10 Organization and Architecture

This chapter covers

- The internal organization and components of all ManifoldCF processes
- What each of these processes is intended to do
- The components that make up these processes, and what they are for

The purpose of this chapter is to prepare you to be able to effectively extend ManifoldCF, through the process of writing connectors. Even though a connector is straightforward to write, you do not yet know enough about how ManifoldCF is put together to be able to write one effectively. Thus, here we will describe ManifoldCF's internal architecture in considerable detail, with the goal in mind of providing that background knowledge that you will need.

10.1 Process architecture

We begin with the big picture. In many ways, an architectural view of ManifoldCF is like blind men feeling an elephant – there are many ways to approach it, and many pictures we could draw. Each has its benefits and its drawbacks. But we will have to start somewhere, and a process-oriented view is not an unreasonable way to begin.

You may recall that in Chapter 1 we spent some short time describing the various processes that comprise ManifoldCF. Let's revisit that, with the idea of going into all of these processes in detail.

10.1.1 Overall organization

Figure 10.1 is a pictorial representation of the processes that comprise ManifoldCF.

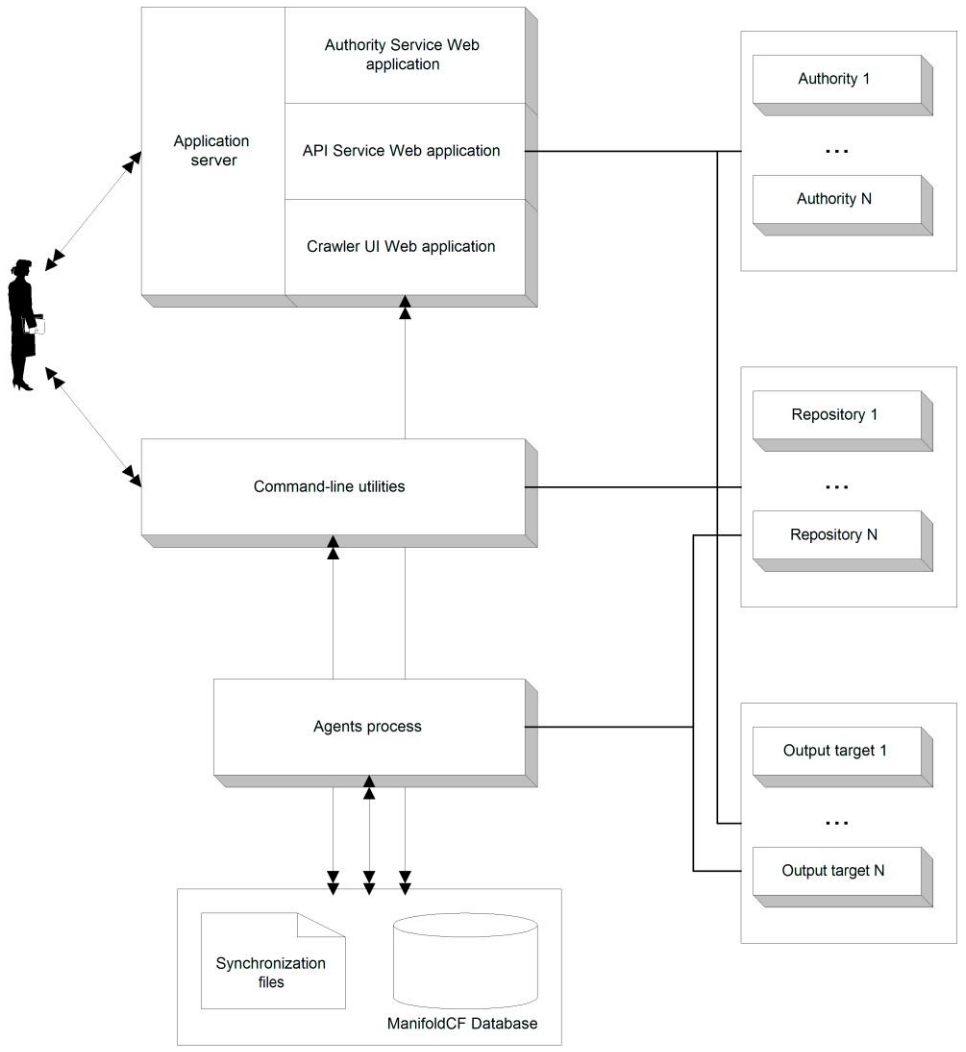

Figure 10.1 The processes that make up ManifoldCF, and their interrelationships with repositories, authorities, and output targets. This picture represents the “multiprocess” method of deploying ManifoldCF. The Quick Start method runs all components in a single process, which avoids the need for synchronization files, and also the need for command-line utilities.

In this diagram, you see on the left hand side the individual processes that comprise ManifoldCF, and you see processes on the right side which correspond to hypothetical processes for a set of authorities, repositories, and output targets. A human being (or another computer system) will typically interact with the ManifoldCF web services, and with the command-line utilities. The Agents process, on the other hand, is autonomous, and requires no interaction. It is sometimes called the *agents daemon* on Unix systems, for this reason.

Let's visit all the boxes on the left side of figure 10.1 in turn, and describe the functions of each in more detail.

