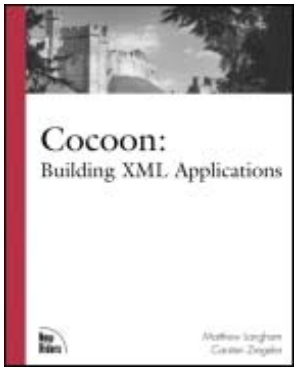


Cocoon: Building XML Applications

Matthew Langham
Carsten Ziegeler
Publisher: New Riders Publishing
First Edition July 19, 2002
ISBN: 0-7357-1235-2, 504 pages

Cocoon: Building XML Applications is a comprehensive hands-on guide to the Apache open source project, Cocoon. Cocoon is an XML publishing platform already being used by companies such as Hewlett Packard and institutions such as NASA to build their next generation of Internet architectures. Developers, administrators and managers will find this detailed resource an invaluable tool whether you are looking for introductory information on XML/XSL technologies, starting out with the open source platform or seeking a guide to extending Cocoon with additional components.

This book combines the knowledge of a key Cocoon developer with the experience of someone who has been building and writing about Internet applications since the early 1990's. It begins by explaining the advantages of XML, then guides the reader through the process of setting up Cocoon and details the architecture from a user's as well as a developer's point of view. The varied examples, from the typical Hello World program to a complete news portal also help to provide an insight into applying open source software to "real world" problems. A detailed reference section documents the various components available in Cocoon and provides the developer with the necessary API documentation.

Table of Content

About the Authors

Matthew Langham was born in England but has lived in Germany since 1976. He has worked in the IT business since the mid-1980s. He wrote his first book on the Internet in 1993 and has since published several articles on the Net and related themes. He currently leads the open-source group at S&N AG, a software company in Paderborn, Germany.

Carsten Ziegeler is the chief architect of the open-source compe-tence center at S&N AG, Paderborn, Germany. His main focus is on web application design and object-oriented component development. He has participated in several open-source projects and is actively involved in various Apache communities. In 2001, he took over the role of release manager for the Apache Cocoon project. He has been a committer on the project since 2000 and played a major role in designing the current architecture.

About the Technical Reviewers

These reviewers contributed their considerable hands-on expertise to the entire development process for *Cocoon: Building XML Applications*. As this book was being written, these dedicated professionals reviewed all the material for technical content, organization, and flow. Their feedback was critical to ensuring that this book fits our readers' needs for the highest-quality technical information.

Marcus Crafter is from Australia and currently works as a software engineer for a Melbourne-based company, ManageSoft Corporation. He has worked extensively with Internet technologies since 1996. He lives in Frankfurt, Germany, where he has been actively involved in various open-source/free software projects, including Apache Cocoon, for the past three years.

Torsten Curdt is the CTO of dff internet & medien GmbH, Göttingen, Germany. He started out as a programmer in the 1980s and has been active in the IT business since the early 1990s. As dff's main software architect, he has been around since Cocoon version 1.7. He became a committer to the Cocoon project in 2001 and is involved in several other open-source software projects.

Acknowledgments

Writing a book is just like working on a software project—it's teamwork. And we had a great one. So here are the people we would like to thank:

Matthew would like to thank Claudia, Christopher, Victoria, and Nicolas for allowing him to write the book during "family" time. He would also like to thank Frank and Holger for getting him started with computers back in the (good old) VIC 20 days.

Carsten would like to thank his wife, Andrea, for all the support and good words in the last few months; his parents and parents-in-law for all the help on the new home, which gave him a lot of time for this book; and his brother, Jörg, who influenced Carsten's career by buying a Commodore C64 nearly 20 years ago. Special thanks go to Paul Russell, who started the vote on accepting Carsten as a Cocoon committer, to Giacomo Pati and Davanum Srinivas for their help during the first steps in Cocoonland, and to the whole Cocoon community for the interesting "work" every day.

