6 A User's Look at the Cocoon Architecture

IN CHAPTER 4, "PUTTING COCOON TO WORK," you saw a simplified view of the Cocoon architecture. You built a first version of a news portal in Chapter 5, "Cocoon News Portal: Entry Version." Now that we have gone over the basics, it is time to fill in the missing pieces from a user perspective. This chapter presents additional Cocoon components and concepts you can use to build more advanced applications than the ones you have seen so far.

We will start by describing the architecture and further features of the sitemap in detail. A Cocoon-based application can become quite large. The sitemap becomes more complicated to manage as you add new pipelines. We will show you how to organize an application's structure so that it is easier to maintain. New components allow you to connect your Cocoon-based application to a database and diagnose what might be going wrong if something does not work as planned. We will also explain how Cocoon can be used without running it in a servlet engine and give some practical tips on how to tune an installation for maximum performance.

The Cocoon Architecture in Detail

Before we begin, let's look at a figure that gives an overview of the Cocoon architecture. It might help you to refer to Figure 6.1 when reading about the individual building blocks that make up Cocoon in the following sections. This figure is actually a simplified view of the architecture, because the dependencies of the components contained in Cocoon are more complicated than this figure shows. We will get into more detail as we progress through this book. Imagine that each chapter is a layer of Cocoon that you are slowly peeling away to see more and more of what is inside.

Cocoon is made up of several blocks of functionality. Starting at the top of Figure 6.1, you see Cocoon integrated into a servlet engine. This can be a standalone servlet engine, such as Apache Tomcat, or part of an application server, such as IBM WebSphere.

The Cocoon framework forms the envelope around the component-based architecture, including the different Cocoon components, such as generators and transformers, that can be used to build document pipelines, the XML and XSLT components, and any custom components built for a specific application.

As you can seen from the figure, each block in the Cocoon architecture has its own configuration file. Until now, we have only talked about the central Cocoon configuration file—the sitemap. The additional configuration files we will look at in this chapter are also important, because they allow you to define and configure various aspects of a Cocoon-based application, such as how a running Cocoon should react to changes in the sitemap or whether Cocoon should cache pipelines. In general, you will need to alter something in these configuration files only when development of the application is finished and you are ready to put it into a production environment.

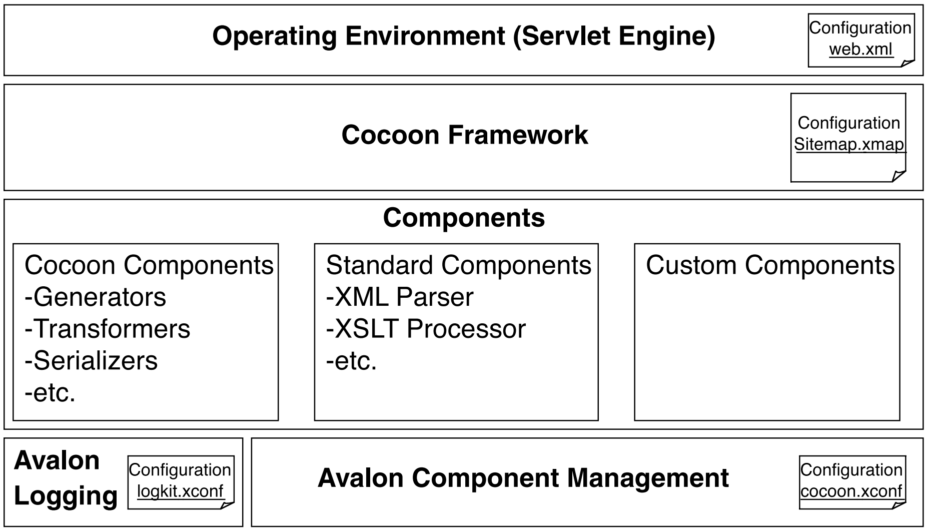

Operating Environment (Servlet Engine)
Configuration web.xml
Cocoon Framework
Configuration Sitemap.xmap
Components
Cocoon Components -Generators -Transformers -Serializers -etc.
Standard Components -XML Parser -XSLT Processor -etc.
Custom Components
Avalon Logging
Configuration logkit.xconf
Avalon Component Management
Configuration cocoon.xconf

Figure 6.1 The big picture of Cocoon.

Cocoon is a component-based system. As such, it uses parts of Avalon, a major Apache project for component-based Java architectures. Apart from Avalon component management, Cocoon also integrates the Avalon logging architecture, as shown at the bottom of Figure 6.1.

Avalon Integrated into Cocoon

In addition to including actual software components that can be used in an application, Avalon provides a set of rules and Java interfaces that are used in Cocoon to configure components. For example, Avalon allows components to be reused via a pooling mechanism. Therefore, Avalon provides components to manage these pools and also defines how a component should be written so that it can be pooled. Cocoon components then implement these interfaces.

The Avalon project is divided into several subprojects. However, not all the subprojects are used in Cocoon. The following is a list of subprojects that *are* used:

- **The Avalon LogKit.** A Java-based logging API. This logging functionality is used throughout all the Avalon-based projects and inside Cocoon. The logging configuration is very flexible, as you will see.
- **The Avalon Framework.** The base of Avalon. It defines several concepts and interfaces for component development in Java. It defines the basics of defining, configuring, and managing software components and how to use them.
- **The Avalon Excalibur project.** Layered on top of the Avalon Framework. It implements common reusable components and offers some component management facilities to fine-tune your installation.

This chapter looks at the possibilities Avalon provides in the context of how they are actually used inside Cocoon. For example, when we talk about logging, we give tips on how to optimize the performance of a Cocoon application. Also, for a more detailed overview of Avalon, see Chapter 8, "A Developer's Look at the Cocoon Architecture."

First, however, we'll start our configuration tour of Cocoon with the configuration file read by the servlet engine when Cocoon is started.

The Web Application Configuration

When Cocoon runs as a servlet, the servlet engine processes a configuration file during the startup phase. The servlet engine reads the web application deployment descriptor (which is located at WEB-INF/web.xml in your Cocoon context directory) and uses the parameters in this file to perform the initial configuration of Cocoon.

The web.xml file contains the startup configuration that is required to get the system running. The most important piece of information is the location of the configuration file for the Avalon-based Cocoon components. In Listing 6.1, which is a snippet from a web.xml file, the name and location of the configuration file are entered as parameters inside the `init-param` tag.

Listing 6.1 **The Avalon Configuration Location in web.xml**

```
<!--
  This parameter points to the main configuration file for Cocoon.
  Note that the path is specified in absolute notation but it will be
  resolved relative to the servlets webapp context path
-->
<init-param>
  <param-name>configurations</param-name>
  <param-value>/cocoon.xconf</param-value>
</init-param>
```

In a default installation of Cocoon, this file is called cocoon.xconf and is located in the Cocoon context directory. You have probably already seen it when looking for the sitemap, which is also located there by default. The cocoon.xconf file is an XML file that contains a description of the used Avalon components for Cocoon and their configuration. Configuring the name and location of this file inside web.xml allows you to choose your own name and location for the file if you wish. However, we recommend that you leave the defaults as is. From now on we will refer to this file simply as cocoon.xconf, regardless of where you place it and what name you choose.

Although the sitemap components, such as transformers and generators, are also Avalon-based components, they are *not* listed inside cocoon.xconf. They are listed inside the sitemap, as you saw in Chapter 4. This means that a site administrator building a Cocoon-based application does not need to know about cocoon.xconf. When designing an application, it is easier to reference only one file instead of having to view several files at once. cocoon.xconf will become important when you want to fine-tune the installation or replace any of the default components, such as the XML parser.

Configuring Components in cocoon.xconf

One of Cocoon's advantages is that it forms a flexible framework around other components that come from different projects, such as those hosted by Apache. For example, instead of being able to use only a specific XML parser, Cocoon allows you to choose which actual implementation you might want to use by allowing these components to be configured via cocoon.xconf. In addition, cocoon.xconf can be used to pass parameters to the components so that different aspects can be configured. Listing 6.2 is a brief excerpt from cocoon.xconf that shows the basics of this configuration.

Listing 6.2 **An Excerpt from cocoon.xconf**

```
<?xml version="1.0"?>
<cocoon version="2.0">

  <parser class="org.apache.cocoon.components.parser.XercesParser"/>
```

```
    <hsqldb-server class="org.apache.cocoon.components.hsqldb.ServerImpl"
                  pool-max="1" pool-min="1">
       <parameter name="port" value="9002"/>
       <parameter name="silent" value="true"/>
       <parameter name="trace" value="false"/>
    </hsqldb-server>
     ...
  </cocoon>
```

Unlike the sitemap, cocoon.xconf does not use a namespace. Each component you want to configure is defined inside the root element called `cocoon` using its own specific element. Listing 6.2 has two configured components: `parser` and `hsqldb-server`. These are the logical names under which Cocoon looks for a concrete implementation. The actual Java class that then implements the expected functionality is configured via the `class` attribute. As you can see from Listing 6.2, the default parser is the Xerces Parser from Apache. Apart from allowing different implementations to be used, cocoon.xconf allows the components to be configured using individual `parameter` tags. Each `parameter` tag consists of a `name` and `value` attribute. This lets you pass information such as the port number to the configured database. HSQLDB is an open-source database that is included in the Cocoon distribution. It is used in the practical database examples later in this chapter. We will also discuss the attributes `pool-max` and `pool-min` when we look at ways to optimize Cocoon's performance.

If you change something inside cocoon.xconf, these changes are not reflected automatically. To apply the changes, you have to reinstantiate Cocoon. One way of doing this is by restarting your servlet engine. However, this is not always an ideal solution, because you will affect other servlets also currently running in the same servlet engine. It might also take some time for the engine to restart.

Fortunately, Cocoon provides another way to force the reload of cocoon.xconf. You can directly request the root node where Cocoon is mounted (such as `http://localhost:8080/cocoon`) and then add the request parameter `cocoon-reload` with the value `true`. The whole URL looks like this:

```
http://localhost:8080/cocoon?cocoon-reload=true
```

This restarts Cocoon with the changed cocoon.xconf.

Because restarting can be a time-consuming process, you should avoid it in a production environment. You can turn off this feature by setting the parameter `allow-reload` in the web application deployment descriptor (web.xml) to `no`. The default for this setting is `yes`, as shown in Listing 6.3.

Listing 6.3 **Allowing Cocoon Reloading in web.xml**

```
<!--
      Allow reinstantiating (reloading) of the cocoon instance. If this is
      set to "yes" or "true", a new cocoon instance can be created using
      the request parameter "cocoon-reload"
.-->
```

continues

Listing 6.3 **Continued**

```
<init-param>
    <param-name>allow-reload</param-name>
    <param-value>yes</param-value>
</init-param>
```

Remember, this parameter is not in cocoon.xconf. It is in the web.xml file used to control certain settings for a servlet. This parameter should be set to `no` in a production environment, because the default allows anyone to start the reloading of your Cocoon installation by accessing the URL just listed. If someone were to abuse this, Cocoon would spend all its time reloading the configuration files, which would prevent any other activity.

In addition to component configuration, another important piece of information contained in cocoon.xconf is the location of the sitemap. The last line of cocoon.xconf looks like this:

```
<sitemap file="sitemap.xmap" reload-method="asynchron" check-reload="yes"/>
```

This definition tells Cocoon where to look for the main sitemap and how to handle its reloading. Although you can change the `file` attribute by entering a different location and name, we have never needed to change this setting. So we recommend that you do not change it either.

Sitemap Reloading

As you might have noticed during your first steps with Cocoon, changes made to the sitemap are automatically reflected after some time without a restart of your servlet engine being necessary.

When configured appropriately, Cocoon occasionally checks the sitemap for changes. Each time a change is detected, the old sitemap is discarded and the new one is used. Cocoon detects this change using the last modification date, which is automatically set by the operating system for a file when it is saved. So even if you do not change the sitemap but save it unchanged, Cocoon assumes that it *has* changed and reloads it.

As explained in Chapter 4, a servlet can act only on an incoming request. So Cocoon can check for changes only when a request for a document is received. The automatic reloading can be done in a synchronous or asynchronous manner. You can set this `reload` method by specifying either `synchron` or `asynchron` in the attribute `reload-method` in cocoon.xconf for the sitemap location. The default is `asynchron`. (Note that this is the correct way to write these parameters—without `ous` on the end.)

