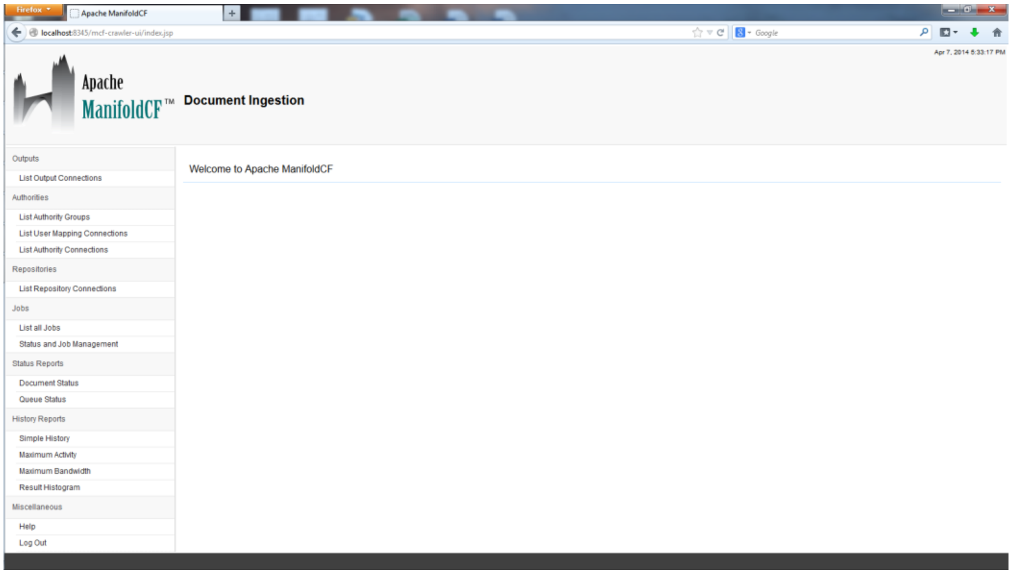

Firefox
Apache ManifoldCF Login
localhost:8345/mcf-crawler-ui/login.jsp;jsessionid=1hgjb6u4fmgdr1udzq61y7soe
Apache ManifoldCF™
Document Ingestion
Login

User ID: admin

Password: •••••

Figure 1.6: The login page of the crawler UI. The default user name and password is 'admin' and 'admin'.

After you log in, your browser will be redirected to the main navigation screen, which appears in figure 1.7:



Welcome to Apache ManifoldCF

Figure 1.7: The ManifoldCF welcome screen. It seems very friendly.

Notice the navigation menu on the left-hand side of the browser window. It's broken into roughly three sections: a section for defining each kind of connection (at the top), a section for defining and running jobs (the middle section), and a section for generating reports (the bottom section). We'll go into all of these sections in much greater depth in subsequent chapters. For now, though, let's create some connection definitions and define a simple job.

1.4 Creating a simple crawl

Now that ManifoldCF is built and seems to work, it is time to figure out how to use it. We'll start with the basics: creating a simple crawl with the Crawler UI. Initially, we will be crawling just your file system, and sending the documents nowhere. We will start by creating the necessary connection definitions. But before we can proceed, we must understand the difference between a connector and a connection definition.

1.4.1 Understanding connectors and connection definitions

If you look carefully at the Crawler UI welcome page, you will notice that the word "connector" doesn't seem to appear on it anywhere. Since you know now that ManifoldCF is all about connectors, how can that be?

It makes perfect sense when you realize that the Crawler UI is designed for the end user, not for a programmer, while a connector is just a piece of code. It is the specific piece of software that knows how to communicate with a specific type of repository, authority, or search engine. Elsewhere within ManifoldCF one may do things with connectors that one might expect to do with other pieces of software – for example, register or unregister them. As we have already learned, this registration process is done automatically by the quick-start example when it reads the `connectors.xml` file upon startup.

Definition A *connector* is a java class which provides the logic necessary to interact with a specific type of repository, authority, or search engine.

For interacting with a repository, ManifoldCF introduces the concept of a *repository connector*. Similarly, for interacting with an authority, ManifoldCF has a notion of an *authority connector*. Finally, ManifoldCF abstracts from the interaction with a search engine by using an *output connector*.

Unlike connectors, the Crawler UI seems to mention connections quite a lot. In fact, almost a third of it is dedicated to managing them. So, what is a connection, exactly?

The word "connection" in the Crawler UI is shorthand for *connection definition*. A connection definition describes how to set up a connector class instance, to use it to communicate with the desired external system. Formally:

Definition A *connection definition* is the configuration information necessary for a specific connector to function in a specific environment.

As an example, suppose that a connector needs to talk with a server. The connector would likely require configuration information describing which server to talk with. A connection definition based on that connector would thus include the class name of the connector along with the name of the specific server that it should communicate with.

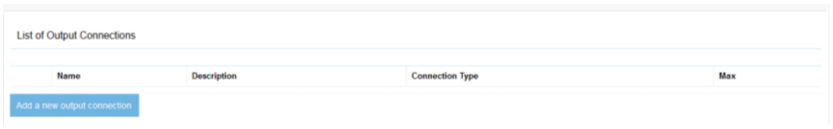
Since there are three different kinds of connectors within ManifoldCF, it follows that there will be three different kinds of connection definitions also. You can manage your output connection definitions, your repository connection definitions, and your authority connection definitions all from within the Crawler UI.

1.4.2 Creating connection definitions

Let's go ahead and use the Crawler UI to set up a `null` output connection definition and a `file system` repository connection definition. The null output connector is very simple and exists wholly for demonstrations such as these: It accepts documents that are handed to it, and just throws them away. It is useful primarily as a way to make it easier to develop repository connectors. It also happens to make it easier to do simple demonstrations without the added difficulty of setting up a search engine instance. The file system repository connector is also highly convenient for setting up a simple example. It is completely self-contained and can be used even if no connection to the internet is available.