10.1.2 Agents process

We're going to start with the part of ManifoldCF that does all the real work. This is the autonomous Agents process.

This process theoretically consists of multiple independent *agents*., along with support systems that these agents can use to perform incremental indexing of documents. An agent is an autonomous body of code that is responsible for synchronizing data between one or more repositories and one or more outputs. But as of this writing, there is only one agent supplied with ManifoldCF – a "pull" agent, or an implementation of the "pull" model, which if you remember our discussion in Chapter 1, is another name for a crawler.

Other agents are nevertheless still allowed. You would theoretically create an agent every time you needed to support a repository by means of a push model. But as I have already pointed out in Chapter 1, repositories that can actually support a useful push model are practically unknown. When ManifoldCF was designed, this was not clear, so the design permits such agents to be included, should the need arise.

See figure 10.2 for a graphical depiction of the internal components of the Agents process.

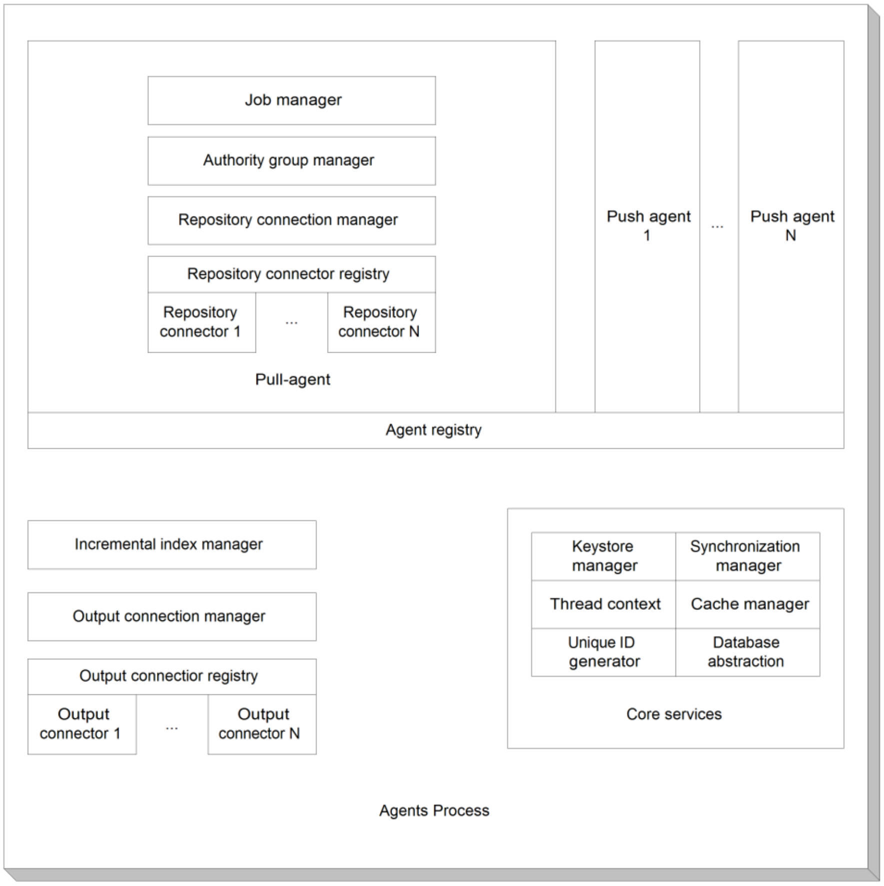

Figure 10.2 Internal components of the Agents process. This is divided into multiple agents, one of which is the pull agent we've been using up until now, plus an incremental index manager, and output connection definition manager.

Examining this diagram closely, you can see that the components that are in common for all agents appear in the bottom half. If you needed to write an agent of your own, you would need to learn these components intimately. But this book doesn't propose to teach you how to do that, because, frankly, I have only limited experience with writing such agents. I have only written the one, and until I have occasion to develop others, it is not clear what should be stressed and what the key techniques are. A later edition, perhaps, might tread here.

CORE SERVICES

On the other hand, the part of the diagram labeled "core services" represents functionality you will need to learn if you are to write any connector code whatsoever. These core services are present in every ManifoldCF process, except the connector-specific processes. We're going to actually need to learn how to use these services in some detail, but we will be doing that in a later chapter. Table 10.1 describes each service and what it does.

Table 10.1 Core services, and their functions

Service	**What it does**
Keystore Manager	This component manages custom trust stores, where certificates involved in secure socket protocols are kept. This includes serialization of these trust stores for storage within the database. The component also includes a custom secure socket factory which permits secure socket connections to be negotiated using a specified local trust store, instead of Java's global trust store.
Thread Context	The thread context component provides thread-local name/value storage, for many purposes. It is often used to cache handles of various manager classes so they do not need to be re-created constantly.
Unique ID Generator	This component generates system-wide unique identifiers, using the services of the synchronization manger component.
Synchronization Manager	The synchronization manager component handles inter-process and inter-thread synchronization and shared data.
Cache Manager	The cache manager component builds on the synchronization manager component to provide robust caching of any kind of objects that are expensive to generate. These objects may expire after a given time, or upon specific invalidation events, or because there are too many cached objects already of a given kind.
Database abstraction	The database abstraction component provides standardized methods of performing database-specific activities. There is also built-in support for transaction-aware database resultset caching, built upon the cache manager component.