In synchronous mode, the new sitemap is generated in memory from the configuration file. After this process is finished, it is used and the request is served with this new sitemap.

In asynchronous mode, the new sitemap is generated in the background, and the incoming request is served by the old sitemap. All further requests are then processed by the old sitemap until the generation is finished. From that time on, all documents are generated using the new sitemap.

Synchronous mode is very useful when you develop your application, because each change to the sitemap is reflected immediately. Asynchronous mode is more useful for a production environment in which the sitemap changes very rarely.

Although the automatic reloading of the sitemap seems to be a very useful feature, it has potential dangers. Assume that you change the sitemap to an invalid state, either by creating invalid XML or by making some other mistake that prevents Cocoon from being able to create the sitemap. The next request enters Cocoon, and the sitemap generation process is triggered.

In synchronous mode, the sitemap is generated immediately, but it fails due to the error you made beforehand. So you get a Cocoon error page, because Cocoon cannot process your request. The whole Cocoon installation is "dead" until you correct the error.

In asynchronous mode, the situation is even worse. When the request comes in, the sitemap generation process is started in the background. The current request and all further requests are processed by the old sitemap. The generation of the new sitemap fails because of the error. All further requests are then served using the old sitemap. If the changes made to the sitemap were only slight, it might take a while before anyone realizes that the old sitemap is still being used.

Cocoon provides a parameter that allows you to control whether the sitemap should be checked and reloaded. You can prevent Cocoon from reloading the sitemap by setting the attribute `check-reload` in cocoon.xconf to `false`. If you use the default, the sitemap is checked for reloading.

But what if you really changed the sitemap and you made a mistake? The first thing to do is check if your sitemap still contains well-formed XML, so load it into your favorite XML editor and check this. If it is well-formed but still does not work, you should use the logging facilities in Cocoon to find any error you perhaps made.

LogKit Configuration

Cocoon is based on the Avalon logging facilities, which are very flexible and powerful. You can configure details about what should be logged and what should be done with the log messages.

Cocoon has five log levels:

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `FATAL_ERROR`

Each component sends out log messages at one of these five levels. The LogKit then decides what should be done with this message.

Using the configuration, you can decide that only certain levels should really be logged to a file. For production sites, you will usually want to log only messages with a level of `ERROR` or `FATAL_ERROR`. In contrast, when developing your application, you will always want to see all levels. Because of the ordering of the different levels, each level contains all the following levels. Therefore, setting the level to `DEBUG` results in all messages being logged. Setting the level to `WARNING` results in all messages with a level of `WARNING`, `ERROR`, or `FATAL_ERROR` being logged.

The first thing you have to configure, however, is where Cocoon can find the LogKit configuration. This is done by another parameter in the web application deployment descriptor (web.xml), as shown in Listing 6.4.

Listing 6.4 **The Location of the LogKit Configuration in the Web Application Deployment Descriptor**

```
<!--
  This parameter indicates the configuration file of the LogKit management
-->
<init-param>
  <param-name>logkit-config</param-name>
  <param-value>/WEB-INF/logkit.xconf</param-value>
</init-param>
```

The standard place for the LogKit configuration is WEB-INF/logkit.xconf inside your Cocoon context directory. This configuration file is an XML document that describes the LogKit configuration. Listing 6.5 is a simple example.

Listing 6.5 **An Excerpt from the LogKit Configuration**

```
<logkit>
  <factories>
    <factory type="priority-filter" class=
    ➥"org.apache.avalon.excalibur.logger.factory.PriorityFilterTargetFactory"/>
    <factory type="servlet" class=
    ➥"org.apache.avalon.excalibur.logger.factory.ServletTargetFactory"/>
    <factory type="cocoon" class=
    ➥"org.apache.cocoon.util.log.CocoonTargetFactory"/>
  </factories>

  <targets>
    <cocoon id="cocoon">
      <filename>${context-root}/WEB-INF/logs/cocoon.log</filename>
      <format type="cocoon">
        %7.7{priority} %{time}   [%8.8{category}] (%{uri})
        ➥%{thread}/%{class:short}: %{message}\n%{throwable}
      </format>
      <append>true</append>
```

```
        <rotation type="revolving" init="1" max="4">
          <or>
            <size>100m</size>
            <time>01:00:00</time>
          </or>
        </rotation>
      </cocoon>

      <priority-filter id="filter" log-level="ERROR">
        <servlet>
          <format type="extended">%7.7{priority} %5.5{time}:
          ➥%{message}\n%{throwable}</format>
        </servlet>
      </priority-filter>
    </targets>

    <categories>
      <category name="cocoon" log-level="DEBUG">
        <log-target id-ref="cocoon"/>
        <log-target id-ref="filter"/>
      </category>
    </categories>
  </logkit>
```

The first part of the configuration file deals with factories for the logging targets. Factories are used inside component-based architectures to allow the flexible creation of components. They remove the need to "hard-wire" specific implementations into the system. You can compare this part of the configuration file with the components section of the sitemap, where you define the available generators, transformers, and so on.

These factories define components that are to receive the log events. In this example, the `cocoon` factory writes log events to a file. The `servlet` factory logs into the servlet log, and the `priority-filter` filters events.

These factories are then used in the `targets` section to instantiate real targets. When the `cocoon` target is instantiated, it receives the location of the log file (the `filename` tag) and in what format (the `format` tag) the log messages should be written.

The third part of the configuration is the `categories` section. Each component inside Cocoon can log into different categories. Usually they all log into the `root` category, which is also called `cocoon`.

So the LogKit configuration defines this category. A category gets a log level and a set of targets. All log events with this log level (or above) are sent to all the targets. So, in this example, all log events with `DEBUG` or higher are sent to a target called `cocoon` (logging into a file) and a target called `filter`.

This "filter" uses the priority filter to filter the log events. In this configuration, the filter discards all messages that do not have the level `ERROR` or `FATAL_ERROR`. Messages with one of these two levels are sent to the servlet target. So they are logged into the servlet log as well.

As you can see from this example, even a simple LogKit configuration can get very complex (and therefore complicated). But in most cases, it is sufficient to change the used log level. You can do this simply by changing the `log-level` attribute of the `cocoon` category. When you use a file-based configuration like this, you also can add new targets and categories without changing the code.

In case of a problem, you should have a look at the log file and see if you can find any description of the problem in the file. If the log level is not `DEBUG`, you should switch it. But be careful: A change to the log level (or any other change in the LogKit configuration) is not reflected immediately. You need to reinstantiate Cocoon in order for this to happen. You can force this by specifying the parameter `cocoon-reload` or by changing cocoon.xconf.

Changing the level to `DEBUG` causes the log file to become very large. Logging is also quite a time-consuming process, so you will want to set the level as low as possible (such as to `ERROR`) in a production environment.

How Requests Are Processed Inside Cocoon

Whenever a request for a document is sent to Cocoon, the root sitemap is taken to respond to the request. The pipelines section of the root sitemap is then processed top-down. All `map:pipeline` sections marked as internal-only using the attribute `internal-only` are skipped. The process follows the steps described next. For the moment, we will neglect the `views` (they are explained in a separate section), because they would only confuse this description:

- If a match directive is found, the matcher tests a value against a given pattern. If the value matches, the directives inside the matcher are executed next, and possible values from the matcher can be used by specific keys. If the value does not match, the next directive on the same level is executed next.
- If an action directive is found, the action is executed immediately. If the action returns keys for value substitution, the directives inside the action are executed next. If no keys are provided, the directive on the same level is next.
- If a selector directive is found, the selector performs the various test cases from top to bottom. When the value is equivalent to the first test case, the directives inside this case are executed next, and all others are ignored. If no test case matches, the default case (if it's available) defines the next directives to execute.
- If a generator directive is found, it builds the starting point for the XML processing pipeline. The next directive on the same level is executed. The generator is not yet started.
- If a transformer directive is found, the transformer is added at the end of the XML processing pipeline, but it is not executed yet. Then the next directive on the same level is executed.

- If a serializer directive is found, the serializer builds the end of the XML processing pipeline, and the buildup pipeline is executed. The generator feeds its XML through the various transformers. The serializer produces the document, and the processing is finished.
- If a reader directive is found, the reader delivers the document, and the processing is finished.
- If a redirect occurs, the processing is stopped. If the redirect points to a sitemap resource, it is processed. If the redirect is an external link, the client is redirected to it. If the link is internal, a new request is processed by Cocoon, starting at the main sitemap.
- If a mount for a subsitemap is found, the processing is passed on to the subsitemap. When the subsitemap processing is finished, the document is processed.
- If a content aggregation directive is found, this special generator is added as the starting point of the XML processing pipeline.
- If an error occurs, the error handler of the current `map:pipeline` is called.

As you can see from this flow description, actions, matchers, and selectors are executed immediately when the sitemap is processed. The same applies for a reader.

But generators, transformers, and serializers are not executed immediately. They are chained to build the processing pipeline. Only when this pipeline is complete (when a serializer is added) is the whole pipeline executed.

Because the XML is processed in this created pipeline, all other sitemap components not chained in this pipeline have no access to the XML. Thus, an action, matcher, or selector cannot be influenced by this XML, nor can they influence it.

Cocoon distinguishes between two pipeline types: the event pipeline and the stream pipeline. As the name implies, the event pipeline deals with SAX events. It consists of the usual XML processing pipeline (generator and transformers) without the serializer. A stream pipeline streams the final document to the client. It consists of only a reader or of an event pipeline in combination with a serializer.

For a Cocoon user, this information is important to know in order to understand caching (which we will explain later) and the `cocoon` protocol.

The `cocoon` protocol invokes an internal request to the sitemap. The resulting document can be used, for example, as the input for a generator or transformer or for content aggregation. All these components require XML. The generator reads produced XML, the xslt transformer uses stylesheets, and the content aggregation aggregates XML documents and generates from these documents one XML document.

But the `cocoon` protocol calls an arbitrary pipeline, which has a serializer at the end. It could, in the best case, return XML as a stream of characters or, even worse, HTML or any other format. How does this work? As you might guess, the answer is the event pipeline.

Whenever the cocoon protocol is used, only the event pipeline is built. Remember, the event pipeline is the XML processing pipeline without the serializer. So the event pipeline directly outputs XML as SAX events. Therefore, all components requiring XML can very easily use the cocoon protocol. Obviously, the cocoon protocol must not point to a pipeline using a reader.

Now let's get on with explaining these mysterious SAX events in detail.

SAX Event Handling

XML pipelines also work internally with the SAX model. Therefore, a generator sends SAX events to the following component in the pipeline. This component sends SAX events to the next one, and so on until the final serializer gets the final SAX events, serializes them, and creates the output document.

It might seem unimportant to a Cocoon user that the SAX model is used, but it has an impact on how pipelines must be built. SAX events have only one direction: from top to bottom, if you think about how they are written in the sitemap. It is not possible to send SAX events back up the pipeline.

A transformer transforms the incoming XML stream. There are two possible categories of transformers. In the first one, a transformer transforms the document as a whole, like the xslt transformer does. The stylesheet for the xslt transformer contains all the information for each node in the XML document.

The other category is a transformer that listens for specific XML elements that it will transform. For example, the sql transformer waits for special elements that set the SQL connection and the SQL query. All other elements surrounding the SQL statements are ignored. By ignored, we mean that they are passed unchanged from the sql transformer to the next component in the pipeline, as shown in Figure 6.2.

In order to get the sql transformer working, the incoming SAX events of the previous component in the pipeline (perhaps the generator) must contain those special elements for the sql transformer. So this is the first simple rule: If a component is listening for specific information, that information must be provided by a previous component in the pipeline.

There are more transformers that act like the sql transformer. The ldap transformer is another example of a transformer that reacts to special tags. It listens for some elements and then queries an LDAP system. If you want to build complex pipelines that have more than one transformer of this category, you have to think carefully about what you really want to do.

Imagine that you want to read an XML document from the local hard drive. This XML document contains information for the sql transformer. The sql transformer fetches data from the database that is then feed into the ldap transformer.

From these requirements, you should be able to build up your XML document. It should look like Listing 6.6.