Note Neither the file system connector nor the null output connector requires any connector-specific configuration information. Thus, if you want to learn a lot about configuring connection definitions, this example is not very interesting. You will likely be more satisfied with the example presented in Chapter 2.

Figure 1.8 shows the pertinent portion of the screen that you should see when you click on the `List Output Connections` link in the navigation area.


List of Output Connections

Name	Description	Connection Type	Max

Figure 1.8: The list of current output connections.

To add a new output connection definition, click the `Add a new Output Connection` button. You can then see the connection definition and its tabs, as in figure 1.9.

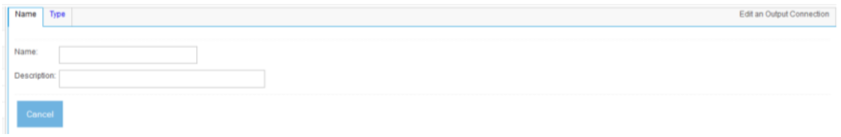
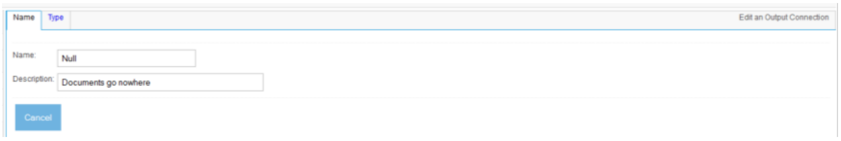


Figure 1.9: Creating a new output connection.

All the editing screens in the ManifoldCF crawler UI are tabbed in this way. Since configuration information (and, later, specification information) can be quite extensive, the tabs help connector writers organize information so that an end user is not subjected to scrollbar whiplash. Tabs also provide a natural boundary between editing pages coming from different components of ManifoldCF. For example, the tabs you see here (`Name` and `Type`) both come from the framework itself. There are no connector-specific tabs yet, because we haven't selected the connector yet, but when the connector is selected, tabs will appear that are furnished by the specific chosen connector.

Let's fill in the `Name` tab and give your new output connection definition a name and description. The name you provide must be unique across all output connection definitions. The description, while optional, helps describe the connection definition to remind you later exactly what the connection definition represents. See figure 1.10.

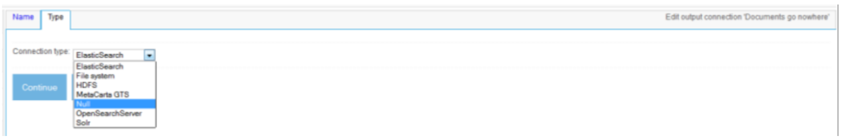


Name: Null

Description: Documents go nowhere

Figure 1.10: Defining an output connection definition; filling in the Null output connection definition name and description.

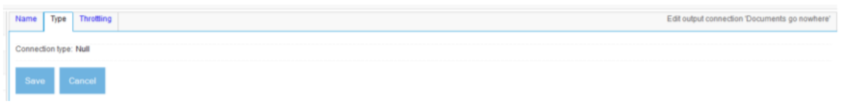
Next, click the `Type` tab. This will permit us to choose the type of the connector to use for the connection definition. Choose the null output connector, since we do not have an actual search engine set up as of yet. See figure 1.11.



Connection type: ElasticSearch

Figure 1.11: Selecting a connector for the output connection definition.

Once we’ve selected a type, we can click the `Continue` button, and the connector-specific tabs will appear. See figure 1.12.



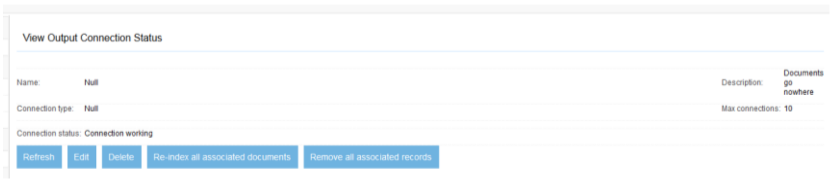
Edit output connection 'Documents go nowhere'

Connection type: Null

Figure 1.12: Output connection definition after connection type selection. The Throttling tab appears.

Another tab has now appeared, and also that the `Continue` button has changed to a `Save` button. This means that since you have now given the connection definition a name and selected a connector for it, you can now save it, if you desire. Don’t worry that you may not have specified everything that is required; if you haven’t, the crawler UI will certainly let you know, by popping up a Javascript warning box and not submitting the page.

The tab that appeared (`Throttling`) is the only one that is appropriate for the null output connector. Throttling is a complex issue, which we will cover in more detail in later chapters, so let’s just go ahead and click the `Save` button, and see what our connection definition looks like. See figure 1.13.


Refresh
Edit
Delete
Re-index all associated documents
Remove all associated records

View Output Connection Status

Name: Null
Description: Documents go nowhere
Connection type: Null
Max connections: 10
Connection status: Connection working

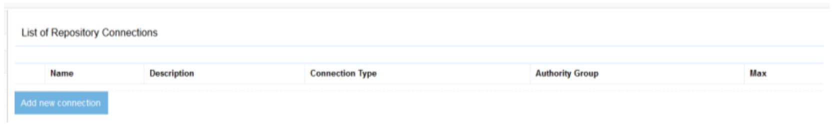
Figure 1.13: A view of the saved null output connection definition. The ‘Connection working’ message is no surprise.

This screen has a connection status field, which says `Connection working`. This status is presented when you view any connection definition, and is designed to allow you to tell, at a glance, whether your configuration is correct or not, and if it’s not working, what the problem is. This check is a connector-specific kind of activity so, as you might expect, there is a connector method for performing a status check.

Second, you may note that there is a set of links across the bottom of the display area. These links allow you to perform various actions related to the connection definition: delete it, refresh the view (so you can get an updated status), edit it again, and lastly, there’s a link labeled `Re-ingest all associated documents`. What exactly does that mean?