Carsten and Matthew would like to thank Stephanie, Fred, Torsten, Marcus, and all those involved at New Riders who made this book possible. Thanks also to Bert, Sylvain, and Andrew for providing last-minute suggestions and corrections. We are very grateful to Klaus, Josef, and Uwe and all our colleagues at S&N for allowing us to work on open source and still get paid.

And last but not least, we would both like to thank Stefano for taking that Xmas holiday in the Alps back in 1998.

Tell Us What You Think

As the reader of this book, you are the most important critic and commentator. We value your opinion, and we want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an Executive Editor for the Web Development team at New Riders Publishing, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book, as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax:	317-581-4663
Email:	stephanie.wall@newriders.com
Mail:	Stephanie Wall Executive Editor New Riders Publishing 201 West 103rd Street Indianapolis, IN 46290 USA

Introduction

Welcome to *Cocoon: Building XML Applications*. We decided to write this book to provide additional documentation on the Cocoon open-source project. However, we also wanted to embed the Cocoon-specific information in a more-general XML application context. Therefore, we have included information that we hope is helpful for anyone starting out with XML.

Who Should Read This Book

This book was written for a wide audience. If you are currently wondering whether your application architecture should move to XML, this book provides some answers. Readers who have already decided on an XML-based architecture will find information on open-source software that will help them build that architecture. The main audience is obviously readers who are interested in the open-source XML publishing platform Cocoon.

As for the skill set you need in order to read this book, it is written for both the guru-developer and the site administrator. If you are more of a manager, you will also find interesting information that will help you decide which technology to employ when building XML applications.

Who This Book Is Not For

If you are totally into Microsoft solutions, perhaps this is not exactly the right book for you. Although you will still find helpful information on XML in general, most of this book centers around open-source software.

Overview

This book begins with an introduction to Internet applications in general and describes how those applications have been built over the years. It also details the drawbacks of HTML as a base for modern application architectures and lists the many challenges that must be met by new Internet-based solutions.

We continue by introducing XML and XML-related technologies as a way to build modern application architectures. The advantages of using XML are listed, and we introduce available software components. Using a flexible XML-based framework, such as Cocoon, allows applications to be built quickly and cost-effectively.

We then explain how to install Cocoon and provide a guide for setting up a Cocoon-based system. All the needed software is contained on the companion CD.

After you have set up Cocoon, it is time to put some of the basic concepts and components to work. The first “hands-on” chapter contains different examples that show

you how Cocoon can be used to build various types of XML applications. All the detailed solutions can be built using the components available in the Cocoon distribution and without any Java know-how.

Throughout this book, you will build more-advanced solutions in separate chapters. After each section of the book, you will use what you have learned to build different versions of a news portal. Each version expands on the previous one and introduces new concepts.

After you build the first version of the news portal, we go into more detail on the Cocoon architecture, but we still do this from a user perspective. The new concepts are then used to enhance the portal you developed.

The next two chapters cover Cocoon from a developer perspective. They require a working knowledge of Java in order for you to understand Cocoon's inner workings and how to design new components that can be used to extend the platform.

The chapter that covers the advanced version of the news portal looks at how Cocoon provides different ways of reaching the same goal and provides some tips on when to use which technology. This theme is expanded in the following chapter, where we take a step back from the technical side and provide some insight into designing applications based on Cocoon.

The final chapter contains an outlook on Cocoon's future and describes some of the developments that did not make their way into the release of Cocoon we used when writing this book.

The appendixes round out the book and provide additional information such as API and component documentation, links to more information on the web, and a description of the companion CD.

Conventions Used in This Book

This book follows a few typographical conventions:

- A new term appears in *italic* when it is introduced.
- Program text, functions, variables, and other "computer language" are set in a fixed, `monospace` font.
- At the beginning of a line of code indicates it is part of the line above it.

Chapter 1. An Introduction to Internet Applications

Apart from being something you would normally associate with butterflies or a Hollywood movie, Cocoon is also the name of an open-source project. It is an XML/XSL-based framework, written in Java, that enables you to build dynamic Internet applications, such as the ones that serve up your favorite web pages or give you your account balance when you access your bank over the Internet or via your mobile phone.