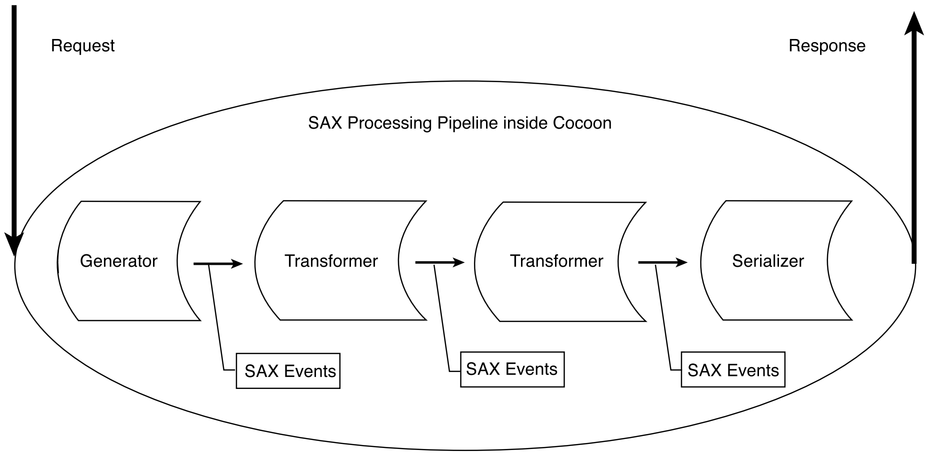


Figure 6.2 SAX event handling.

Listing 6.6 **An Example of Dependent Components**

```
<?xml version="1.0"?>
<document>
    <LDAP>
        <LDAP-INFORMATION>
            <SQL>
                 <SQL-INFORMATION/>
            </SQL>
        </LDAP-INFORMATION>
    </LDAP>
</document>
```

The information for the sql transformer is surrounded by the elements for the ldap transformer. Because the fetched data is the input for the LDAP query, it must be contained inside the LDAP elements.

In order to make the example work, you have to define your pipeline according to your XML document. As the ldap transformer waits for information from the sql transformer, the pipeline should look like Listing 6.7.

Listing 6.7 **A Pipeline of Dependent Components**

```
<map:generate src="document.xml"/>
<map:transform type="sql"/>
<map:transform type="ldap"/>
<map:serialize/>
```

The sql transformer needs to come before the ldap transformer. Why is this so? The answer lies in the SAX events. As mentioned, SAX events are sent in only one direction. The ldap transformer needs information from the sql transformer, so the SQL query must be done first.

If you put the sql transformer after the ldap transformer, the statements and elements for the sql transformer would be directly used as the information for the ldap transformer. This LDAP query would then fail, and the sql transformer would never get its information.

So the second important rule is this: When building pipelines, you need to be aware of the events or data flow. In other words, you need to know the dependencies between your transformation steps. For example, if transformer A needs information from transformer B, you have to put transformer B before transformer A in the pipeline, and the elements for transformer B must be nested inside those for transformer A.

Of course, you need not stick to this simple rule. In some cases, the information delivered from one transformer cannot be used directly by another transformer. Then you should use intermediate stylesheet transformation, which converts the data of the first transformer to usable input for the second transformer.

In the preceding example, the order of the components in the pipeline would still be the same, but you could then add a stylesheet transformation between the sql transformer and the ldap transformer stage. This stylesheet would convert the response from the sql transformer into a suitable request for the ldap transformer.

Using an intermediate stylesheet is very important if you have circular dependencies. Imagine a pipeline in which you first have a SQL query, and then a dependent LDAP query, and after that a second SQL query that needs information from the LDAP transformation.

The simple approach shown in Figure 6.3 will not work. If you follow the rule we set up, you would build the structure of the commands as set out in the first block at the beginning of the chain in the figure—first the outer tags for the last sql transformer, and then the tags for the ldap transformer, and inside them the tags for the first sql transformer. However, because a sql transformer is in front of the ldap transformer, the last sql transformer never receives any of its commands, because the first sql transformer will have already processed them. There is no way to tell each sql transformer which SQL tags are for the first transformer and which are for the second.

The only solution that works in a case like this is to use an intermediate stylesheet, as shown in Figure 6.4.

The starting document containing the commands must then contain only the LDAP query with the nested SQL query for the first sql transformer. After the ldap transformer in the pipeline, you need a stylesheet transformation, which adds the SQL statement for the last sql transformer around the data fetched from the LDAP query. This can then be processed by the following sql transformer.

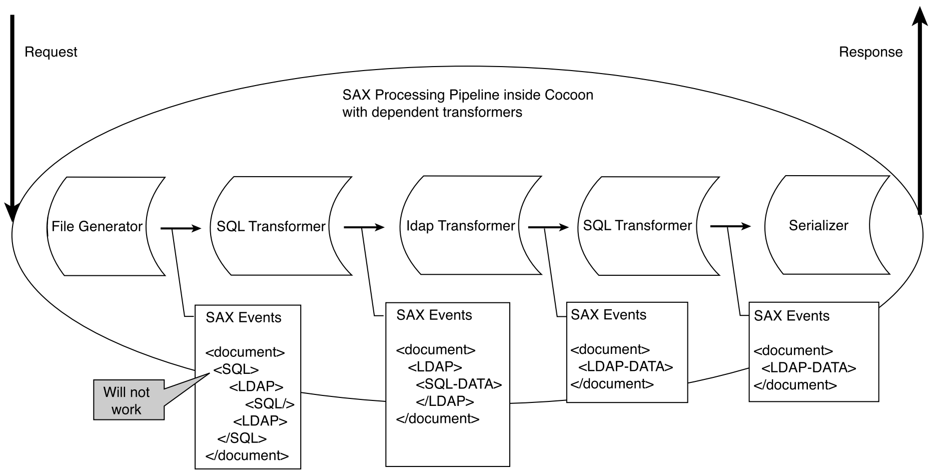

Request
Response
SAX Processing Pipeline inside Cocoon with dependent transformers
File Generator
SQL Transformer
Idap Transformer
SQL Transformer
Serializer
SAX Events <document> <SQL> <LDAP> <SQL/> <LDAP> </SQL> </document>
Will not work
SAX Events <document> <LDAP> <SQL-DATA> </LDAP> </document>
SAX Events <document> <LDAP-DATA> </document>
SAX Events <document> <LDAP-DATA> </document>

Figure 6.3 Incorrect chaining of dependent transformers.

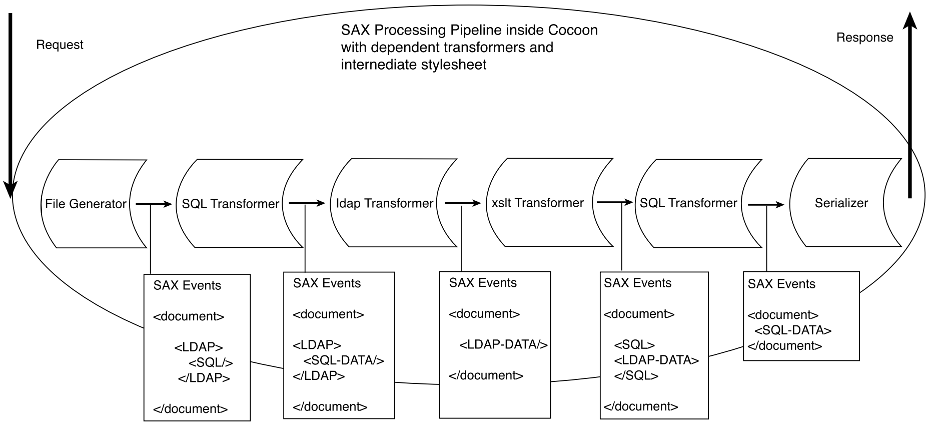

Request
Response
SAX Processing Pipeline inside Cocoon with dependent transformers and internediate stylesheet
File Generator
SQL Transformer
Idap Transformer
xslt Transformer
SQL Transformer
Serializer
SAX Events <document> <LDAP> <SQL/> </LDAP> </document>
SAX Events <document> <LDAP> <SQL-DATA/> </LDAP> </document>
SAX Events <document> <LDAP-DATA/> </document>
SAX Events <document> <SQL> <LDAP-DATA> </SQL> </document>
SAX Events <document> <SQL-DATA> </document>

Figure 6.4 Using an intermediate stylesheet.

As you can see from the example that uses transformers and intermediate stylesheets, pipelines can get quite complicated.You need to be aware of how things work in order to build your pipeline. However, in our experience with Cocoon, we have very rarely had such complex dependencies. It is more often the case that you need more than two transformers, but they are not dependent, so you do not need an intermediate stylesheet transformation.

This section introduced the additional files that control how Cocoon is configured. It also showed you how components in Cocoon can receive parameters through these configuration files. Cocoon components are based on design principles set out by the Apache project Avalon. Cocoon also uses the Avalon logging mechanism. We also looked at how a request is processed inside Cocoon and how the XML tags are sent through a pipeline as SAX events. After taking a user's look at the various configuration files, we can now return to the sitemap, which is the most important configuration file from a user perspective. We will look at the features not already explained in Chapter 4.

Advanced Sitemap Features

If you are already somewhat familiar with Cocoon, you will have noticed that we left out some features when we first introduced it. The main reason for this was to make it easier for first-time users to get started with Cocoon. Now that we have expanded on the first block of information with examples and the first version of the news portal application, we can complete the description of the sitemap features from a user perspective.

One of the most important functions in Cocoon is its ability to obtain data from various sources. This is done through different protocols. This section introduces some Cocoon-specific protocols. We will also explain some new sitemap component types and the views and resources sections of the sitemap. However, before we dive into the details, let's begin our look at the sitemap with a slightly different type of component—the action-set.

Action-Sets

Chapter 4 introduced the component type action, which can be used in any pipeline to fulfill a defined task. Cocoon also offers a more flexible approach to using actions: action-sets.

In contrast to other sitemap component types, an *action-set* is a combination of formerly defined actions that can be used in a pipeline as though it were a single component. Defining an action-set is like defining a pipeline, which is a combination of sitemap components. An action-set is also defined inside its own sitemap section, the `map:action-sets` section.

Each action-set is introduced with the `map:action-set` element, which receives a unique name via the attribute `name`. Inside this element you can enter as many actions as you like, as shown in Listing 6.8. You arrange a set of actions to form a group.

Listing 6.8 **An Example of an Action-Set**

```
<map:action-sets>
  <map:action-set name="myactionset">
   <map:act type="log-start-action"/>
   <map:act type="add-action" action="add"/>
   <map:act type="del-action" action="delete"/>
   <map:act type="log-end-action"/>
  </map:action-set>
</map:action-sets>
```

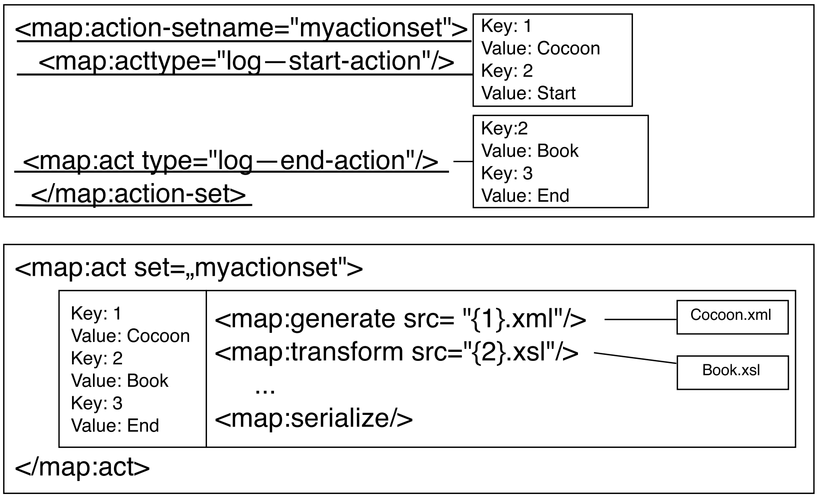
A defined action-set can be used in the pipeline just like a normal action via the tag `<map:act set="myactionset"/>`. The difference is that the attribute `set` is used instead of `type`.

If you use an action-set, all actions of this set are called in the order they are defined. In addition, it is possible to selectively call an action inside an action-set. To do this, you can define each action in the action-set to have an attribute `action`. If the current request being processed by the pipeline contains a request parameter called `cocoon-action`, the action with the corresponding `action` attribute in the action-set is called.