If you will recall, ManifoldCF is an incremental crawler. In order to be incremental, it needs to keep track of what documents it has handed to each index. However, it is always possible that some external agent might modify a search engine’s index unbeknownst to ManifoldCF. This button basically tells ManifoldCF to forget everything it thinks it knows about the target search engine. Clicking it forces ManifoldCF to assume nothing as far as the state of documents on the other side of the output connection definition.

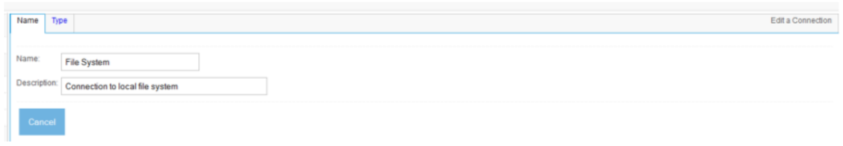
Next, let’s go on to create the file system connection definition that we will crawl with. The process within the UI is nearly identical. First, click on the `List Repository Connections` link in the navigation area. See figure 1.14.


List of Repository Connections

	Name	Description	Connection Type	Authority Group	Max

Figure 1.14: A list of the current repository connection definitions.

Next, just as we did before, click the `Add new connection` link, and give your new connection definition a name and description. See figure 1.15.

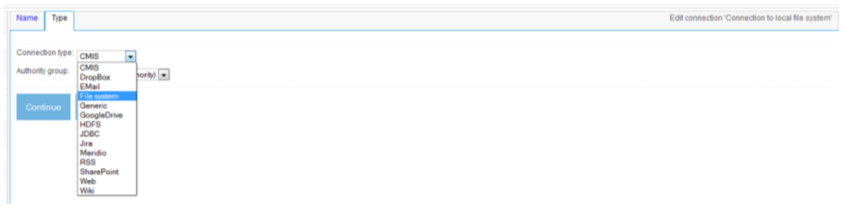


Name: File System

Description: Connection to local file system

Figure 1.15: Creating a new repository connection definition, and giving it a name and description.

Once again, click the `Type` tab to select a type. See figure 1.16.



Connection type: CMIS

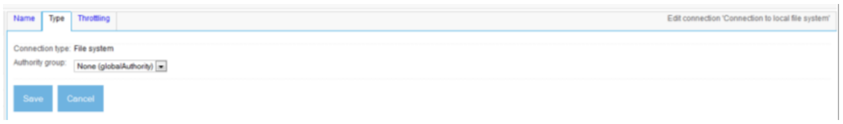
Authority group: ...ority)

Figure 1.16: Defining a repository connection definition, selecting the file system connector.

Oh, but wait a minute – this screen seems to have something about selecting an authority group. What should we fill in here?

As we have seen, an authority is an arbitrator of security – specifically, of the ability of a given user to see documents. Authority connection definitions can be created in the crawler UI just like repository connection definitions and output connection definitions. Authority groups represent collections of these authority definitions. A repository connection definition may refer to a specific authority group as the provider of security for its documents. That's what the Crawler UI is asking you to provide here. But since these documents are not really going anywhere, we don't need to worry about their security, so it is perfectly fine to just leave the default option in place.

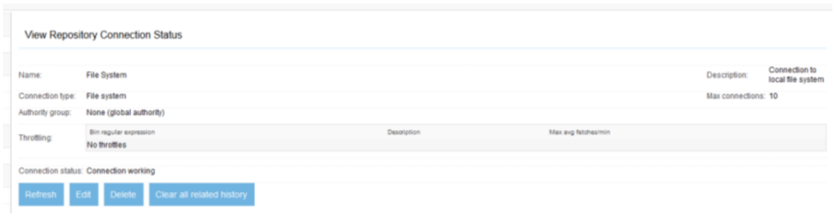
Now, click the `Continue` button to populate the rest of the tabs for the connection definition, based on the connection type we chose. See figure 1.17.



Connection type: File system
Authority group: None (globalAuthority)

Figure 1.17: Defining a repository connection definition, after the repository connection type was selected.

Once again, the `Save` button appears, and so does a `Throttling` tab, which we can ignore for now. Click the `Save` button to save the connection definition. The result is displayed in figure 1.18.



Name: File System
Description: Connection to local file system
Connection type: File system
Max connections: 10
Authority group: None (global authority)
Throttling:

Bin regular expression	Description	Max avg fetches/min
No throttles		

Connection status: Connection working

Figure 1.18: View of the saved file system connection definition. The 'Connection working' status means that the connection definition has been successfully verified, which doesn't mean much for the file system connector.

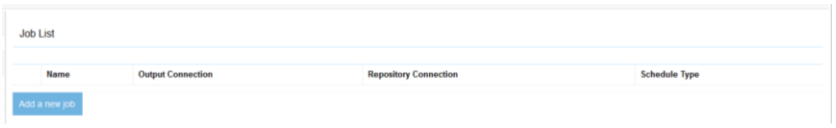
Congratulations, you have successfully created your first connection definitions! Next, we must create a job that will use these connection definitions to perform a simple crawl.

1.4.3 Defining a job

Now that you have successfully defined a repository connection definition and an output connection definition, you can create a job definition which uses these connection definitions. Many people often make the mistake when first dealing with ManifoldCF of presuming that a ManifoldCF job is just a task – something that is run once and then forgotten. But since the goal is synchronization, ManifoldCF in fact has a rather different philosophy: each job is meant to be run multiple times. You can think of it this way: a ManifoldCF job defines *what* and *when*, rather than *how*. Thus, when you define a job, you will spend your time specifying sets of documents and metadata, and setting schedules.

> **Definition** A *job definition* describes a set of documents, specifying the source of those documents, where they should be sent, and when this activity should take place.