INCREMENTAL INDEX MANAGER

This component is the underpinning of incremental indexing. It is responsible for keeping track of what has been sent to each output target. For each document that has been indexed, this manager stores its identifier, its URL, and also something called the *version string*. The version string is an unlimited-length opaque Unicode string which uniquely describes the version of the document being indexed. For example, for the file system connector, the version string may well contain the file's last-modified timestamp. Other repository connectors use completely different version string formats.

The purpose of the version string is to give the Agents process a means of deciding whether a document has changed or not. Thus, the only fundamental requirement on its form is that when the source document changes, it must also change. But, as we will see, it is also a convenient way to cache information about the document that is expensive to obtain more than once.

OUTPUT CONNECTION MANAGER

The Output Connection Manager component keeps track of the individual output connection definitions as they are defined by the user. As we have already discussed, an output connection definition is described by a connector class name, and connection configuration information. This component stores and retrieves both kinds of information.

OUTPUT CONNECTOR REGISTRY

The Agents process keeps track of the various output connectors available to the system by means of a registry. This registry is needed so that the Agents process knows what classes it should consider to be valid output connector implementations. But the Output Connector Registry component does little more than store an output connector's class name, along with a description.

AGENT REGISTRY

The Agent Registry component keeps track of all agents that the Agents process is aware of. It keeps track of the name of the class implementing each agent, along with a human-readable description. The Agents process uses this information in order to start up and tear down the threads that comprise each agent in an orderly fashion, and also to insure data integrity when output connection definitions are being deleted or output connectors are being deregistered.

The Pull Agent component is the only agent currently part of ManifoldCF. Its purpose is to crawl all of those repositories best suited to a pull model. This turns out to be all of the repositories ManifoldCF supports at this time. We'll examine its internal structure in the next section.

10.1.3 Pull Agent

The purpose of the Pull Agent component is to crawl repository documents and index them incrementally. Like all agents, it can be started and stopped, which (if you think about it) means that it has threads of its own. Indeed, it has many; the thread architecture of the Pull Agent will be described in detail in Chapter 12.

These threads are supported by numerous components that manage stored information, which abstract from the database tables where this information is actually kept. You can see these components pictorially described in figure 10.2. There are components for managing authority connection definitions, repository connection definitions, and job definitions.

Job Manager

The Job Manager component is the most important Pull Agent component. It keeps track of two kinds of information about jobs.

The first kind of information it stores is information about each job's definition. This includes the job's name, configuration, repository connection definition, and output connection definition. It also includes something else: information about the job that is required by the job's repository connection definition. This is called a job's *document specification*, and its precise form is determined by the underlying repository connector.

> **Definition** A *document specification* is a body of job configuration information that mainly describes what documents are considered part of a job. It may also contain information describing should be included with, or otherwise affect, indexed documents belonging to the job. Its actual form is repository connector specific.

An example of a bit of information often found in a document specification is the path to documents that should be included in the job. Another example might be special metadata names and values that are intended to be indexed with every document that is part of the job.

There may also be similar information that is required by the job's output connection definition. For example, some output connectors permit documents to be segregated into *collections*. Jobs that use output connection definitions based on such connectors need to have a way of specifying the collection for the job's documents. We call this kind of information, which also contributes to how a document gets indexed, an *output specification*.

> **Definition** An *output specification* is a body of job configuration information which should be included with, or otherwise affect, the indexed documents for the job. Its form is specific to the kind of output connector being used.

The data managed for both of these specifications is treated in an opaque manner by the job manager. By this, I mean that the Job Manager component doesn't pay any attention to the details of what is in the document specification or output specification, or how it is laid out. But it does know how to store and retrieve this job definition information from the underlying database tables.

The second kind of information that is managed by the Job Manager component is information about the job's operational status. This dynamic information includes what the state of the job is, what documents are part of the job, what the hop count is from a seed to document, and basically all the information needed to run jobs. These data structures, and the algorithms that use them, are described in detail in Chapters 6 and 7.

Repository Connection Manager

The purpose of the Repository Connection Manager component is similar in many ways to the Output Connection Manager component described for the Agents process, earlier. It keeps track of connection definition names, repository connector class names, and connector-specific configuration information. But unlike the Output Connection Manager, the Repository Connection Manager is also responsible for keeping track of history information, for the purpose of building history reports. This information is stored so that each history record is associated with a specific repository connection definition.

The decision to attach history information to repository connection definitions is not an obvious one. Why attach history records there, and not to output connection definitions instead? The reason is partly historical: in ManifoldCF, the concept of multiple repository connection definitions preceded the implementation of multiple output connection definitions by many years. But also, this choice seems to better reflect the way people think about crawls. It's not surprising that people who need to use ManifoldCF view their problem as primarily about crawling. Attaching history information to repository connection definitions rather than output connection definitions serves to support that prejudice.

Authority Group Manager

I've drawn the Authority Group Manager as part of the Pull Agent, but I must emphasize that its presence here is not significant. It does not play a major role in crawling. It is used in the pull agent solely to provide qualifiers for access tokens that are attached to documents as they are indexed.