In Listing 6.8, if the action-set `myactionset` is used, `log-start-action` is invoked. If the request currently being processed contains a `cocoon-action` parameter with the value `add`, the action `add-action` is invoked. If instead the `cocoon-action` parameter has the value `delete`, `del-action` is invoked. Finally, `log-end-action` is always invoked. The `cocoon-action` parameter can contain only one value, so either `add-action` or `del-action` or neither is invoked, but never both at the same time.

Do you remember value substitution, discussed in Chapter 4? An action can provide key-value pairs for other sitemap components. All components nested inside the action have access if they know the key's name.

Value substitution for action-sets is very similar, as shown in Figure 6.5. Whereas all values of an action are accessible using the key for nested components, all values of all called actions of the action-set are available inside the action-set element. Therefore, the value substitution algorithm collects all values from all actions. However, if two actions use the same key inside an action-set, only the value of the last action is available. It overrides the previous one.



<map:action-setname="myactionset">
<map:acttype="log—start-action"/>
Key: 1
Value: Cocoon
Key: 2
Value: Start
<map:act type="log—end-action"/>
</map:action-set>
Key:2
Value: Book
Key: 3
Value: End
<map:act set=„myactionset">
Key: 1
Value: Cocoon
Key: 2
Value: Book
Key: 3
Value: End
<map:generate src= "{1}.xml"/>
<map:transform src="{2}.xsl"/>
...
<map:serialize/>
Cocoon.xml
Book.xsl
</map:act>

Figure 6.5 Value substitution for action-sets.

Using action-sets allows you to build modular components that can be used flexibly in pipelines. Often, actions are used to control the flow inside a pipeline and to determine such things as which data source needs to be accessed for the current request. Using the various protocols available in Cocoon allows a variety of different possibilities when it comes to retrieving data or calling internal functions as part of processing.

Protocols

A concept widely used inside sitemaps is the definition of URIs. On the one hand, you define the sitemap to spawn a virtual URI space, which is served by Cocoon, but more obviously, you use URIs to specify which resources are to be read by the various sitemap components. For example, the file generator needs an XML document as input; the xslt transformer processes a stylesheet, and so on.

As we discussed in Chapter 4, you can use any protocol supported by Cocoon to define your URIs and to access resources. For example, you can use an HTTP connection to retrieve an XML document from a remote server, an FTP connection to read a stylesheet, or the `file` protocol to read a file from the local hard drive.

In addition to these standard protocols, Cocoon offers additional protocols that can also be used inside the source definition of a generator, a transformer, or any other component. All these protocols follow the general pattern for building URIs: `*protocolname*://*path to the resource*`. Cocoon supports a resource protocol, a context protocol, a cocoon protocol, and a protocol that is used implicitly.

The Implicit Protocol

The most important protocol is the implicit protocol, which you have already used without noticing. As the name suggests, this protocol is used implicitly whenever a protocol definition is missing. For example, if you write something like `<map:generate src="mydocument.xml"/>`, Cocoon can handle it even though the protocol is missing.

How Cocoon handles this depends on how you deployed the web application. There are two ways of doing this. You can bundle everything into a web archive (WAR) file, or you can deploy everything as individual files. If your web application is not a WAR file, Cocoon implicitly adds the file protocol. All the references are then resolved relative to the location of the current sitemap using the file protocol. If you have a WAR file, Cocoon implicitly adds the protocol provided by the servlet engine to access these files, again relative to the location of the current sitemap.

This means that you don't need to worry about explicitly using a protocol when you define your pipelines and the resources they are to access. However, it is always better to add the protocol explicitly, because this makes your sitemap entries more readable to someone who is not as familiar with the inner workings of Cocoon.

The Context Protocol

The `context` protocol is used to access any resource belonging to the Cocoon web application. If you deployed the Web application from a directory on your hard drive, the `context` protocol is directly mapped to the filesystem. So the resource definition `context://mydocument.xml` is translated to a file URI pointing to the Cocoon web application directory—more precisely, to a file called mydocument.xml inside this directory.

If you have deployed your Cocoon web application as a WAR file, you access the resources inside the WAR file using the `context` protocol. The argument following the protocol is a path relative to the root of the WAR file. So again, `context://mydocument.xml` references a file named mydocument.xml stored at the root of the WAR file.

So, if you use the `context` protocol, you can abstract from how you deployed your Cocoon web application. Cocoon can determine whether to use the filesystem or the WAR file to resolve the resource you might want to load.

Whereas the `context` protocol can be used to access resources inside a WAR file or in a filesystem, the `resource` protocol can locate resources inside Java archives (JAR files).

The Resource Protocol

Because Cocoon is implemented using Java, it consists of several JAR files that contain the various parts. A JAR file can contain more than Java code. It can hold any resource, such as images, XML documents, or stylesheets. All these JAR files are located in the WEB-INF/lib directory of your Cocoon context and are loaded automatically at startup by your servlet engine.

If you want to read such a resource, you can simply use the `resource` protocol followed by a path specifying the resource precisely. Cocoon then searches all loaded JAR files for this resource. For example, `resource://org/apache/cocoon/components/language/markup/xsp/java/xsp.xsl` specifies a file named xsp.xsl. This file is in one of the JAR files in the directory structure org/apache/cocoon/components/language/markup/xsp/java. So one JAR file has a root directory called org, which has a subdirectory named apache, and so on.

So far, we have looked at protocols that allow you to access static resources. But what if you want to access resources that are not available as a unit but must be built by a process?

The Cocoon Protocol

Because Cocoon is a processing framework that can build documents using processing pipelines, sooner or later you might want to use a Cocoon resource as the input for a generator in another resource. Doing this lets you use the result of a resource as the starting point for a pipeline or as the input for any other component. So what you need is a way to access the result of one pipeline from another pipeline.

The `cocoon` protocol allows you to do exactly this. It accesses pipelines inside the sitemap. For example, `<map:generate src="cocoon:/helloworld"/>` uses the `file generator` that reads an XML document created by a request for the document helloworld against the sitemap.

Whenever you use the `cocoon` protocol, Cocoon internally processes a new request for the specified document and uses this result for the ongoing processing of the original request.

The main use of this protocol is content aggregation, in which you can build a document from more than one source, as you will see in the next section. But you can, of course, use this protocol everywhere in the sitemap—for example, as an input to the xslt transformer.

All in all, the different protocols allow a very flexible mechanism for accessing data sources. You can also add your own new protocol if you like. We will show you how to do this in Chapter 9, "Developing Components for Cocoon." As soon as you have set up pipelines to access the various data sources, content aggregation allows you to combine them inside the sitemap.

Content Aggregation

When designing web applications, such as a portal, you often need to build complex documents consisting of several parts. Consider a typical information web site. The document consists of a header displaying, for example, the name of the company, a navigation bar, a block of content that was chosen from the navigation bar, and perhaps a footer displaying some static information.

Although this is a single document, it consists of four parts: header, navigation bar, content, and footer. Many documents follow this scheme. For each piece of content you display on your web site, you have exactly one document consisting of three static parts—header, navigation bar, and footer—and the content. How can documents like this be created easily?

One solution is to define a separate pipeline for each document. Each pipeline then reads an XML document containing not only the content but also XML information for the header, footer, and navigation bar. The XML information is then formatted by a stylesheet to present the complete page.

The problem with this solution is that you cannot access just the content. You would need to do this if you wanted to format the data into a PDF document, where you do not need the additional information on a header or footer.

Even worse, defining separate pipelines mixes concerns. The content should not need to know about the other parts, and vice versa. So the ideal solution would be to create the parts as separate documents and then be able to combine them.

That's where content aggregation comes in handy. You can define a document that is a combination or aggregation of other documents. To do this, you need to define a pipeline in the sitemap and use some tags specific to content aggregation, as shown in Listing 6.9.

Listing 6.9 **An Example of Content Aggregation**

```
<map:pipeline internal-only="true">
    <map:match pattern="header">
        <map:generate src="header.xml"/>
        <map:serialize type="xml"/>
    </map:match>
    <map:match pattern="footer">
        <map:generate src="footer.xml"/>
        <map:serialize type="xml"/>
    </map:match>
    <map:match pattern="navigation">
        <map:generate src="footer.xml"/>
        <map:serialize type="xml"/>
    </map:match>
    <map:match pattern="*">
        <map:generate src="docs/{1}.xml"/>
        <map:serialize type="xml"/>
    </map:match>
</map:pipeline>
<map:pipeline>
    <map:match pattern="docs/*">
        <map:aggregate element="document">
            <map:part src="cocoon:/header"     element="header"/>
            <map:part src="cocoon:/navigation" element="navigation"/>
            <map:part src="cocoon:/{1}"        element="content"/>
            <map:part src="cocoon:/footer"     element="footer"/>
         </map:aggregate>
         <map:transform src="all2html.xsl"/>
         <map:serialize type="html"/>
    </map:match>
 </map:pipeline>
```

Listing 6.9 has some new elements we need to define before proceeding with our discussion. The most obvious one is the `map:aggregate` command. It is used inside an XML processing pipeline as a replacement for the `map:generate` instruction you would have in a normal pipeline. It defines a content aggregation of the parts, which are defined as nested `map:part` elements. In our example, we are building a complete document containing a header, a footer, navigation, and content. The attribute `element` of `map:aggregate` defines the root element of the generated XML document. Each part can have an element, under which you can find this part in the aggregated content. See Listing 6.10.

Listing 6.10 **Aggregated Content**

```
<?xml version="1.0"?>
<document>
    <header>
        <!-- here is the content of the header document -->
    </header>
    <navigation>
        <!-- here is the content of the navigation document -->
    </navigation>
    <content>
        <!-- here is the content of the content document -->
    </content>
    <footer>
        <!-- here is the content of the footer document -->
    </footer>
</document>
```

As you can see from Listing 6.10, the content is aggregated by the various parts. The following components in the pipeline, such as the xslt transformer, can transform this aggregated document into HTML or whatever format is required.

You do not need to define an `element` attribute for a part. If it is omitted, the part's content is directly included under the document's root node.

The `cocoon` protocol is used for each part. Therefore, each part is defined by another pipeline somewhere in the sitemap. In this example, these pipelines are all inside their own `map:pipeline` section in the sitemap.

Normally, because the separate parts are pipelines in the sitemap, you would be able to access them individually using a browser. This is not what you want, however, because it would result in your receiving only part of a document.

Because you do not want to be able to receive only the document header or navigation or footer or the content itself without the surrounding parts, this `map:pipeline` section is protected with the attribute `internal-only` set to `true`. With this attribute set, all marked `map:pipeline` sections are skipped when Cocoon processes a request. These pipelines can only be invoked "internally" by using the `cocoon` protocol from within another pipeline.

You can control content aggregation using three more attributes for an aggregated part: `prefix`, `ns`, and `strip-root`. So, a full-featured part might look like this:

```
<map:part src="cocoon:/header" strip-root="true" prefix="header"
ns="header://version/1.0" element="header"/>
```

The top-level element for the header part is called `header`. It gets the namespace defined by the attribute `ns`. The attribute `prefix` is used to define the prefix. So the top-level element looks like this:

```
<header:header xmlns:header="header://version/1.0"/>
```

You can leave out the attribute `prefix`.

In addition, you can use the attribute `strip-root` with a Boolean value. If it is set to `true`, the root element of the aggregated part is stripped off. So if the pipeline for the document `header` has the root element `myheader`, it is not included. All children of the `myheader` element are included under the root element of the part.

Although you might get the impression that you must use the `cocoon` protocol to aggregate parts, this is not true. You can use any protocol available. The simplest case is aggregating XML files.

Later you will see practical examples and tips and a real-world example of content aggregation. This example—the Cocoon online documentation—also uses some other features not explained yet. One of them is the concept of subsitemaps.

Subsitemaps

When you develop large web applications, or when more than one person is editing the sitemap, it can be very difficult to maintain, because it is a single big XML document.

To simplify sitemap editing and maintenance, Cocoon offers the concept of subsitemaps (see Figure 6.6). A subsitemap looks like a normal sitemap, but it is *mounted* into the main sitemap. By mounting, we mean that you usually define a URI prefix for a subsitemap. All incoming requests starting with this prefix are then handled by the subsitemap.