Creating a job definition is easy, and similar in many respects to creating connection definitions. In the crawler-UI navigation area, click the `List all jobs` link. You will see a screen similar to figure 1.19.

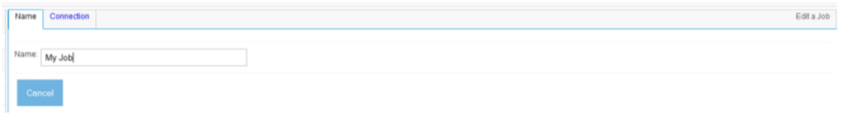


Job List

Name	Output Connection	Repository Connection	Schedule Type

Figure 1.19: Displaying a list of all currently-existing job definitions.

Click the `Add new job` link at the bottom, and you will see your new job definition. The newly-created job definition has tabs much like connection definitions. See figure 1.20.

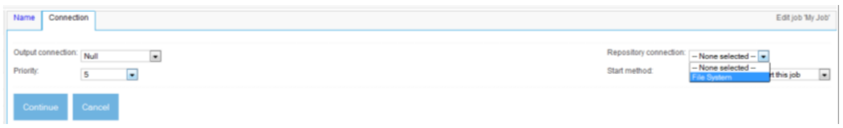


Name: My Job

Figure 1.20: Screenshot of the Name tab for a new job.

Here you should pick a name for your job definition. Job definition names are not required to be unique, so you will want to be careful to pick a name that is sufficiently descriptive so that you will not get your jobs confused with one another. Real people have wasted a lot of time trying to figure out why their documents weren't getting indexed, all because they were running the wrong job all along. Since such errors can be highly embarrassing, I recommend that you invest the few seconds necessary to keep the meaning of each of your jobs clear.

When you are done selecting a name, click the `Connection` tab. See figure 1.21.

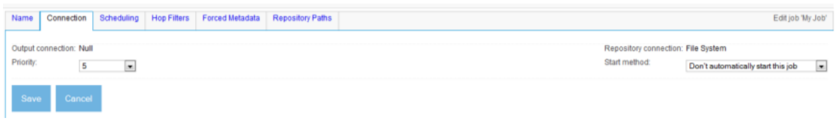


Output connection: Null
Priority: 5
Repository connection: -- None selected --
-- None selected --
File System
Start method: t this job

Figure 1.21: Selecting connection definitions on the job definition connection tab.

Here you must choose what connection definitions to use for the output connection and the repository connection fields. For this example, you will need to select the null output connection definition and file system repository connection definition you defined earlier.

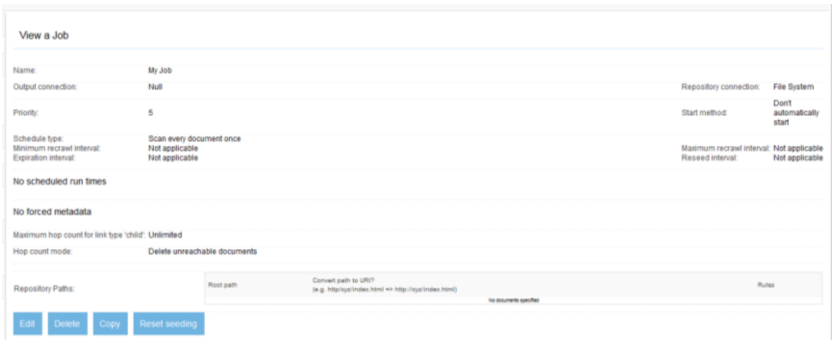
Notice that on this tab you can also select the job's priority, which controls how the crawler prioritizes documents from this job versus any others that are running, and also whether the job must be started manually, or starts by itself based on a schedule. We will go into these options in more detail in Chapter 2. For now, just click the `Continue` button. By now it is not a surprise that additional tabs appear. See figure 1.22.



Output connection: Null
Priority: 5
Repository connection: File System
Start method: Don't automatically start this job

Figure 1.22: Selecting a job's connection definitions, so as to use the file system repository connection definition and the null output connection definition we created earlier.

The new tabs all have their uses, which we will explore further in Chapter 2. Click the `Save` button. The saved job definition is shown, as it should appear, in figure 1.23.



View a Job

Name:	My Job		
Output connection:	Null	Repository connection:	File System
Priority:	5	Start method:	Don't automatically start
Schedule type:	Scan every document once		
Minimum recrawl interval:	Not applicable	Maximum recrawl interval:	Not applicable
Expiration interval:	Not applicable	Reseed interval:	Not applicable

No scheduled run times

No forced metadata

Maximum hop count for link type 'child':	Unlimited
Hop count mode:	Delete unreachable documents

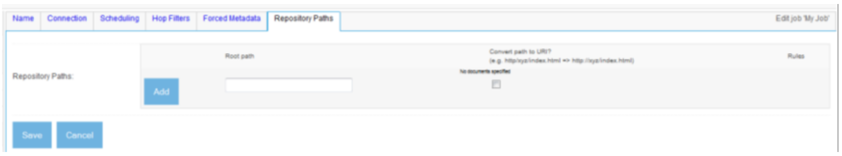
Repository Paths:

Root path	Convert path to URI? (e.g. http/xyz/index.html => http://xyz/index.html)	Rules
No documents specified		

Figure 1.23: The view of the saved job definition. Instead of a folder or two to crawl, 'No documents specified' is displayed.

But what is that line that says, "No documents specified"? Perhaps we have missed something. Specifically, we've neglected to specify what documents to include with the job. Such a job will not be very interesting to run; it will simply stop as soon as it starts, with no other activity.

To fix this, let's edit the job definition again. Click the `Edit` link. Your job definition once again appears in tabbed format, for editing. This time, click the `Paths` tab, as shown in figure 1.24.