If you recall the discussion in Chapter 4 about how authorities work, and what they do, you will remember that ManifoldCF segregates tokens from different authorities in order to prevent cross-contamination. The method used to achieve this segregation is to attach the authority group name to the front of every access token. For example, an access token `Fred` will become `MyAuthorityGroup:Fred` when the token is handed to the index. The sole function of the Authority Connection Manager in the context of the Pull Agent is to add those qualifiers to the appropriate tokens.

Rest assured, though, that the Authority Group Manager component is used more fully by other ManifoldCF processes. We'll see this later in the chapter.

Repository Connector Registry

The Repository Connector Registry component keeps track of repository connector class names and their descriptions, very much like the Output Connector Registry component keeps track of output connector class names in the Agents process, and for similar reasons. It stores a set of repository connector class names and descriptions, and does little else.

We've now described all of the components of the pull agent. Next, let's examine the internals of the Crawler UI web application.

10.1.4 Crawler UI web application

In figure 10.1, the Crawler UI web application is one of the ways in which a user can directly interact with ManifoldCF. Being a web application, it requires a Java application server to run, which we are going to presume that the reader is already somewhat familiar with. In particular, the Crawler UI is built using standard Java Server Pages, or JSPs for short.

The Crawler UI web application's internal structure has a significant amount in common with the agents process. See figure 10.3.

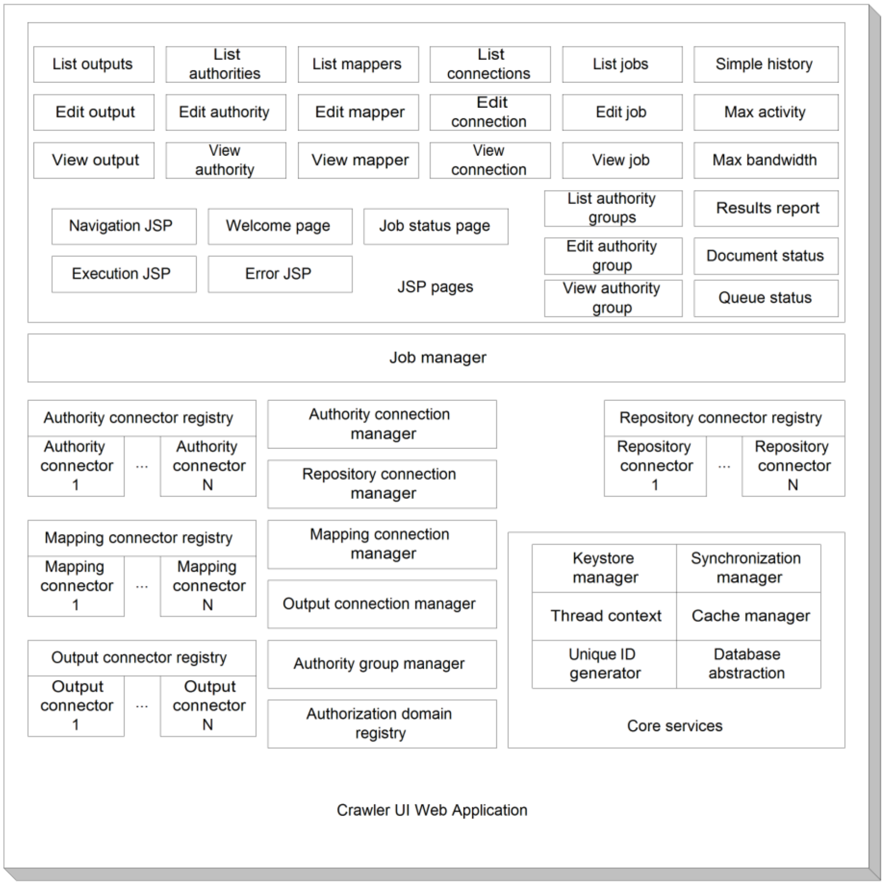

Figure 10.3 The internal components of the Crawler UI web application. This web application shares many components with the Agents process, but also includes significant functionality in the form of JSP pages.

The components we've not seen before include those having to do with authority connections, mapping connections, domain registration, and specific JSP pages. We'll cover each of these in turn.

Authority Connection Manager

The Authority Connection Manager component keeps track of individual authority connection definitions as they are defined by the user, in addition to the prerequisite relationships between authority connections and any mapping connections that may be specified. As we have already discussed, an authority connection definition is described by a connector class name, and connection configuration information. This component stores and retrieves both kinds of information.

Authority Connector Registry

The Authority Connector Registry keeps track of the class name and description of each authority connector available for use within ManifoldCF. It's implementation and capabilities are very similar to those of the Output Connector Registry and Repository Connector Registry, described above.

Mapping Connection Manager

The Mapping Connection Manager component tracks individual mapping connection definitions in a manner similar to how the Authority Connection Manager does this, including keeping track of the required prerequisite mapping connections needed for a mapping connection. This manager tracks data which consists of a connector class, connection configuration information, and the prerequisite mapping connection name, if any.

Mapping Connector Registry

Since mapping connectors are a variant of the basic connector paradigm, it makes sense that they would be registered in a manner similar to output, repository, and authority connectors. The Mapping Connector Registry does that job, in a manner similar to the registries for the other kinds of connectors in the ManifoldCF system.