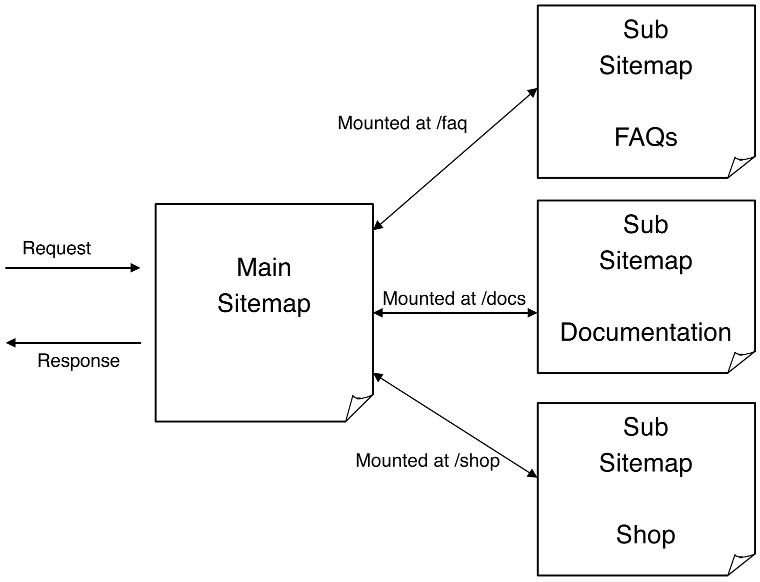


Figure 6.6 Subsitemaps.

The mount points allow you to cascade your sitemaps. This ensures more readability and supports sitemap editors managing the web application. Each subsitemap can then be maintained by a different person. After mounting, you can imagine the whole construction as a tree, with the main sitemap being the root.

When a request for a document enters Cocoon, it is always processed by the main sitemap first. If a mount point for a subsitemap is reached, the processing is passed to the subsitemap (see Listing 6.11).

Listing 6.11 **A Basic Example of Mounting a Subsitemap**

```
<map:match pattern="faq/*">
  <map:mount check-reload="yes" src="faq/sitemap.xmap" reload-method="synchron"/>
</map:match>
```

The `src` attribute defines the location of the subsitemap. If it ends in a slash, `sitemap.xmap` is automatically appended to find the sitemap. Otherwise, Cocoon assumes that the `src` attribute directly defines a file containing the subsitemap.

Like the root sitemap, subsitemaps can be configured with respect to reloading. The configuration is similar to that of the root sitemap in cocoon.xconf. The `check-reload` attribute, which defaults to `true`, defines whether changes to the subsitemap should be reflected.

If this reload checking is activated, `reload-method` specifies whether the subsitemap regeneration should be synchronous or asynchronous. Here the same rules apply as those explained for sitemap reloading at the beginning of this chapter.

The fourth attribute for `map:mount` is the `uri-prefix` attribute. As explained, when a request enters Cocoon, the root sitemap is processed with the incoming URI. Now, if a mount point for a subsitemap is reached and Cocoon processes this subsitemap, the same URI is passed in.

For example, if you requested for a document called faq/installation, and the mount defined in Listing 6.11 is reached, this URI is passed on to the subsitemap unchanged. Even though you mounted the sitemap under the path faq, you still have to match this prefix inside the subsitemap. If you want to mount your subsitemap under a different path, such as old-example, you have to update the root sitemap to add a prefix and also all matches inside your subsitemap to reflect this new location (see Listing 6.12).

Listing 6.12 **Mounting a Subsitemap with Prefix**

```
<map:match pattern="faq/*">
  <map:mount uri-prefix="faq/" check-reload="yes" src="faq/sitemap.xmap"
  ➥ reload-method="synchron"/>
</map:match>
```

To avoid these problems and to make the subsitemap more independent from the root sitemap, you can use the `uri-prefix` attribute to pass only the important part into the subsitemap. In the example, you want to pass only `installation` into the subsitemap.

Because the subsitemap is mounted using the path faq/, you have to remove it from the URI that is passed to the subsitemap. And that's exactly what you do with the `uri-prefix` attribute. You define a string starting on the left side of the URI. It is removed from the original when processing is passed to the subsitemap. In the example, you want to remove faq/ and therefore give this value to the `uri-prefix` attribute. Cocoon automatically checks for a trailing slash, so writing either `faq` or `faq/` is equivalent. However, we suggest that you add the slash to make it easier to read your entry.

A subsitemap can look the same as the main sitemap. It can have the same sections, starting with a components section and ending with a pipelines section.

In fact, these two sections are the ones required to make a subsitemap work, as you can see from Listing 6.13. But you can, of course, have all the other sections as well.

Listing 6.13 **An Example Subsitemap**

```
<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">
    <map:components>
        <map:generators default="file"/>
        <map:transformers default="xslt"/>
        <map:readers default="resource"/>
        <map:serializers default="html"/>
        <map:selectors default="browser"/>
        <map:matchers default="wildcard"/>
    </map:components>
    <map:pipelines>
        <map:pipeline>
            <map:match pattern="*">
                <map:generate src="{1}.xml"/>
                <map:transform src="faq2html.xsl"/>
                <map:serialize/>
            </map:match>
        </map:pipeline>
    </map:pipelines>

</map:sitemap>
```

All requests entering the main sitemap that start with the prefix faq/ are passed to the subsitemap. The prefix is removed from the URI, and the subsitemap receives only the part of the URI that comes after this prefix.

So a request for `faq/installing` is passed as a request for `installing` to the subsitemap. As defined in the subsitemap in Listing 6.13, the request reads an XML document named installing.xml, transforms it, and serializes it as HTML.

As you can see from this example, you can use all the sitemap components from the main sitemap without declaring them again, but in order to make the subsitemap work, you have to declare the default component for each component type.

However, in order to separate concerns, you can define specific sitemap components in the components section of your subsitemap. These components are then accessible only in this subsitemap, not in the parent sitemap. You can also redefine a component inherited from the parent sitemap but with another configuration. Again, this configuration is used only in the subsitemap.

Using subsitemaps helps you manage your web site. Each sitemap editor has his own separate sitemap that cannot interfere with the other sitemaps. Even if a subsitemap stops working due to a mistake made in the subsitemap, the main sitemap and all other subsitemaps still work.

The hierarchical structure of sitemaps is not limited to two levels (one main sitemap with several subsitemaps). Because a subsitemap is a full-featured sitemap that inherits from the parent (or main) sitemap, it can have its own subsitemaps. So you can build a big tree of sitemaps using this concept.

Each subsitemap can have its own directory to store resources such as XML documents and stylesheets. All URIs that do not have an explicit protocol are resolved according to the sitemap's directory. In the example, the subsitemap is stored in the directory faq. The pipeline for a document reads an XML document that is resolved relative to this directory faq.

Apart from using the concept of subsitemaps to maintain your web site, you can also use views to organize what you send to the client application.

Views

Chapter 4 glossed over the explanation of the `map:views` and `map:resources` sections in the sitemap. Let's now fill in this gap, starting with `views`.

A request you send to Cocoon is mapped to a pipeline in the sitemap. That pipeline uses a combination of components to generate an end result, a document that is returned to you as a result of your request. You can think of the end result as being the default view of the document generated by that particular pipeline. However, Cocoon also lets you configure and request other views of a particular document.

Cocoon offers a wide variety of configurable views for its documents. You can request a document's content view, and you will get the content in that document's XML format. Or you can ask for a document's link view and get all the links to other documents contained in this document.

The views concept is complex. So we'll start our discussion of views by looking at some simple examples and examining some use-cases. The first thing you need to know is how to specify which view of the document you want when sending the request to Cocoon. You do so using the request parameter `cocoon-view` with the value of the view name you ask for. So if you ask for `http://localhost:8080/cocoon/helloworld?cocoon-view=content`, you receive that document's content view.

The more complex question is how Cocoon knows what to do when a view is requested. Generally speaking, a view is an alternative pipeline for a document. It starts like the original pipeline for the document, but it has a different ending.

Assume that you have a standard pipeline consisting of a file generator, an xslt transformer, and an html serializer. You can then define a different view using the same file generator but a different transformer and serializer.

A view definition consists of two parts, as shown in Listing 6.14. The first part specifies which parts or beginning of the original pipeline should be used for the view. The second part defines the alternative ending. The ending is defined in the `map:views` section of the sitemap.

Listing 6.14 **Views**

```
<map:views>
    <map:view name="content" from-label="content">
        <map:serialize type="xml"/>
    </map:view>
    <map:view name="links" from-position="last">
        <map:serialize type="links"/>
    </map:view>
</map:views>
```

For each possible view, you create a `map:view` element with the attribute `name` specifying the view's unique name. Inside this element, you define the pipeline's ending. Because this is only the ending, you must not define a generator. However, you can use transformers, and you must provide a serializer.

Listing 6.14 shows two defined views: the content view and the links view. Each new view contains only a serializer. Looking at the links view, you can see that the attribute `from-position` has the value `last.` This tells Cocoon where the new pipeline should take over from the original when the links view is requested. In this case, the alternative ending for this view starts at the last position of the original pipeline.

In other words, the serializer of the original pipeline is ignored, and instead, all sitemap components enclosed in this view are appended. So the links view differs from the original document in that it uses the `links` serializer (see Listing 6.15).

Listing 6.15 **The Link Serializer**

```
<map:serializers>
   <map:serializer name="links"
   ➥ src="org.apache.cocoon.serialization.LinkSerializer"/>
</map:serializers>
```

The link serializer is a special serializer that outputs plain text. It extracts all links and references from a document and puts each link in a separate line of the output text. These links and references are searched for in the original document by searching for the attributes `src` and `href`.

Another possibility is to define the value `first` for the view's `from-position` attribute. Then the alternative pipeline starts immediately after the original generator.

But Cocoon wouldn't be Cocoon if these were the only possibilities for defining views! You can define more fine-grained views by using the attribute `from-label` on the view. The value of this attribute marks a label that can be used in the original pipeline for the sitemap components.

With this label attached to sitemap components such as generators and transformers, you define which components of the original pipeline should be used for the view. Listing 6.16 shows an example.

Listing 6.16 **An Example of Labeled Views**

```
<map:generators default="file">
    <map:generator name="file" label="content"
    ➥src="org.apache.cocoon.generation.FileGenerator"/>
    <map:generator name="html" src="org.apache.cocoon.generation.HTMLGenerator"/>
    ...
</map:generators>
...
<map:pipeline>
    <map:match pattern="document_one">
        <map:generate src="document.xml"/>
        <map:transform src="document2html.xsl"/>
        <map:serialize/>
    </map:match>
    <map:match pattern="document_two">
        <map:generate src="page.html" type="html"/>
        <map:transform label="content" src="restructure.xsl"/>
        <map:transform src="document2html.xsl"/>
        <map:serialize/>
    </map:match>
</map:pipeline>
```

The component definition of the file generator is labeled with a label called `content`. This indicates that whenever a view is requested and this view uses the label `content`, the generator is included in the pipeline for this view. Similarly, you can mark other generators and transformers in the components section as well.

The pipeline for the first document, called `document_one` (see Figure 6.7), is assembled by the file generator, an xslt transformer, and the html serializer. When the content view is requested, Cocoon looks at the `map:views` section and finds the definition for this view. This view indicates that the label `content` is used. During the pipeline assembly, the components for this pipeline are checked for the label.

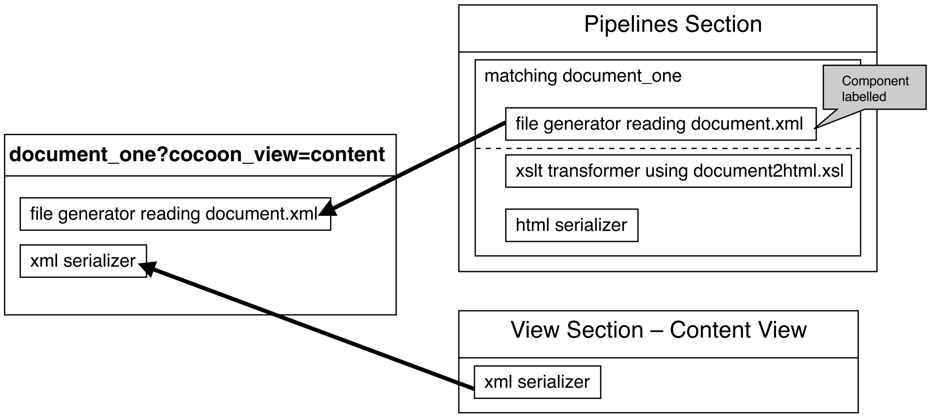

Pipelines Section
matching document_one
Component labelled
file generator reading document.xml
xslt transformer using document2html.xsl
html serializer
document_one?cocoon_view=content
file generator reading document.xml
xml serializer
View Section – Content View
xml serializer

Figure 6.7 A simple example of using views.