Repository Paths:

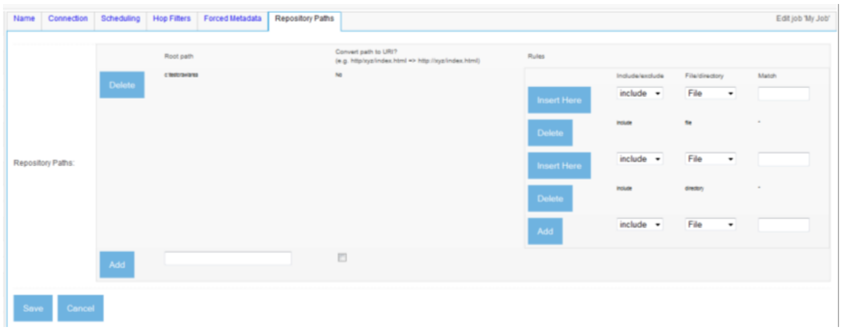
	Root path	Convert path to URI? (e.g. http/xyz/index.html => http://xyz/index.html)	Rules
		No documents specified	
Add		☐	

Figure 1.24: The File System Connector Paths tab, before any path has been entered.

This is where the file system connector allows you to select the files you want to include in a job definition's document set. Here you can add a root path, and rules for deciding what documents under that root path should be included.

> **Note** ManifoldCF defines documents quite broadly. Not all documents will be indexed. It is up to an individual connector to decide what a document is, and how it will be used. For example, the file system connector treats both files and directories as documents, but only extracts document references from directories, and only ever indexes files.

Type in the path to the documents you created earlier, and click the `Add` button. Now, the screen looks like figure 1.25.



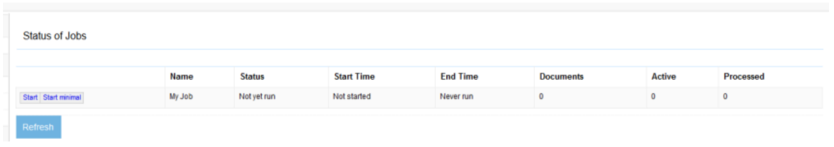
	Root path	Convert path to URI? (e.g. http/xyz/index.html => http://xyz/index.html)	Rules			
				Include/exclude	File/directory	Match
Delete	c:\testcrawlarea	No	Insert Here	include	File	
			Delete	include	file	*
			Insert Here	include	File	
			Delete	include	directory	*
			Add	include	File	
Add		☐				

Figure 1.25: Adding a path to a file system job definition. The default rules include all files and all subdirectories.

You will notice that you automatically get two rules for free: a rule to include all files matching "*", and a rule to include all subdirectories matching "*". The File System Connector evaluates rules in order, so you are also given a chance to insert new rules before each existing rule, and to append a new rule at the end.

That's good enough for our first crawl! Click `Save` to save the job definition.

Now that the job definition is created, there's nothing left to do but to start the job and see what it does. In the navigation area, click the `Status and Job Management` link. You will see a screen similar to figure 1.26.


Status of Jobs

	Name	Status	Start Time	End Time	Documents	Active	Processed
Start Start minimal	My Job	Not yet run	Not started	Never run	0	0	0

Figure 1.26: Job status page as it appears before the job has been run the first time.

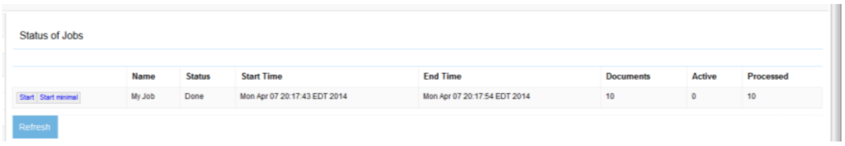
If you click the `Start` link, the job will start, and you will see the `Status` column update to `Starting up`, which indicates that the job is starting.

The status page does not automatically update. In order to see the progress of the job, you thus need to periodically click the `Refresh` link. You will see that the counts displayed under the `Documents`, `Active`, and `Processed` columns change as the job runs. These columns give you some idea of what's happening, if you know how to interpret them. Table 1.1 describes them.

Table 1.1: Job status columns and their meanings

Column	Meaning
`Name`	The name of the job
`Status`	The current status of the job, e.g. `Not yet run`, `Starting up`, `Running`, `Aborting`, `Terminating`, `End notification`, or `Done`
`Start Time`	The local time the job was last started, if any
`End Time`	The local time the job last completed, if any, unless the job is currently running or was aborted
`Documents`	The total number of documents that exist in all states that are part of the job
`Active`	The total number of documents that currently need to be processed before the job completes
`Processed`	The total number of documents that are part of the job that have been processed at some point in the past, either in this run of the job, or in a previous run

You will find much more on this topic in Chapter 2.
Figure 1.27 shows what the job status page will look like when your job is done.


Status of Jobs

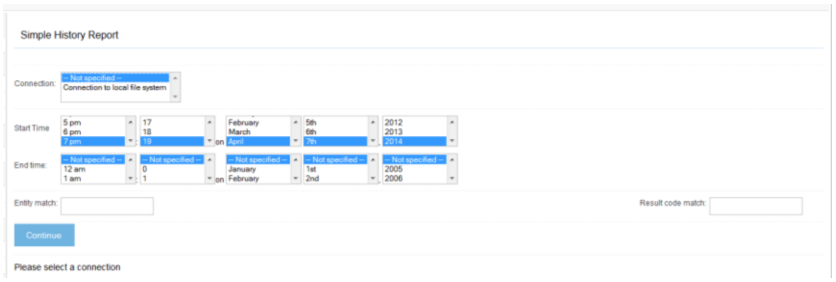
	Name	Status	Start Time	End Time	Documents	Active	Processed
Start Start minimal	My Job	Done	Mon Apr 07 20:17:43 EDT 2014	Mon Apr 07 20:17:54 EDT 2014	10	0	10

Figure 1.27: Job status page at the end of the crawl. Your job is done!