These applications typically lie between the client you are using, such as an Internet browser, and the systems that provide the data. So an Internet application built with Cocoon to serve up your account information runs on a system that your browser contacts and then connects to, say, a mainframe to obtain the necessary details.

Although there are already Internet applications that do all these things, traditional systems are still unable to solve many problems in an effective manner. Cocoon, because of its architecture and the technologies it incorporates, provides a better solution for realizing Internet applications, especially when a high degree of flexibility (both in publishing and systems integration) is necessary.

The first questions people often ask when confronted with new software products are “Why?” and “How?” Why do I need yet another product, and how can I use it to solve my problems? In order to answer the question of why Cocoon is needed, we must look at how Internet applications are written today, how they were written in the past, and what problems modern application architectures still need to solve.

This chapter discusses the history of Internet applications and the key areas that any Internet solution needs to resolve. We will then introduce you to the world of Cocoon and show how you can use it to build applications that can range in shape and size from a simple picture gallery to a full-blown personalized news portal. We will also show you how to extend Cocoon to meet your specific needs.

But before we go on to what you might be able to do in the future, we need to take a look at the past.

A Brief History of Internet Applications

Even though the popular Internet is still relatively young, dating back to the early 1990s, the way Internet applications are written has changed considerably over the past decade.

During the time we have been writing *about* the Internet and writing applications *for* the Internet, we have seen it change from an exotic underground "thing" to being a part of our everyday lives. Internet applications have grown up from being just collections of static pages and now offer dynamic and personalized solutions. Internet access is no longer confined to simple browsers but is now available via your phone or car radio.

Currently, our main focus is on software for financial institutions in Germany. Financial institutions are interesting companies to write software for, because they are very quick to latch onto new technologies, and they have a diverse base of both software and hardware to write programs for. They also offer one of the oldest online applications around: online banking.

Using this application, we will show you how the development of this type of solution has changed over the years. In this chapter, our historical journey starts in 1995, the year before we wrote our first Internet banking solution. (We will take you back even further in time in Chapter 2, "Building the Machine Web with XML.")

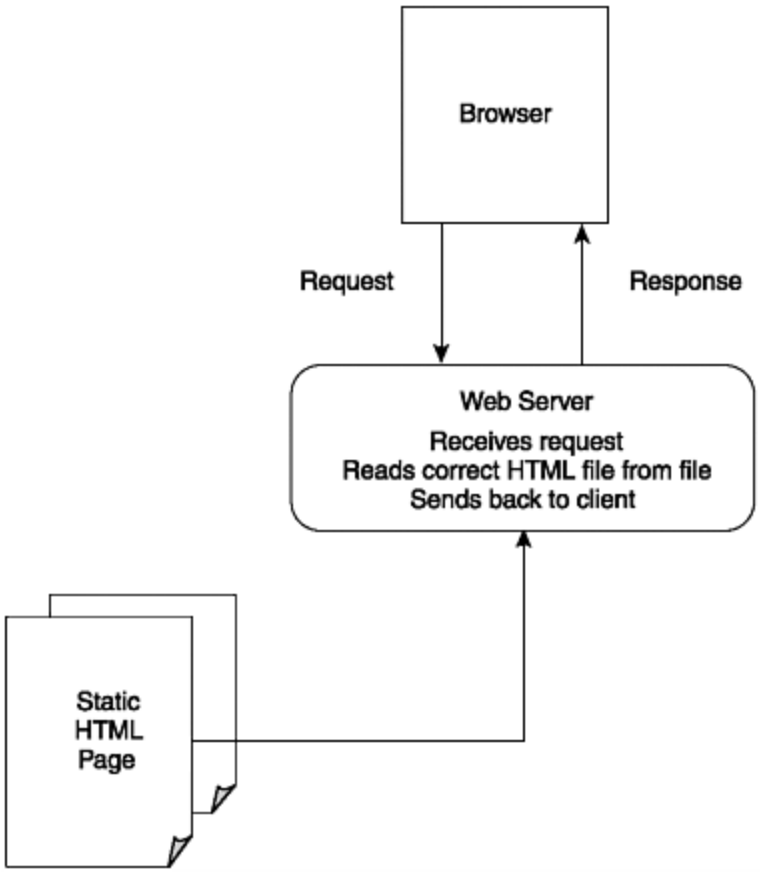
Static Pages

In February of 1995, the most popular server software on the web was the public-domain HTTP daemon developed by Rob McCool at the National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign. NCSA also developed Mosaic, one of the first web browsers.