The file generator is labeled, so it is used. If a component is labeled, it is added to the pipeline for the view, and the usual pipeline processing is passed to the views section. All other sitemap components of the original pipeline are ignored, and the components of the views section are appended.

The pipeline for the second example (`document_two`), shown in Figure 6.8, is assembled by the html generator, two xslt transformers, and the html serializer. Note that neither the html generator nor the xslt transformer is labeled in the components section. When the content view of this document is requested, the original pipeline is searched for the label.

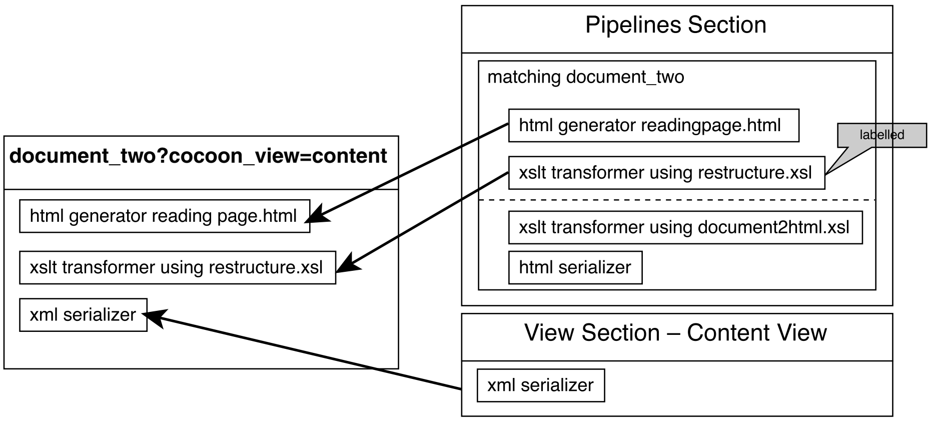

Pipelines Section
matching document_two
html generator readingpage.html
labelled
xslt transformer using restructure.xsl
xslt transformer using document2html.xsl
html serializer
document_two?cocoon_view=content
html generator reading page.html
xslt transformer using restructure.xsl
xml serializer
View Section – Content View
xml serializer

Figure 6.8 An advanced example of using views.

In general, the xslt transformer is not labeled, so it usually isn't added to the pipeline for the view. But for this special pipeline, you can indicate that the transformer should be added by giving it an attribute `label` with the value `content`. The first xslt transformer is labeled using the attribute `label` with the given value.

The process here is the same as in the first example. All sitemap components are added to the pipeline until one component is labeled. This component is added as well, but the following ones are skipped. Then the view's sitemap components are appended. For this example, the view is assembled from the html generator, the first xslt transformer, and the xml serializer from the content view.

Regardless of whether the label is defined in the components or pipelines section of the sitemap, the original sitemap is left immediately after the first component containing the label. Even if you have more than one component in the pipeline marked with the required label, only the first component containing it is used.

As you will see at the end of this chapter, the links view is important for the offline generation of documents using Cocoon's command-line interface.

Now that you know about Cocoon's views, you know about nearly all of a sitemap's sections. So, let's discuss the last one.

Sitemap Resources

The last section we have yet to explain is the `map:resources` section (see Listing 6.17). This section is very similar to the `map:pipeline` section. You can define XML processing pipelines containing a generator, transformers, and a serializer and give this pipeline a name for further use in the `map:pipelines` section of the sitemap.

Listing 6.17 **An Example of a Sitemap Resource**

```
<map:resources>
    <map:resource name="Not authorized">
        <map:generate src="notauthorized.xml"/>
        <map:transform src="tohtml.xsl"/>
        <map:serialize/>
    </map:resource>
</map:resources>
```

You can refer to this resource from the `map:pipelines` section using the unique name for these sitemap resources. So a sitemap resource can be compared to a macro or a placeholder.

Currently, the only place in Cocoon where you can use sitemap resources is for redirects.

Redirects

Basically, a redirect allows you to jump from one pipeline to another. You can redirect to a totally different URI or to a previously defined sitemap resource. Listing 6.18 shows two examples.

Listing 6.18 **Examples of Redirects**

```
<map:redirect-to uri="helloworld"/>
<map:redirect-to resource="Not authorized"/>
```

Unfortunately, the semantics of the `map:redirect` statement differ a bit from the semantics of the other sitemap components. Usually if you specify a source, such as for a generator, and you do not specify a protocol for the URI, Cocoon automatically adds the `context` protocol.

However, for a redirect to a relative URI, this is not the case. Cocoon implicitly adds the same protocol used to request the original document. For example, if you request a document with `http://localhost:8080/cocoon/original_document`, and this results in the execution of the previous redirect to `helloworld` as shown in Listing 6.18, Cocoon generates a new URI using the old one as a base. The redirect then references `http://localhost:8080/cocoon/helloworld`. So a relative URI is translated into an absolute URI.

Cocoon does not directly process redirects. Instead, an HTTP response to the client is generated. This response contains the information to process a redirect in addition to the redirect URI as content. The client itself recognizes this redirect and starts a new request with the new URI. Whenever you use a redirect, this results in at least two requests to your server. The first one identifies the redirect, and the second requests the redirected document.

If you redirect to a sitemap resource, the processing flow is continued in the new sitemap resource. Thus, the sitemap components defined in this resource are executed.

Now that you know about all the additional sitemap features and some Cocoon configuration points, it is time to bring in two new components and show you some examples that use them and the concepts described in this chapter.

Connecting to a Database

You can use the sql transformer in a pipeline to integrate a database as one of the data sources in a Cocoon application. Using this transformer, you can send any SQL command to a database. The transformer is controlled by commands contained in the XML stream processed by the transformer. If the SQL command fetches data from the database, the data is converted into XML.

You might wonder why this is a transformer and not a generator. The key point is usability. In general, SQL statements can have many options and parameters. This starts with specifying the database to use, the tables, and the rows, and it ends with complex information such as search phrases. If you want to use a generator, you have to specify all this in the sitemap as parameters for the generator. Changing a simple value would then require changing the sitemap.

Using a transformer allows you to build more-complex pipelines in which the information on what to fetch from the database is determined at runtime using the file generator, for example. When the request is processed, the file generator reads an XML file that contains the actual parameters for the transformer. Because the file generator can request the XML file via a protocol such as HTTP, this allows the dynamic generation of those commands.

Listing 6.19 shows the configuration of the sql transformer in the sitemap and how to use it in a pipeline.

Listing 6.19 **SQL Transformer**

```
<map:transformers>
    <map:transformer name="sql"
                  src="org.apache.cocoon.transformation.SQLTransformer"/>
</map:transformers/>
...
<map:pipeline>
    <map:match pattern="test">
        <map:generate src="document.xml"/>
        <map:transform type="sql"/>
        <map:transform src="tohtml.xsl"/>
        <map:serialize/>
    </map:match>
 </map:transform>
```

You can send any valid SQL command to the database. This is triggered by your XML document. Listing 6.20 shows an XML document that is read by the file generator and then is transformed by the sql transformer.

Listing 6.20 **A Simple SQL Example**

```
<document>
    <sql:execute-query xmlns:sql="http://apache.org/cocoon/SQL/2.0">
        <sql:use-connection>personnel</sql:use-connection>
        <sql:query>
            select id,name from department_table
        </sql:query>
    </sql:execute-query>
</document>
```

The sql transformer is triggered by XML elements that have the transformer's namespace, `http://apache.org/cocoon/SQL/2.0`. Each command is started by the element `execute-query`. Nested inside this element is all the information for the sql transformer, a combination of elements and text information.

The element `use-connection` defines which connection (or database) should be used for the SQL command. The following example will show you how you can configure database connections. For now, just assume you have defined a database connection named `personnel`.

Inside the `query` element, you can see the actual SQL command to be sent to the database. When the sql transformer receives such an XML block, it removes it from the XML document. If the SQL command fetches some data, this data is converted to XML and is inserted instead of the XML block controlling the sql transformer.

How is this data converted? An element rowset is created. Inside this element for each fetched row, an element named `row` is created. Inside this element, for each fetched column of this row, an element is created and is named the same as the column name. Inside this element is a text node with the value of that column from the database. All these elements get the namespace of the sql transformer.

You could then simply add a stylesheet to the XML processing pipeline, converting the rowset to an HTML table or whatever you like. The output displayed in Listing 6.21 is an intermediate XML document that is created during the pipeline processing. Because you will receive HTML in your browser, you will never notice this document; you will see only the starting XML document and the final output.

Listing 6.21 **The Document after a SQL Transformer Run**

```
<document xmlns:sql="http://apache.org/cocoon/SQL/2.0">
    <sql:rowset>
        <sql:row>
            <sql:name>Matthew</sql:name>
            <sql:id>1</sql:id>
        </sql:row>
        <sql:row>
            <sql:name>Carsten</sql:name>
            <sql:id>2</sql:id>
        </sql:row>
    </sql:rowset>
</document>
```

But what if your resulting document does not display the data you wanted? You need to know what the sql transformer has output in order to see if your SQL statement is working. You can, of course, change your document's pipeline definition. Instead of using a stylesheet to produce HTML and the html serializer, you can simplify the pipeline by removing the stylesheet and using the xml serializer. This shows you the data delivered by the sql transformer directly in your browser.

Another answer to this problem is to use the log transformer to see what is happening in the pipeline.

Logging

Usually pipelines consist of three or more sitemap components, starting with a generator, going to some transformers, and ending with a serializer. In the case of the file generator, you can see the starting XML document that is read by this component and the end result of the pipeline processing.

But what can you do if your output document doesn't look as you expected? One simple solution is to change your pipeline. Just remove all transformers after the component you want to test, and add the xml serializer. You will get the output of the transformer you want to test directly in XML.

If this stage of your pipeline looks right, you can then remove the next transformer in the chain and look at that output, and so on until you know where the fault is.

Another possibility is the log transformer (see Listing 6.22), which can be chained between two sitemap components. As the name suggests, this transformer logs the output of the sitemap component before the log transformer.

Listing 6.22 **The Log Transformer**

```
<map:transformers>
    <map:transformer name="log"
                  src="org.apache.cocoon.transformation.LogTransformer"/>
</map:transformers/>
...
<map:pipeline>
    <map:match pattern="test">
        <map:generate src="document.xml"/>
        <map:transform type="sql"/>
        <map:transform type="log">
            <map:parameter name="logfile" value="logfile.log"/>
            <map:parameter name="append" value="no"/>
        </map:transform>
        <map:transform src="tohtml.xsl"/>
        <map:serialize/>
    </map:match>
 </map:transform>
```

In Listing 6.22, the output of the sql transformer is logged. When no parameter is set to the log transformer, it outputs everything to the servlet log of your servlet engine. But you can, of course, redirect the output to a file on your local hard drive. The sitemap parameter `logfile` defines the location of that file. With the parameter `append`, you can specify whether a new log file should always be written, or if the output should be appended to an existing file.

But be careful with using the log transformer in a servlet environment. It is not safe for concurrent requests. So if more than one client requests a document containing the log transformer, the output is mixed by these two pipelines. So for debugging, you should be sure that only one client invokes the request at a time.

This section covered the advanced features of the sitemap. You saw that a Cocoon application is not limited to just one sitemap, but that sitemaps can be cascaded. This feature is particularly useful when the application consists of separate parts. Using the available protocols and components such as the sql transformer, you can integrate existing data sources into your application. Content aggregation allows configured information sources to be flexibly combined into a single document. The document you receive as a pipeline's output is only one of the views Cocoon can provide. Using the views concept, you can define alternative pipelines that can return, for example, only the content or the links of a particular document. You can use the logging mechanism to check on what is happening in your pipeline, which is important if things do not work as expected.