Congratulations! It looks like you have successfully run your first ManifoldCF job! But, what happened? How can you know that something useful took place?

1.4.4 Viewing the history of a connection definition

The easiest way to see what happened is by viewing the repository connection definition's activity history. ManifoldCF keeps a record of every activity that has taken place since you created your repository connection definition, so all we need to do is to look at it. The kinds of activities that get recorded depend in part on the kind of connectors involved, although there are a number of activities that the framework itself tracks. So let's see what got recorded. In the navigation area, click the `Simple History` link. See figure 1.28.



Simple History Report

Connection: -- Not specified -- / Connection to local file system

Start Time: 5 pm / 6 pm / 7 pm : 17 / 18 / 19 on February / March / April 5th / 6th / 7th , 2012 / 2013 / 2014

End time: -- Not specified -- / 12 am / 1 am : -- Not specified -- / 0 / 1 on -- Not specified -- / January / February -- Not specified -- / 1st / 2nd , -- Not specified -- / 2005 / 2006

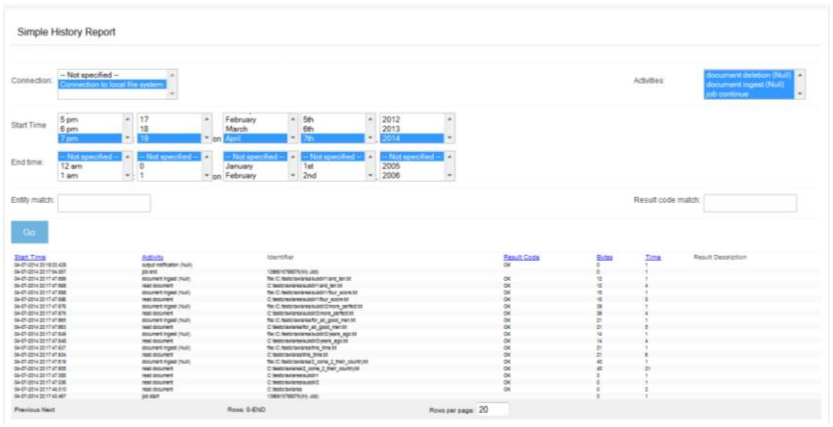
Entity match: Result code match:

Continue

Please select a connection

Figure 1.28: The Simple History report, before the connection definition is chosen.

Here you must choose the name of the connection definition whose history you want to browse. You may also select a time interval that you are interested in (default is the last hour), and limit the records you want to see by their recorded identifier. But since we want to see everything, just select the name of your connection definition, and click the `Continue` button. Figure 1.29 shows the results.



Simple History Report

Connection: -- Not specified -- / Connection to local file system

Activities: document deletion (Null) / document ingest (Null) / job continue

Start Time: 5 pm / 6 pm / 7 pm : 17 / 18 / 19 on February / March / April 5th / 6th / 7th , 2012 / 2013 / 2014

End time: -- Not specified -- / 12 am / 1 am : -- Not specified -- / 0 / 1 on -- Not specified -- / January / February -- Not specified -- / 1st / 2nd , -- Not specified -- / 2005 / 2006

Entity match: Result code match:

Go

Start Time	Activity	Identifier	Result Code	Bytes	Time	Result Description
04-07-2014 20:18:03.428	output notification (null)		OK	0	1	
04-07-2014 20:17:54.587	job end	[illegible](My Job)		0	1	
04-07-2014 20:17:47.699	document ingest (null)	file:C:\testcrawlarea\subdir1\and_ten.txt	OK	12	1	
04-07-2014 20:17:47.698	read document	C:\testcrawlarea\subdir1\and_ten.txt	OK	12	4	
04-07-2014 20:17:47.688	document ingest (null)	file:C:\testcrawlarea\subdir1\four_score.txt	OK	15	1	
04-07-2014 20:17:47.686	read document	C:\testcrawlarea\subdir1\four_score.txt	OK	15	8	
04-07-2014 20:17:47.676	document ingest (null)	file:C:\testcrawlarea\subdir2\more_perfect.txt	OK	39	1	
04-07-2014 20:17:47.675	read document	C:\testcrawlarea\subdir2\more_perfect.txt	OK	39	4	
04-07-2014 20:17:47.665	document ingest (null)	file:C:\testcrawlarea\for_all_good_men.txt	OK	21	1	
04-07-2014 20:17:47.663	read document	C:\testcrawlarea\for_all_good_men.txt	OK	21	5	
04-07-2014 20:17:47.649	document ingest (null)	file:C:\testcrawlarea\subdir2\years_ago.txt	OK	14	1	
04-07-2014 20:17:47.648	read document	C:\testcrawlarea\subdir2\years_ago.txt	OK	14	4	
04-07-2014 20:17:47.637	document ingest (null)	file:C:\testcrawlarea\this_time.txt	OK	21	1	
04-07-2014 20:17:47.634	read document	C:\testcrawlarea\this_time.txt	OK	21	6	
04-07-2014 20:17:47.619	document ingest (null)	file:C:\testcrawlarea\2_come_2_their_country.txt	OK	40	1	
04-07-2014 20:17:47.605	read document	C:\testcrawlarea\2_come_2_their_country.txt	OK	40	21	
04-07-2014 20:17:47.588	read document	C:\testcrawlarea\subdir1	OK	0	1	
04-07-2014 20:17:47.536	read document	C:\testcrawlarea\subdir2	OK	0	1	
04-07-2014 20:17:46.510	read document	C:\testcrawlarea	OK	0	2	
04-07-2014 20:17:43.487	job start	[illegible](My Job)		0	1	