The first Internet applications to go public on these platforms were made up of static HTML pages. These pages were used to publish unchanging information over the Internet. Users who could code HTML and knew exactly which data they wanted to present authored these HTML pages. If the data changed, the HTML pages had to be completely reauthored and deployed to the web server.

The first generation of web servers (such as Microsoft's Internet Information Server 1.0) could serve these pages over the Net, as shown in Figure 1.1, and provided a limited means of integrating data (such as via a specific database extension called Internet Database Connection). The first-generation web servers also provided a way of passing requests on to external programs via a standardized interface called CGI. (We will look at CGI in the next section.)

Figure 1.1. A web server delivers static HTML.



Most of the web applications we wrote during this period were more concerned with showing potential customers what this "web thing" was, so we concentrated on static HTML with a set layout. Basically, we learned HTML as we went along, so the simple pages we built contained just the basic tags and the colors that looked best on the particular browser we were using. Fortunately for us, at that time very few programs were able to present these HTML pages, so no one was yet worried about being able to publish flexibly for different applications and devices.

Being able to see the pages that had just been deployed to a web server from a location that was miles away was a completely new experience for most people. It caused quite a sensation when it became possible to view information that someone had made available on the other side of the globe.

Even though at that time mostly static information was being published (at least we were), it quickly became clear to us that applications such as the online banking solution German banks already had (of which you will read more in Chapter 2) would eventually migrate to the Internet world.

At the beginning of 1996, our company decided to attend one of the first online trade fairs in Hamburg, Germany. Because we had a history of writing software for banks and had also already worked on online banking projects, it seemed like a good idea to present an Internet-based version of this application and see what people's reactions would be.

The first version of the Internet banking solution we built fit entirely on one floppy disk (and this included the Netscape browser). Of course, this simple application couldn't really do anything. It was more of a presentation in HTML, demonstrat-ing how this sort of application might look and feel in the future.

Working in the financial industry means you get to play with lots of interesting hardware, such as Automatic Teller Machines (ATMs). In 1996, some ATMs were already completely PC-based and ran operating systems such as Windows NT. This meant that they could run a browser such as Netscape just as well as a normal PC or workstation.

So, in Hamburg, we presented our online banking solution running on a desktop PC and an ATM at the same time. This might seem simple today, but in 1996 it was one of the first times an application had been used to present the same information over different channels (a PC and an ATM).

Today, being able to present the same information on different devices at the same time is called *multichannel*. For corporations, such as banks, it is becoming increasingly important to provide applications that can be accessed by various devices and applications. For such companies, each device or application is considered a separate (sales) channel.

Multichannel solutions, such as online banking via browser or mobile phone, are now commonplace—and this is one of Cocoon's key strengths. The multichannel concept that Cocoon enables is far more flexible than what we could do in 1996, because it allows the same information to be presented differently for each separate channel. This means that you can publish identical data to, say, a PC and a mobile phone in different formats. Cocoon also allows for such things as the time of day and even the weather to be taken into consideration when the pages are generated.

Static HTML pages are fine for publishing information that doesn't change too often, but they are no good for dynamic information or for use in applications.

For the Internet to become a success as an infrastructure for applications, a more dynamic way of publishing HTML pages was needed, and the available Internet servers needed to support this by offering the necessary components and interfaces.

Programmable Components

The first role web servers played was that of a file server. The browser requested a particular file, and the web server read that file from the hard drive and returned it to the client. In order to allow the HTML file to be changed or generated dynamically before being returned, the web server had to provide some way of hooking up to the serving process.