Domain Registry

The names of all the authorization domains known for a ManifoldCF instance are kept track of in a registry of its own, called the Domain Registry. This tracks only a name and a description for each authorization domain.

JSP Pages

One unique component of the Crawler UI web application is the set of JSP pages that actually form the user interface. This is not unexpected; the UI needs access to many of the same data structures as the Pull Agent, and we will not revisit these. The JSP pages, on the other hand, deserve some consideration. You may well recognize many of these from the UI walkthrough we did in Chapter 2. Others we have not yet talked about, because they function more as building blocks than as pages in their own right. For completeness, I've summarized them all in table 10.2.

Table 10.2 Crawler UI web application JSP pages and their functions

JSP page	Function

Navigation JSP	This JSP provides the navigation portion of all pages.
Execution JSP	The execution JSP is the universal target of all of the UI's JSP POST operations. All POSTs use this page as a target. This page then includes the real target page when it is done processing form information.
Error JSP	If an error is detected, another page may redirect to this one to present it to the user.
List, edit, and view output pages	These pages correspond to their obvious output connection definition UI equivalents. The edit output page presents all non-connector-specific tabs.
List, edit, and view authority group pages	These JSP pages correspond to the crawler UI's ability to list, edit, and view authority groups.
List, edit, and view authority pages	These pages correspond to their UI equivalents. The edit authority page presents all non-connector-specific tabs.
List, edit, and view mapper pages	These pages correspond to the crawler UI's ability to list, edit, and view mapping connections. The edit mapping connection page presents all non-connector-specific tabs.
List, edit, and view connection pages	These pages map to their repository connection definition UI equivalents in the expected way. The edit connection page presents all non-connector-specific tabs.
List, edit, and view job pages	These pages list, allow editing of, and permit viewing of job definitions. The edit job page presents all tabs that do not come from the repository or output connectors.
Job status page	This page presents the job status UI page.
Simple history, max activity, max bandwidth, and result report pages	These pages present their associated history reports in the UI.
Document status and queue status pages	These pages present their associated status reports in the UI.

With that, we're done with the components of the Crawler UI web application. Let's consider the API Service web application next.

10.1.5 API Service web application

The second way a user can directly interact with ManifoldCF is via the API Service. As shown in figure 10.1, this service is also structured as a web application. This time, however, it does not use JSPs, since a REST-style API is not well suited to that technology. Instead, the ManifoldCF API Service is constructed using a Java servlet. As you might expect, the API Service also requires the support of a Java application server. See figure 10.4.

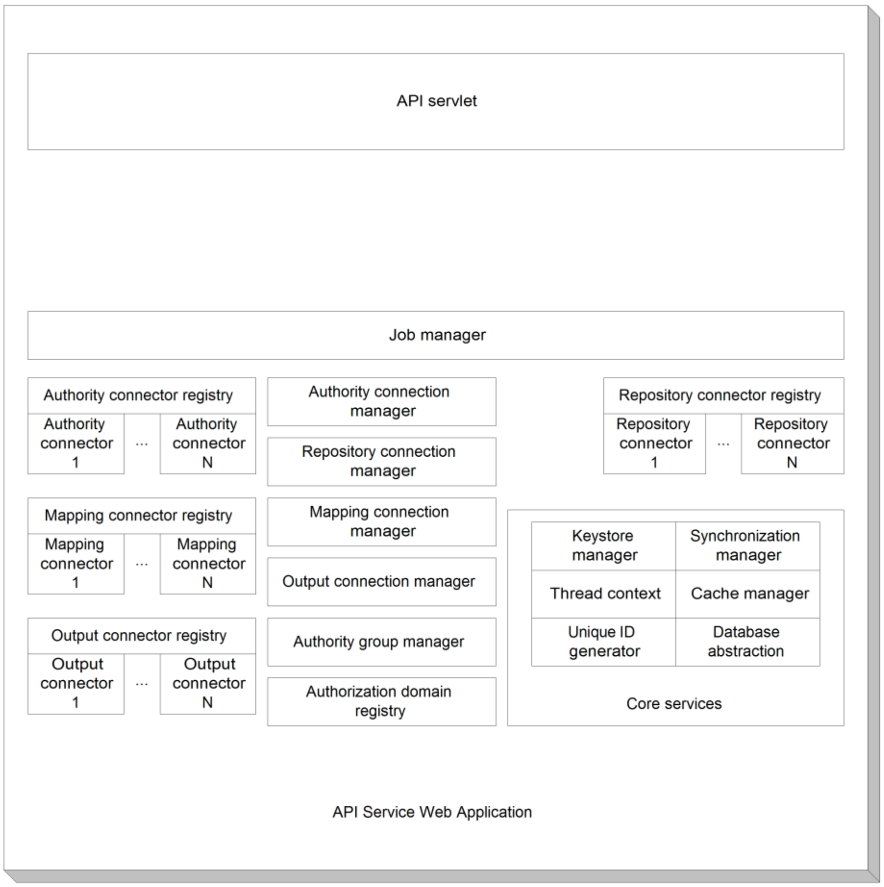

Figure 10.4 The internal structure of the API Service web application. The only unique component is the API servlet itself; all other components are found elsewhere throughout ManifoldCF.