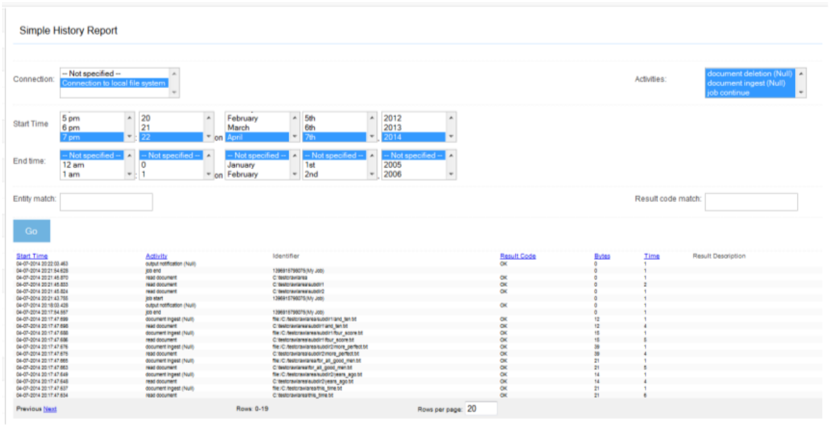
Previous Next Rows: 0-END Rows per page: 20

Figure 1.29: A Simple History report for the file system connection definition. All possible activities are selected by default. The initial time range is selected to be the last hour.

The upper right corner of the display area has a list of activities you can select. These activities are specific, in part, to the underlying connector you've chosen. Initially, all activities are selected, but you can change that if you need to in order to see only particular ones. You can also change the time limits, the entity match, and the result code match regular expressions as well. If you change anything and want to redisplay the report, just click the `Go` button.

From the data, which is initially in reverse chronological order, it is clear that the crawl took place, and indeed handed the expected documents off to the null output connector, which of course did nothing with them. This is great news!

Let's explore what will happen if we run the job again. Since ManifoldCF is an incremental crawler, and since there have been no changes to anything in the tree, we should see no additional "document ingest" activities in that case. Let's try it: go back to the `Job Status and Management` page, and click the `Start` button again. Click the `Refresh` link until the job is done, and then again enter the Simple History report to view what happened. Figure 1.30 shows the results.



Simple History Report

Connection: Connection to local file system

Activities: document deletion (Null), document ingest (Null), job continue

Start Time: 7 pm : 22 on April 7th, 2014

End time: -- Not specified -- : -- Not specified -- on -- Not specified -- -- Not specified --, -- Not specified --

Entity match:

Result code match:

Start Time	Activity	Identifier	Result Code	Bytes	Time	Result Description
04-07-2014 20:22:03.463	output notification (Null)		OK	0	1	
04-07-2014 20:21:54.628	job end	1396915798075(My Job)		0	1	
04-07-2014 20:21:45.870	read document	C:\testcrawlarea	OK	0	1	
04-07-2014 20:21:45.833	read document	C:\testcrawlarea\subdir1	OK	0	2	
04-07-2014 20:21:45.824	read document	C:\testcrawlarea\subdir2	OK	0	1	
04-07-2014 20:21:43.755	job start	1396915798075(My Job)		0	1	
04-07-2014 20:18:03.428	output notification (Null)		OK	0	1	
04-07-2014 20:17:54.557	job end	1396915798075(My Job)		0	1	
04-07-2014 20:17:47.699	document ingest (Null)	file:/C:/testcrawlarea/subdir1/and_ten.txt	OK	12	1	
04-07-2014 20:17:47.698	read document	C:\testcrawlarea\subdir1\and_ten.txt	OK	12	4	
04-07-2014 20:17:47.688	document ingest (Null)	file:/C:/testcrawlarea/subdir1/four_score.txt	OK	15	1	
04-07-2014 20:17:47.686	read document	C:\testcrawlarea\subdir1\four_score.txt	OK	15	5	
04-07-2014 20:17:47.676	document ingest (Null)	file:/C:/testcrawlarea/subdir2/more_perfect.txt	OK	39	1	
04-07-2014 20:17:47.675	read document	C:\testcrawlarea\subdir2\more_perfect.txt	OK	39	4	
04-07-2014 20:17:47.665	document ingest (Null)	file:/C:/testcrawlarea/for_all_good_men.txt	OK	21	1	
04-07-2014 20:17:47.663	read document	C:\testcrawlarea\for_all_good_men.txt	OK	21	5	
04-07-2014 20:17:47.649	document ingest (Null)	file:/C:/testcrawlarea/subdir2/years_ago.txt	OK	14	1	
04-07-2014 20:17:47.648	read document	C:\testcrawlarea\subdir2\years_ago.txt	OK	14	4	
04-07-2014 20:17:47.637	document ingest (Null)	file:/C:/testcrawlarea/this_time.txt	OK	21	1	
04-07-2014 20:17:47.634	read document	C:\testcrawlarea\this_time.txt	OK	21	6	

Previous Next Rows: 0-19 Rows per page: 20

Figure 1.30: Simple history results from the second job run. No documents were indexed, and only three documents were read – all directories.

Clearly the ManifoldCF incremental magic worked. The job didn't do much of anything, because it somehow knew that it did not need to. But how did it do this?

Simply put, ManifoldCF keeps track of the version of every document it knows about. This version is represented internally as an unlimited-length string, and all ManifoldCF needs to do to ensure that the version is indeed correct is to check the current version string of the document against the previous version string in order to decide if the document will need to be indexed again. Single-document deletions are noted in a similar manner: if the document used to exist, but has disappeared, then the crawler can safely delete the document from the output index.

You may have noticed a flaw in this algorithm. The job definition describes documents in terms that depend on the kind of connector, and some connectors might well want to specify a set of documents by means of what is reachable starting at a given point. Large numbers of documents may therefore get orphaned – they still exist, but they do not fulfill the job's document criteria anymore. (The Web connector, among others, has this characteristic). Or, the user might change the job definition between crawls in a way that eliminates some of the documents previously indexed. How does ManifoldCF handle these situations?