Although the most common form of running Cocoon is as a servlet, this is only one way of using the framework. In fact, it is only a very small part of Cocoon that is servlet specific. This part is only one of the interfaces Cocoon provides to the outside world. Another important interface that allows Cocoon to be used in different environments is the Command-Line Interface.

Using the Command-Line Interface

We previously mentioned one challenge when building web applications: the offline generation of web sites. You start a process, and this process builds the whole web site into a directory. You can then put it on your web server or on a CD.

This generated web site then does not need sophisticated software components on the server to run. It only needs a simple web server that can serve static files from the filesystem. All the real work is already done in the generation process.

That's where Cocoon's Command-Line Interface (CLI) comes into play. You can utilize it to generate a whole web site. This might seem like a great idea, but there are limitations. You can generate an offline version only if the content conforms to certain rules.

All the documents need to be static, which means that each time a document is requested, the content should be the same. For example, if you want to create a document that always displays the current stock account, this cannot be generated for offline viewing. If your documents are personalized, this is not possible with offline generation either.

So if you look at the challenges for current web applications, there seem to be only rare cases in which offline generation is really useful.

The Cocoon CLI can be used for other purposes as well. Invoking the CLI is nearly the same as requesting a document from Cocoon using the servlet engine. For example, it could be used to generate invoices offline as PDF files. Rather than having someone invoke a web page that generates bills, save them to disk, and then mail them to the customers, you can write a script that is invoked periodically to fulfill the same task using the CLI.

We have shown you how the Cocoon documentation is built using Cocoon itself. In addition, the Cocoon developers use the CLI to generate this documentation and put it on the Apache web server. All the offline generated images and HTML files must be put on the server, because it currently does not run a servlet engine where Cocoon could be installed. Listing 6.23 shows the Cocoon CLI.

Listing 6.23 **Cocoon's Command-Line Interface**

```
Usage: java org.apache.cocoon.Main [options] [targets]

Options:
        -h, --help
                print this message and exit
        -u, --logLevel <argument>
                choose the minimum log level for logging (DEBUG, INFO, WARN,
                 ERROR, FATAL_ERROR) for startup logging
        -c, --contextDir <argument>
                use given dir as context, this defaults to ./webapp
        -d, --destDir <argument>
                use given dir as destination
        -w, --workDir <argument>
                use given dir as working directory
        -r, --followLinks <argument>
                process pages linked from starting page or not (boolean
                argument is expected, default is true)
```

The CLI is implemented by a Java class (`org.apache.cocoon.Main`). So the CLI is started by starting this Java class. Because this class is contained in a JAR file, the command looks like this if you are inside the directory where all JAR files for Cocoon are stored: `java -jar cocoon.jar` followed by the options.

The most important option is `-c`. It defines where the Cocoon context directory can be found. This directory must contain cocoon.xconf. With the option `-u`, you set the log level. The destination directory (option `-d`) defines the location where the generated documents are stored. The work directory holds temporary files (option `-w`).

After the options, you define the documents you want to generate. Cocoon then processes these documents one after the other and saves them to the destination directory.

If `followLinks` is turned on (which is the default), Cocoon processes not only the documents you gave as input but also all documents referred by this one. So it crawls the whole web site. This is in fact used for the Cocoon Documentation System. Only the starting URL is specified (index.html). Because this document includes the navigation bar, all other documents are referenced by this document.

The crawling is done using views. The CLI first gets the link view of a document. This returns all the document's links and references (including images). Then the document is processed and saved to the destination directory. Afterwards, all collected links are processed, one after the other. Of course, the CLI makes sure that each document is processed only once and that no infinite recursion occurs.

After the CLI is finished, you have the whole web site in your destination directory. This includes all HTML documents, all images, all rendered SVG graphics, and so on. You could then copy this directory to a CD or to a web server for publishing.

For your first steps with Cocoon, the CLI might not be that important, but as you learn more and more about Cocoon, sooner or later you might need it. But you don't have to worry. Just start creating your own web site, documents, and so on and learn the Cocoon way. The following practical examples and tips will help you build more-advanced applications with Cocoon.

Practical Examples and Tips

This chapter has covered a lot of topics so far. Hopefully you have been able to use some of these new features to extend an application you already have built. We will now look at some examples and give you a few tips on getting the most out of Cocoon when you use it to build applications that other people might also use.

The two following examples help you understand the components and concepts presented so far. The first one is a small example showing you how to use the sql transformer to fetch data from a database. You might need to use the log transformer in this example if you have any problems connecting to the database. The second example is a bigger real-world example: the Cocoon Documentation System. This system uses nearly all the concepts explained so far.

We will then look at how you can make sure that your Cocoon application is set up to handle all the requests it might receive when you release it into a production environment.

A SQL Example

The following example requires a database that can be used from Java, so you need a JDBC driver. Instead of using your own database, you can use the included HSQLDB shipped with the Cocoon distribution. This database is completely written in Java and can be started automatically when Cocoon is run.

However, if you want to use your own database, you have to include a suitable driver. Put this driver class either in a JAR file in Cocoon's WEB-INF/lib directory or as a class file in the WEB-INF/classes directory. In order to make the driver available, you have to add it to the list of loaded classes in the web application deployment descriptor (web.xml), as shown in Listing 6.24. The parameter `load-class` gets a list of classes that are automatically loaded at startup.

Listing 6.24 **Adding Drivers**

```
<!--
      This parameter is used to list classes that should be loaded
      at initialization time of the servlet.
      Usually these classes are JDBC Drivers used
-->
    <init-param>
        <param-name>load-class</param-name>
        <param-value>
           <!-- For HSQLDB: -->
           org.hsqldb.jdbcDriver
           <!-- ODBC -->
           sun.jdbc.odbc.JdbcOdbcDriver
        </param-value>
    </init-param>
```

Next you have to add a connection to your database in cocoon.xconf. Listing 6.25 is an excerpt from cocoon.xconf that shows a custom connection called `personnel`.

Listing 6.25 **Configuring Data Sources**

```
<datasources>
    <jdbc name="personnel">
        <dburl>jdbc:hsqldb:hsql://localhost:9002</dburl>
        <user>sa</user>
        <password></password>
    </jdbc>
</datasources>
```

For this connection, you can define the URL to the database, the username, and the password. These three settings depend on which database you use. The user and password might be optional. If you want to use the HSQLDB, the values shown here should work right out of the box.

After you have defined your database connection, you can use it in the sql transformer by specifying the `use-connection` element for the transformer. Save the XML document shown in Listing 6.26 to the Cocoon context directory, and name it sqlexample.xml.

Listing 6.26 **A Simple SQL Example**

```
<document>
    <sql:execute-query xmlns:sql="http://apache.org/cocoon/SQL/2.0">
        <sql:use-connection>personnel</sql:use-connection>
        <sql:query>
            select id,name from department
        </sql:query>
    </sql:execute-query>
</document>
```

If you are using your own database, you might need to adjust the `select` statement. A stylesheet for the SQL data, transforming it to a simple HTML table, could look like Listing 6.27. Save this stylesheet, and name it sqlexample.xsl.

Listing 6.27 **A Simple SQL Stylesheet**

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sql="http://apache.org/cocoon/SQL/2.0">

<xsl:template match="document">
    <html><body><table>
        <xsl:apply-templates select="sql:rowset/sql:row"/>
    </table></body></html>
</xsl:template>

<xsl:template match="sql:row">
    <tr>
        <xsl:apply-templates/>
    </tr>
</xsl:template>

<xsl:template match="sql:id¦sql:name">
    <td>
        <xsl:value-of select="."/>
    </td>
</xsl:template>

</xsl:stylesheet>
```

Again, if you use a custom database and table, you might have to adjust the stylesheet to reflect different column names. The pipeline for this example is very simple. It is shown in Listing 6.28.

Listing 6.28 **A Sample SQL Pipeline**

```
<map:pipeline>
    <map:match pattern="sqldocument">
        <map:generate src="sqlexample.xml"/>
        <map:transform type="sql"/>
        <map:transform src="sqlexample.xsl"/>
        <map:serialize/>
    </map:match>
</map:pipeline>
```

Start your browser, and request `http://localhost:8080/cocoon/sqldocument`. You will get the XML data from the database displayed as an HTML table. If you are using your custom database and you face any problems, add the log transformer after the sql transformer to see what data is coming from your database.

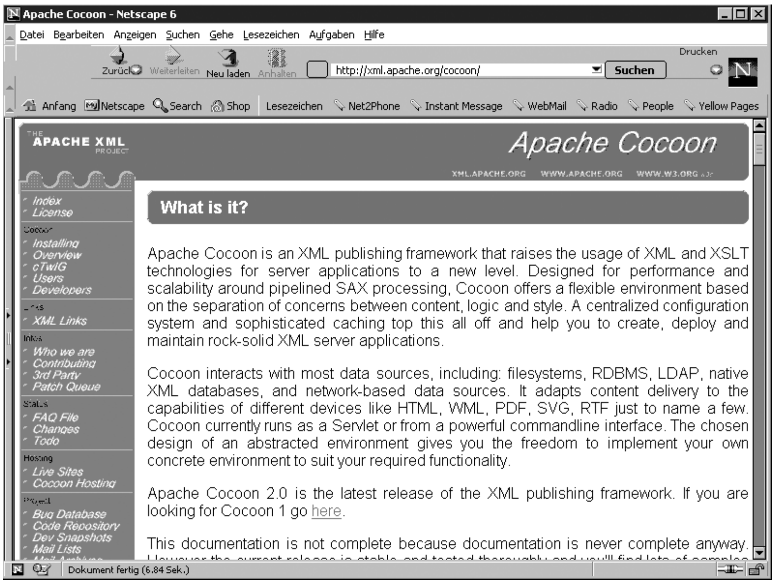
Using databases with Cocoon is very easy, as you can see from this example. To demonstrate more of the features introduced in this chapter, we will now look at a larger working example.

The Cocoon Documentation System

One of the best sample applications using many of the features we have described in this and the previous chapters is the Cocoon Documentation System. Because Cocoon itself is an XML publishing framework, the documentation is, of course, generated by Cocoon. Some of the features the documentation system uses include content aggregation, subsitemaps, the `cocoon` protocol, and image generation using SVG. All these features allow the documentation to be written in a fashion that separates the content from the layout.

Because this example is rather complex and uses many resources, we will examine only the basic idea behind this system. In addition, we will look at some excerpts from each of the files. If you're interested in seeing more than what's presented here, the whole system can be found inside the Cocoon distribution.

The Cocoon documentation (see Figure 6.9) is served by a subsitemap that is independent of the main sitemap. (You will find the subsitemap and all other resources in the documentation directory of your Cocoon context directory.)


Apache Cocoon - Netscape 6
Datei Bearbeiten Anzeigen Suchen Gehe Lesezeichen Aufgaben Hilfe
Zurück Weiterleiten Neu laden Anhalten
http://xml.apache.org/cocoon/
Suchen
Drucken
Anfang Netscape Search Shop Lesezeichen Net2Phone Instant Message WebMail Radio People Yellow Pages
THE APACHE XML PROJECT
Apache Cocoon
XML.APACHE.ORG WWW.APACHE.ORG WWW.W3.ORG
Index
License
Installing
Overview
cTwIG
Users
Developers
XML Links
Infos
Who we are
Contributing
3rd Party
Patch Queue
Status
FAQ File
Changes
Todo
Hosting
Live Sites
Cocoon Hosting
Bug Database
Code Repository
Dev Snapshots
Mail Lists
Dokument fertig (6.84 Sek.)

What is it?

Apache Cocoon is an XML publishing framework that raises the usage of XML and XSLT technologies for server applications to a new level. Designed for performance and scalability around pipelined SAX processing, Cocoon offers a flexible environment based on the separation of concerns between content, logic and style. A centralized configuration system and sophisticated caching top this all off and help you to create, deploy and maintain rock-solid XML server applications.

Cocoon interacts with most data sources, including: filesystems, RDBMS, LDAP, native XML databases, and network-based data sources. It adapts content delivery to the capabilities of different devices like HTML, WML, PDF, SVG, RTF just to name a few. Cocoon currently runs as a Servlet or from a powerful commandline interface. The chosen design of an abstracted environment gives you the freedom to implement your own concrete environment to suit your required functionality.

Apache Cocoon 2.0 is the latest release of the XML publishing framework. If you are looking for Cocoon 1 go here.

This documentation is not complete because documentation is never complete anyway.

Figure 6.9 The Cocoon documentation.

The documentation is currently available in HTML. Each HTML page consists of a static header, a navigation bar on the left side, and the content for the current document on the right.

The navigation bar is created by an index, which is called a *book* in Cocoon. The documentation is arranged in several hierarchically nested books. There is one main book, and it contains documents and subbooks. You can compare this to a directory structure such as your filesystem. A book is similar to a directory: It has a name, and it contains documents (files) or other books (directories).

As you might have already guessed, each HTML document is created using content aggregation and the `cocoon` protocol. Let's have a look at the sitemap entries, shown in Listing 6.29.

Listing 6.29 **An Excerpt from the Cocoon Documentation Sitemap**