The API servlet box on the diagram above represents the single class that implements the servlet. I will presume that the reader already has some familiarity with the Java servlet API, and will not digress to explain it in any detail. Suffice it to say that all four REST verbs have corresponding methods in the API servlet class. These methods decode the JSON inputs and encode the JSON outputs, while in the middle actually use the other components depicted to perform the desired actions.

The remainder of the diagram is identical to figure 10.3. This should be expected because the API Service web application is equivalent in most functional respects to the Crawler UI web application. The differences that do exist (which, as of this writing, are confined to lack of support in the API for reports) do not show up as missing components.

That's a pretty complete description of the components of the API Service web application. Next, we'll look into the Authority Service web application and see what we find there.

10.1.6 Authority Service web application

The Authority Service web application is the third way by which users can directly interact with ManifoldCF. The protocol is also not well suited to a JSP paradigm, so the web application is implemented as a servlet, and requires the support of a Java application server. See figure 10.10.

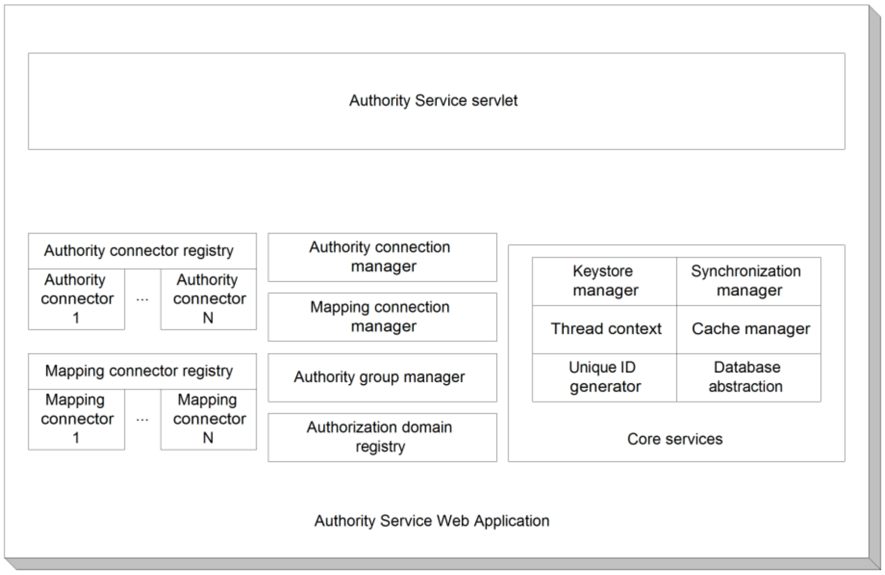

Figure 10.5 The Authority Service web application. This web application makes use of far fewer common modules than any other ManifoldCF component, because it is only necessary for it to use the authority connector infrastructure.

The Authority Service servlet is the only component that is unique to the Authority Service web application. This servlet accepts GET requests and converts them into requests to individual authority connectors, and then amalgamates the results to construct a response. There is, however, quite a bit more to this servlet than that. For one thing, it has its own thread pool, because it uses separate threads to allow it to communicate with all authorities simultaneously on each request. The proper management of this thread pool requires that the servlet's initialization and teardown methods be properly implemented so that threads are not left dangling. We will cover this in more depth in Chapter 12.

The other boxes in the diagram should by now be becoming familiar, since they are present in all of the other ManifoldCF processes. Perhaps the only notable difference is that the Authority Connection manager and connectors are used fully only by this web application.

We're done with our explanation of the components of the Authority Service web application. There is only one figure 10.1 process remaining to describe: the command-line utilities. We'll look at those next.

10.1.7 Command-line utilities

There are many individual command-line utilities available in ManifoldCF. Their major purpose, in the aggregate, is to provide access to required ManifoldCF functions which are not handled in any other way. For example, initializing the database requires the use of a command-line utility, as does registering (or deregistering) a connector.

None of this applies to the Quick Start example, which is what we've been mainly using for examples in this book. In the case of the Quick Start, it turned out to be possible to incorporate many of the functions of the command-line utilities into the example startup class. This has the advantage of simplicity. The cost, on the other hand, is borne in terms of loss of flexibility, and in terms of extra cost at startup time.

> **Note** The Quick Start example is configured to run ManifoldCF as one process, and **only** one process. It is therefore impossible to use the Quick Start together with the command-line utilities. Attempting to do so will likely result in many kinds of errors, both explicit and subtle in nature.

For the rest of this section, we are going to talk about the command-line utilities as they are used in a multi-process ManifoldCF deployment.

There are several different types of ManifoldCF command-line utilities. The types denote the level of functionality within ManifoldCF that the command makes use of. For example, the core-level commands expose core service component functionality exclusively, while the agents-level commands expose agent process functionality to the outside world. Each command is a class with a static `main()` method, which can thus be invoked directly by firing up a new Java Virtual Machine instance. The commands have the convention of getting any required input by way of command-line arguments, and presenting output using the standard output stream. Standard error is also used, but is used exclusively for status and error messages. A command's return value is always zero if the command was successful, or non-zero if an error occurred.

Table 10.3 lists some commonly-used command-line utilities.