One way of doing this was by passing the request to an external program by way of a defined interface. The Common Gateway Interface (CGI) was defined for this purpose,

and programs supporting this interface could be written in a variety of languages (such as Perl or C). CGI has been around almost as long as web servers themselves. It originally was supported by version 1.0 of the NCSA HTTPD server in 1994. The code from the NCSA server went on to become the basis of the now-popular Apache web server.

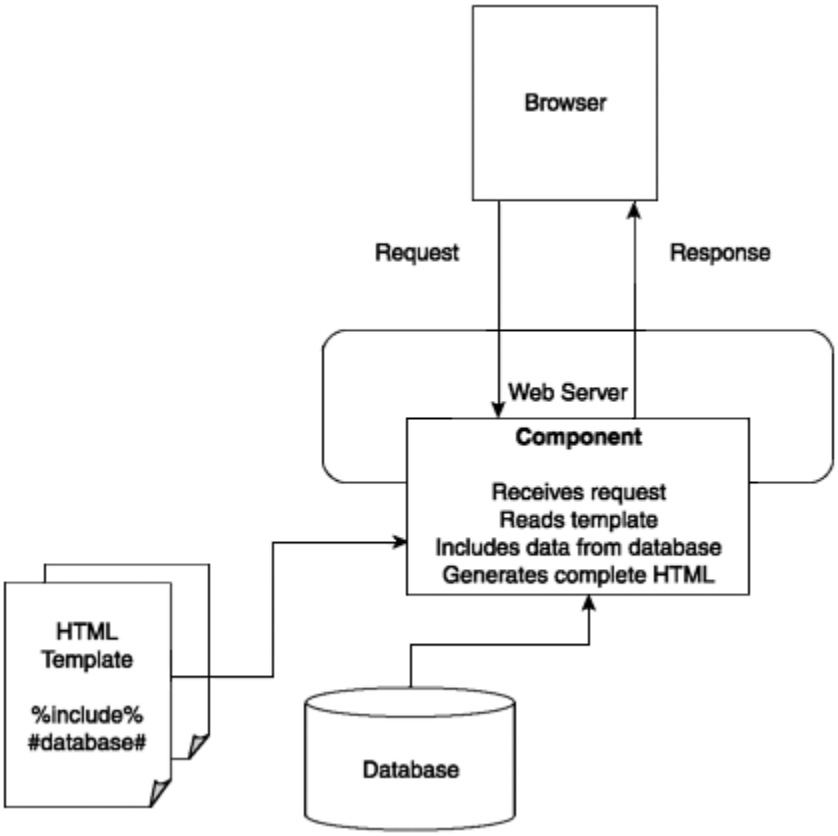
Another alternative was the provision of a defined interface inside the web server for programmable components (as opposed to applications). Interfaces such as Microsoft's ISAPI and Netscape's NSAPI allowed you to plug your own components (such as dynamic link libraries, in the Windows world) into the web server.

Because the web server passed the incoming request to either the external application or the component, it was then possible to generate the resulting HTML page dynamically. Because the components contained logic and were able to interface with existing data or applications, this was the first step toward building truly dynamic web solutions.

Instead of just being able to serve static pages, the web server now could return pages that were generated on-the-fly. Each component was then written for a specific purpose and could handle a defined set of requests. So, for example, a module written to provide HTML pages containing the current weather situation would receive all the incoming requests for the weather. Before generating the HTML page, the component would first access the current weather data from a database and then generate the resulting page to include exactly that data.

When we decided to write the components to provide an Internet banking solution, we defined our own templates for each page we wanted to return. Instead of the whole HTML page being dynamically generated, the component would read in the correct template and fill in the missing pieces, adding the data it had obtained from the external banking system to build the page. Figure 1.2 shows how this works.

Figure 1.2. An integrated component generates HTML from a template.



Using the architecture shown in Figure 1.2, we designed one of the first Internet banking solutions in Germany and installed it in 1997. That solution was able to integrate financial data obtained from other sources into HTML templates. The end result was a complete HTML page that contained your bank account information and provided your banking transactions to date.