```
<map:pipeline>

    <map:match pattern="*.html">
        <map:aggregate element="site">
            <map:part src="cocoon:/book-{1}.xml"/>
            <map:part src="cocoon:/body-{1}.xml"/>
        </map:aggregate>
        <map:transform src="stylesheets/site2xhtml.xsl">
            <map:parameter name="use-request-parameters" value="true"/>
            <map:parameter name="header" value="graphics/{1}-header.jpg"/>
        </map:transform>
        <map:serialize/>
    </map:match>

    <map:match pattern="**book-**.xml">
        <map:generate src="xdocs/{1}book.xml"/>
            <map:transform src="stylesheets/book2menu.xsl">
                <map:parameter name="use-request-parameters" value="true"/>
                <map:parameter name="resource" value="{2}.html"/>
            </map:transform>
            <map:serialize type="xml"/>
        </map:match>

        <map:match pattern="body-**.xml">
            <map:generate src="xdocs/{1}.xml"/>
            <map:transform src="stylesheets/document2html.xsl"/>
            <map:serialize/>
        </map:match>

</map:pipeline>
```

The document names available from these pipelines do not follow our recommendation: They use explicit endings such as .xml and .html. The HTML document is aggregated by two parts—a part called `book`, and a part called `body`. The book part reads the current book and creates the navigation bar from it. This navigation bar is transformed by a stylesheet to partial XHTML.

The body part reads the real content from the XML document and transforms it into partial XHTML as well. The main pipeline for the document aggregates these two parts and combines the XHTML fragments using a stylesheet. It also adds the constant header.

The navigation bar and title displayed in the document's header are actually images. These images are rendered using SVG. We left out the pipelines for the images, but they are specified in the installed Cocoon application. Inside the Cocoon context directory is a directory called documentation. This directory contains a subsitemap named sitemap.xmap that contains all pipelines for the whole documentation system.

This example of a real application shows how a web site can be built very easily with Cocoon. By using content aggregation, you separate the different parts of one document and can maintain them more easily. Just take your time and have a look at this application and how it works. It will help you understand the concepts you have learned so far. You will also get a look behind the scenes of Cocoon's documentation system. Of course, one of the most important features of any Internet application, such as the documentation system or a portal built with Cocoon, is how fast the required information is returned. After all, no one wants to wait around for minutes until the browser displays the requested document. Cocoon provides two methods of speeding up the application: pipeline caching and component pooling.

The Cocoon Caching Mechanism

As you have seen, Cocoon generates documents using pipelines that contain a variety of components. You have seen that each time a request reaches a pipeline, the required document is generated and returned to the calling application. Using Cocoon's caching mechanism, you can control whether the document is actually generated or whether it can be returned from a cache. This speeds up the time it takes to return the document, because the pipeline does not have to be processed completely. Cocoon's caching algorithm is very flexible, but fortunately it is also very easy to handle. Let's start with a description of the caching algorithm.

Cocoon generates a stream pipeline for each request. This stream pipeline either is a reader or consists of an event pipeline and a serializer. The event pipeline in turn is assembled by a generator and the used transformers (if any).

Cocoon's caching algorithm can cache the result of a stream pipeline and/or an event pipeline. The caching for such a pipeline is turned on or off in cocoon.xconf (see Listing 6.30). Because everything in Cocoon is implemented using Avalon components, you simply specify which implementation for an event or stream pipeline should be used: the caching or the noncaching one. You will learn more about these components when we explain Cocoon from the developer perspective in Chapter 8.

Listing 6.30 **Turning on Caching in cocoon.xconf**

```
<event-pipeline class=
➥"org.apache.cocoon.components.pipeline.CachingEventPipeline"/>
<stream-pipeline class=
➥"org.apache.cocoon.components.pipeline.CachingStreamPipeline"/>
```

These lines turn on caching for both pipelines. The code shown in Listing 6.31 turns it off. Of course, you can mix it and turn on caching for event pipelines but not for stream pipelines. If you want to change your setting, locate the lines for `event-pipeline` and `stream-pipeline` in your cocoon.xconf and change the `class` attribute.

Listing 6.31 **Turning off Caching**

```
<event-pipeline class=
➥"org.apache.cocoon.components.pipeline.NonCachingEventPipeline"/>
<stream-pipeline class=
➥"org.apache.cocoon.components.pipeline.NonCachingStreamPipeline"/>
```

But what does it mean if caching is turned on? The following explanation is simplified for the user perspective. We will look at the full power of the caching algorithm in Chapter 8.

But for now, let's start with the stream pipelines. The result of a stream pipeline, for example, can be cached if it is a reader, which can cache. So we can redefine the question: When can a reader cache?

A reader (and this is also true for the other sitemap components, as you will soon see) can cache if it can detect that the content has changed since it was last read. For example, the resource reader reads a file. It can detect whether the file has changed by looking at what time the file was last changed.

So the first time the resource reader reads a document, the caching algorithm stores this document, along with the current time. The next time this document is requested, the caching algorithm provides this time to the reader, which simply checks whether the cached content is still valid. If it is, the cache serves the document. If it is not valid, the cached content is discarded, the reader reads the file again, and the cache stores this along with the current time.

But there are cases in which the reader cannot detect content changes, such as if it gets the read file via HTTP or any other connection. In this case, the reader can't support caching, so nothing is cached. This means that even though Cocoon provides a means of caching pipelines, it is still dependent on the data source to provide a means of determining whether the content has changed since it was last accessed.

If the stream pipeline consists of an event pipeline and a serializer, both parts must support caching. Most serializers in Cocoon support caching, because they are only dependent on the XML they receive from the event pipeline.

The question of whether an event pipeline can be cached is more complex, because the pipeline consists of several components. It is cacheable only if all the components are themselves cacheable. In the event pipeline, the caching algorithm asks each component if it supports caching, starting with the generator. For each component that supports it, a unique key is generated. Then the next pipeline component is queried. This process continues until either all components are queried or one component is not cacheable.

The keys of all cacheable components are chained, and together they build the cache key. The request is processed, and the document is built. The cache stores the result of the last component, indicating cacheability. The next time this document is requested, the key is built, and the cached content is fetched from the cache.

Next, the cache asks all components of the event pipeline if their input has changed since the time the content was cached. For example, the generator checks this by looking at the last modification date of the XML document, the xslt transformer checks the date of the stylesheet, and so on. Only if all state that the content is still valid is it used from the cache. Otherwise, the document is generated from scratch. So the event pipeline tries to cache as much of the XML processing pipeline as possible.

Caching the pipeline results and being able to return them as fast as possible is perhaps the key factor to whether an Internet application built with Cocoon will be successful and whether people will like using it. Cocoon's built-in caching already provides a powerful mechanism for doing this and should be used whenever possible. Another important factor in any component-based system is the performance at which new components are created when they are needed.

Pooling Your Components

Nearly everything inside Cocoon is an Avalon component. Without going into too much detail about the Avalon component model and the life cycle of components, we'll explain how you can fine-tune your application in this area.

For each request received by Cocoon, a lot of Avalon components are generated—one event pipeline, one stream pipeline, one generator, one or more transformers, and a serializer. (In fact, there are more, but these will do for the moment.)

If several documents are requested at the same time, this set of components is created for each request. For example, if 50 documents are requested simultaneously, you end up with 50 event pipelines, 50 stream pipelines, 50 generators, and so on.

One of the most time-consuming operations in Java is the creation and destruction of new objects. Therefore, the Avalon component model supports the pooling of objects. This means that a component is created once, locked when used inside a request processing, and released for further use after the request is processed. It is not destroyed and can be reused for the next request.

If only one request at a time is processed, such a pooled component is created once, locked for this request, used for this request, and released afterwards. When the next request arrives, the same process starts again.

If more than one request is processed at the same time, a pooled component must be created for each request. If 50 requests arrive simultaneously, 50 components must be created. If they all can be pooled, the pool grows to 50 components. At first glance, this seems desirable, but imagine that one day 1000 requests are processed simultaneously. You end up having 1000 components in your pool, although the average of simultaneous requests is less.

In order to adjust your application to the load you might have, you can control the pooling of the Avalon components. You can define how many components are to be stored inside the pool by specifying a minimum and maximum number, as well as how the pool should grow if no free component is available from the pool. If your pool reaches the maximum, but there are more requests to serve, Avalon creates new components to process the request, but these components are discarded afterwards and are not added to the pool.

The configuration of this pooling is on a per-component basis. So you set the values separately for each component—for the stream pipeline, for the event pipeline, for the file generator, and so on. Listing 6.32 shows a sample pooling configuration.

Listing 6.32 **An Example of a Pooling Configuration**

```
<stream-pipeline class=
➥"org.apache.cocoon.components.pipeline.CachingStreamPipeline"
                 pool-max="32" pool-min="16" pool-grow="4"/>

<generator name="file" src="org.apache.cocoon.generation.FileGenerator"
                 pool-max="64" pool-min="16" pool-grow="4"/>
```

In Listing 6.32, you see the configuration for the stream pipeline, which is done in cocoon.xconf, and for the file generator, taken from the sitemap. Remember that both the sitemap and cocoon.xconf contain components that are based on Avalon and therefore can be pooled.

Both configurations are similar in that they use three special attributes. `pool-min` defines the minimum number of components in the pool. When the pool is instantiated, this number of components is created at startup. `pool-max` defines the maximum number of components to hold in the pool. `pool-grow` gives the number by which the pool increases each time no free component is available.

If you set the log level to `DEBUG`, you can see if your pools are too small by searching for a message containing the phrase "decommissioning instance of." This message is output each time a poolable instance is created when the pool has reached maximum capacity. The component's class name follows the phrase, so it is possible to adjust the setting for exactly this component.

With the tips on caching and component pooling, we have covered the two most important ways to make a Cocoon application as fast as possible. These features are provided by Cocoon and can be used in different application scenarios. Depending on the type of application being built, other factors can influence the application's performance. We will cover some further aspects when we talk about different types of applications in Chapter 11, "Designing Cocoon Applications."

Wrapping Up the User Perspective

We have reached the end of our tour through Cocoon from the user perspective. All the Cocoon features we have discussed up to this point are available without your having to write any Java code to use them. You learned about the additional ways to configure Cocoon and, in particular, which configuration parameters exist to allow a Cocoon-based application to return the requested documents as quickly as possible.

Apart from the more common components, such as transformers and generators, Cocoon also provides additional components such as action-sets, and it allows different pipelines to be combined using content aggregation. We completed the explanation of the different sitemap sections, especially views and sitemap resources. We also looked at some examples, such as connecting Cocoon to a database.

Building applications using these concepts can get quite complicated, but luckily Cocoon provides ways of staying on top of what you are doing. Splitting the separate areas of an application into different subsitemaps is one way of making sure the solution is modular. Using the log transformer inside a pipeline allows potential errors to be found quickly and also shows you how the different components can be plugged in to a pipeline to extend the functionality.

We realize that this is a lot of information to take in. We suggest that you try and adapt the examples we have presented to do something different. Perhaps you could integrate a different data source into your application or provide a different output format for your data. Play around with the components and see what types of pipelines you can build. Add the log transformer to a pipeline and look at what goes on between the different components.

You might also find some ideas for your own applications in the next chapter, where you will expand the news portal you built in the last chapter and add some of the things you have just learned.