Table 10.3 Common command-line utilities, plus their types and functions

Command	**Type**	**Function**
`org.apache.manifoldcf.core.DBCreate`	Core	Creates the ManifoldCF database instance
`org.apache.manifoldcf.core.DBDrop`	Core	Removes the ManifoldCF database instance
`org.apache.manifoldcf.agents.Install`	Agents	Creates the agent process's database tables
`org.apache.manifoldcf.agents.Uninstall`	Agents	Tears down the agent process's database tables
`org.apache.manifoldcf.agents.Register`	Agents	Registers an agent, also initializing all of the agent's database tables
`org.apache.manifoldcf.agents.UnRegister`	Agents	Unregisters an agent, tearing down any of the agent's database tables while it is at it
`org.apache.manifoldcf.agents.RegisterOutput`	Agents	Registers an output connector
`org.apache.manifoldcf.agents.UnRegisterOutput`	Agents	Unregisters an output connector
`org.apache.manifoldcf.agents.AgentRun`	Agents	Starts the agents process
`org.apache.manifoldcf.agents.AgentStop`	Agents	Causes the agents

		process to shut itself down
`org.apache.manifoldcf.crawler.Register`	Pull Agent	Registers a repository connector
`org.apache.manifoldcf.crawler.UnRegister`	Pull Agent	Unregisters a repository connector
`org.apache.manifoldcf.authorities.RegisterAuthority`	Authorities	Registers an authority connector
`org.apache.manifoldcf.authorities.UnRegisterAuthority`	Authorities	Unregisters an authority connector
`org.apache.manifoldcf.authorities.RegisterMapper`	Mappers	Registers a mapping connector
`org.apache.manifoldcf.authorities.UnRegisterMapper`	Mappers	Unregisters a mapping connector

There are many other command-line utilities than those listed here. You can find a complete list at http://manifoldcf.apache.org/release/trunk/en_US/how-to-build-and-deploy.html.

As you can see, there are four basic types of ManifoldCF command-line utility. (Individual connectors can also have command-line utilities, but these are rare, and they are typically used to provide a command-line-based way of performing API-like functions, so I am not going to cover those in this architecture discussion.) Each of the basic types of command-line utility requires its own architecture diagram. We'll look at all of them in turn.

CORE COMMANDS

The core commands rely only on the ManifoldCF core services. See figure 10.6.

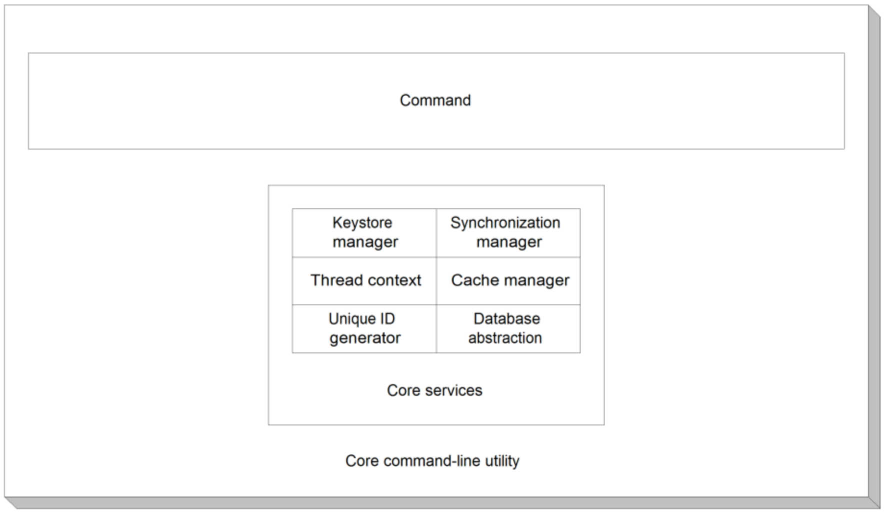

Figure 10.6 Internal structure of a core-type command-line utility. Only core components are used or exposed.

AGENTS COMMANDS

The agents commands make use of the components found in the agents process. See figure 10.7.

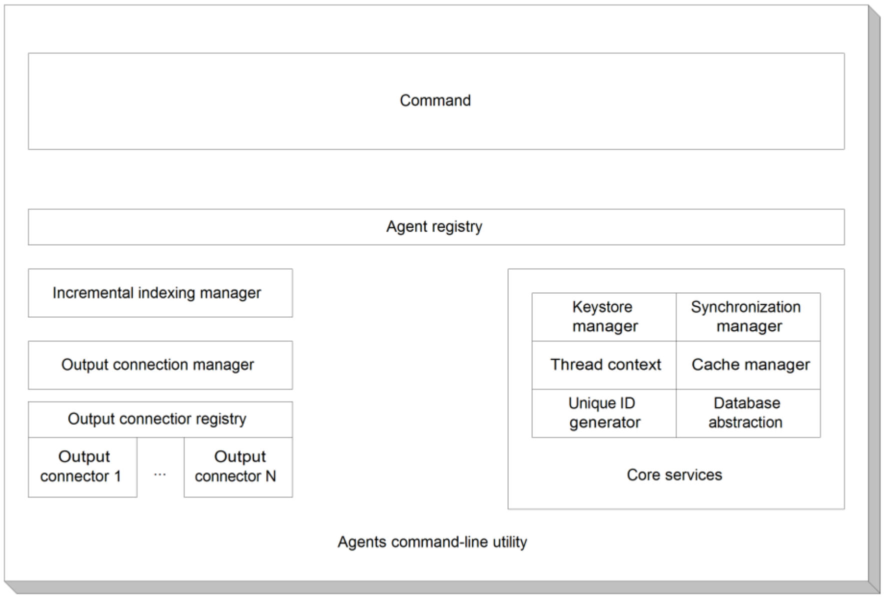

Figure 10.7 An agents command's internal components. These types of commands expose functionality for the agent registry, output connector registry, and output connection definitions. They have access to core services but do not serve to expose that functionality.