What ManifoldCF does on every job run is to segregate the documents that it has seen so far in that run from those that it hasn't seen yet. When the job is done, those documents that were never encountered at all are deleted from the output index. The phase of the job that deletes unreachable documents (see section 1.1.3) is described by a job status of `Terminating` in the crawler UI. When it finishes, the job run is complete.

Now that you have a working job set up, please feel free to modify your data and your job definition, and experiment. Add documents and directories, and recrawl. Delete documents and recrawl. Change the job definition's document criteria, and recrawl. What happened? Is it what you would have expected?

Since you are beginning to understand what a job actually is, can you guess what might take place when you delete one? Because documents in the index are associated with jobs, you would be correct to guess that deleting a job will cause the documents managed by the job to be deleted from the output index. The only exception to this rule is when, by chance, the same document is present in more than one job. This is a rare case, and we'll have a chance to discuss it later.

Now that you are getting somewhat familiar with ManifoldCF, you might be thinking, "well, that's a nice toy, but what can it really do?" In particular, you might wonder at what kinds of limits ManifoldCF has that might make it less useful than it first appears. We'll discuss those limits in the next section.

1.5 ManifoldCF scalability

The next time you catch up with your friend Bert, he's not very happy. He's been told that he needs to crawl 500,000,000 documents every day, which he helpfully notes is more than 5700 documents per second. He is not certain that ManifoldCF can deliver on that kind of performance. This is not doing his ulcer any good.

After you think about it, you realize that there really are two distinct issues Bert is dealing with: performance, and capacity. We'll address each of these in turn.

1.5.1 Performance

First, recognize that Bert has a point. No software can possibly perform as well as he's told his must. But this applies equally to the content repository he's trying to crawl. You tell Bert about the old story of the bear, and the people running away from the bear, and note that just like you only need to run faster than the slowest person in order to escape the bear, a crawler only needs to be faster than the content repositories it is connected to in order to be as successful as possible.

This is not necessarily the case for open web crawling with ManifoldCF. In that scenario, ManifoldCF (or its underlying database) may indeed become a performance bottleneck, if a completely un-throttled crawl is done. But, except in very specific cases (such as web archiving), real-world web crawls are almost always constrained in very specific ways:

- They are host or domain limited
- Throttling is done per host or per domain

So, unless there are a **lot** of domains or hosts, once again the constraints of the repository largely determine the overall performance of the system.

In the rare case that ManifoldCF itself winds up being the bottleneck, it is good to remember that ManifoldCF permits you to run multiple agents processes, which are all coordinated through Apache ZooKeeper. The real limit, therefore, is the underlying database. Depending on which database you use, this also can be made to scale in a similar manner.

It’s not hard at all to run a simple performance test with ManifoldCF, just to get some idea of where things stand. The file system connector is not ideal for something like this, because the crawl is limited by the speed of the disk being crawled. Nevertheless, using ManifoldCF with the file system connector and PostgreSQL, my cheap tower system running Windows Vista readily achieves some forty documents per second throughput. A fairly typical server with fast disks has achieved more than eighty documents per second.

More on how to tune the performance of ManifoldCF can be found at http://manifoldcf.apache.org/releases/trunk/en_US/performance-tuning.html.

1.5.2 Capacity

The capacity question is a real concern as well. In this case, the limiting factor tends to be the back-end search engine into which you will be sending your documents. Search engines have performance concerns too; in order to handle a large capacity with acceptable latency, search engines very often break down the search space into chunks, and search these chunks in parallel. Breaking the crawling task down in a way that corresponds to the index chunks will make the crawling numbers much more manageable, and allow the crawling work to be distributed across many machines.

It is also worth noting here that many search-type systems tend to be significantly overspecified. In my experience, document counts as stated by a client are often inflated by one, and sometimes two, orders of magnitude. Many times an enterprise really doesn't know how many documents they are dealing with until they actually try to crawl them, and so they deal with that lack of knowledge by coming up with estimates that are ridiculously conservative. So it is always worthwhile questioning these sorts of assumptions.

ManifoldCF capacity also depends critically on how well the underlying database performs. Other factors, such as the specific connector chosen, usage of features such as hop count limitation, and whether a crawl rescans documents or merely expires them also can greatly affect perceived capacity. But, technically, ManifoldCF's document capacity is a determined by how well the underlying database can manage the document queue. It usually boils down in the end to how well a single query works – the query that obtains the next bunch of documents for processing – and also how well the database works in a very parallel environment. Some databases are unable to perform the critical query without loading the entire document queue into memory and sorting it. Some databases are effectively single-threaded and cannot get the most out of the multiprocessor nature of modern hardware. Such databases will not achieve real-world scalability goals.

This is one reason why I recommend PostgreSQL as the ManifoldCF database of choice as of this writing. It has the best parallelism I've seen, and can essentially scale infinitely as far as ManifoldCF is concerned. Using PostgreSQL 8.3.7, and selecting all the correct options, I have successfully performed ManifoldCF web connector crawls totaling some five million documents in a single job. PostgreSQL, currently on version 9.1, has continued to improve since then.

In the more distant future, ManifoldCF may well include a database implementation which is based on an underlying distributed key-value store, such as Voldemort. This would allow ManifoldCF itself to be distributed across multiple machines. Experiments we have done so far, however, are not encouraging; the kinds of constraints that are needed for ManifoldCF to function properly do not seem to map well to a key/value kind of solution.

1.6 Summary

In this chapter, we've begun to learn the basics of ManifoldCF. We now know how to build it and run it, and we understand what its major components are. We understand what fundamental kinds of crawling there are, and what *connector*, *connection definition*, and *job* really mean in the context of ManifoldCF. We've also looked at how to think about sizing and scalability concerns where ManifoldCF is concerned.

The next chapter will build upon this foundation, and examine in depth one of the principle building-blocks: the crawler UI.