PULL AGENT COMMANDS

A Pull Agent command exposes functionality from within the components of the Pull Agent. See figure 10.8.

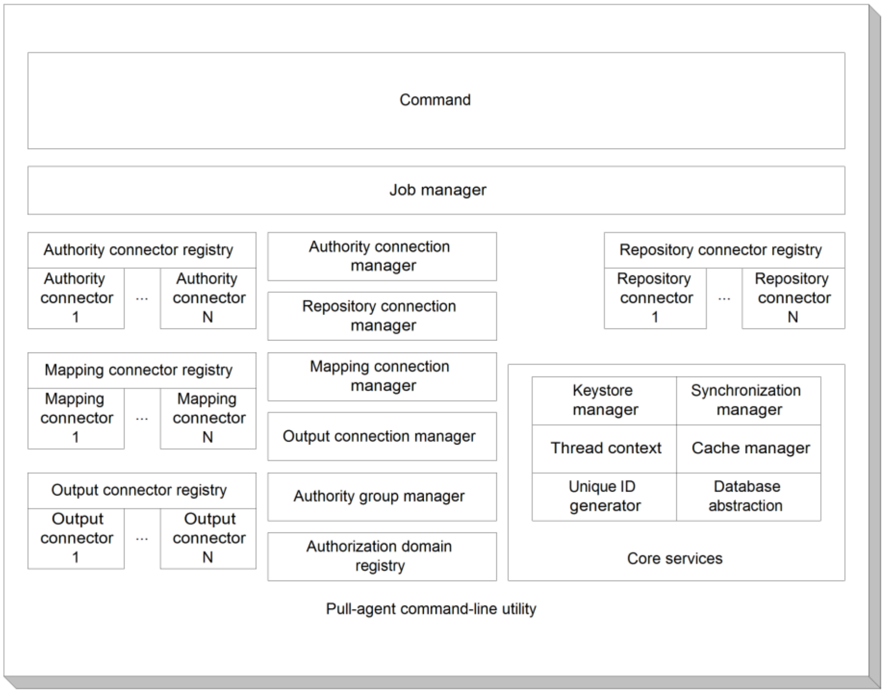

Figure 10.8 Components of a Pull Agent command-line utility. These are similar to the components of the Pull Agent itself.

AUTHORITY COMMANDS

The last type of command-line utility is the authority type. Components for this type are similar to those for the Authority Service. See figure 10.9.

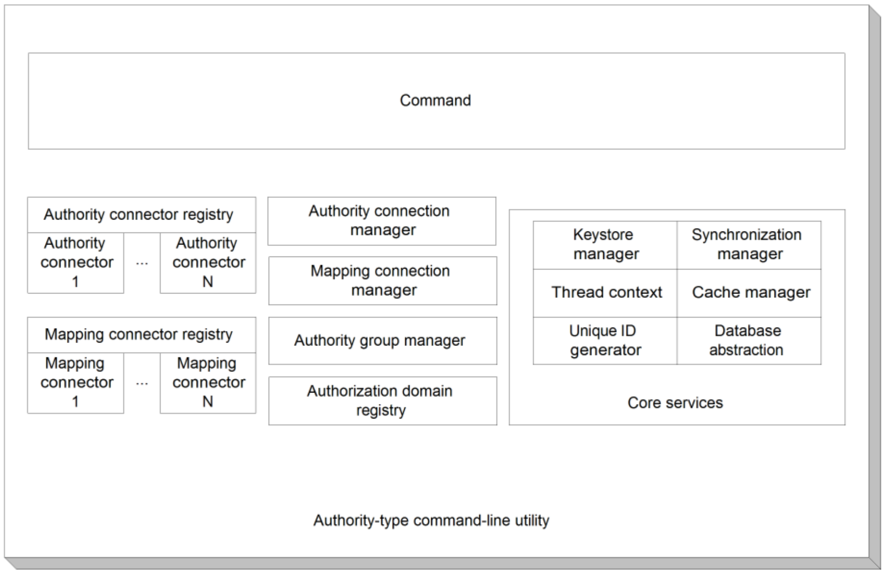

Figure 10.9 The components of an authority-type command-line utility. Other than the command class, these are similar to the Authority Service web application.

We're now done with presenting the components of command line utilities, and indeed we have finished describing all the components of all of ManifoldCF's various processes.

10.2 Summary

In this chapter we have explored the process architecture of ManifoldCF. In the next chapter, we'll build on this to learn about the data structures that underlie these processes and components. After that, we'll delve into the algorithms that tie all of ManifoldCF together.