The solution started out running on Microsoft's IIS 2.0. We were really pleased that we could generate the HTML output using our own template language. Our HTML generator even allowed you to script inside those templates, providing simple if-then-else combinations. At runtime, our HTML generator interpreted the scripting commands inside the templates and allowed the HTML to be built, dependent on some data obtained for the customer. A simple example was to format the account balance in red if the value was negative.

Because at that time none of the web servers provided a standardized way of writing templates and scripts, we wrote our own little scripting language. Of course, the disadvantage was that only our component could understand the language, and that component ran on only a specific vendor's web server. But all in all, it worked quite well. The solution we wrote in 1997 was not replaced until the middle of 2001, even though other alternatives of writing Internet applications appeared soon after we installed our first version.

Scripting Languages

About two days after we installed the solution in 1997, Microsoft released the first version of its Active Server Pages (ASP) technology. The world of building Internet applications changed abruptly.

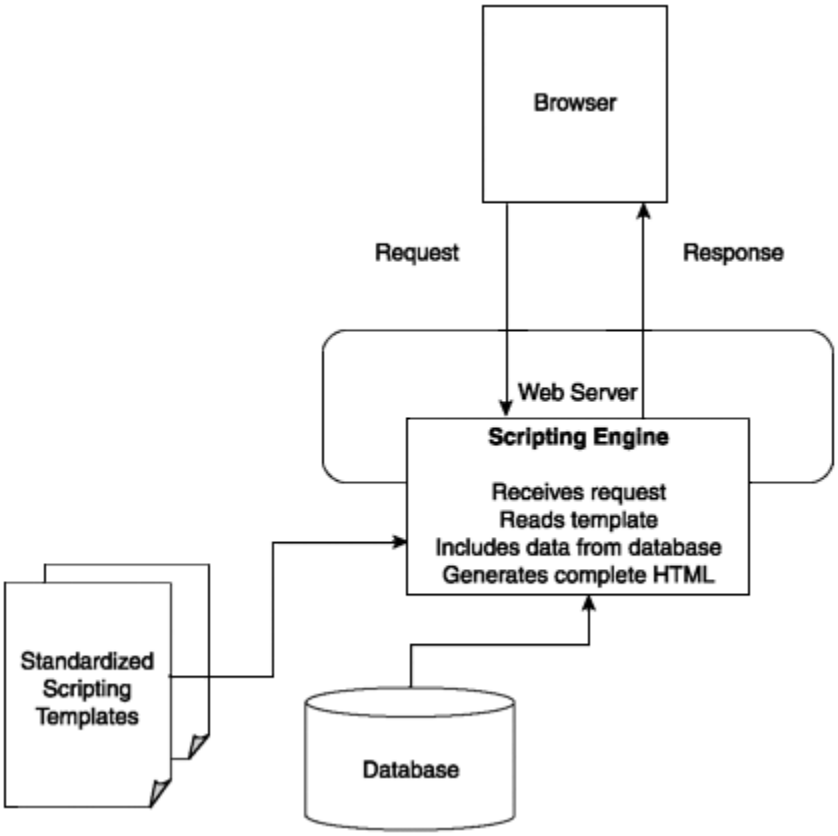
In addition to allowing programmable components to be integrated, the servers began providing for scripting languages. Scripting languages such as Microsoft's ASP and the Java-based Java Server Pages (JSP) were developed to allow HTML to be generated on-the-fly as opposed to being served from a static file.

These scripting languages became very popular, and many of today's Internet applications are written using one of them.

These languages allow you to author your page, including scripting commands that control how the resulting HTML page is built. Because these scripting languages are edited in normal text files and do not need to be compiled by the author, they have opened up the world of web applications to people who would not normally write software.

Writing a script using a language such as JSP means that any web server that supports JSP, either itself or by way of an additional component, is able to understand that script and process it to build the resulting HTML page. Figure 1.3 shows how scripting can be used inside an Internet application architecture.

Figure 1.3. Standardized scripting allows dynamic HTML generation.



This means that the templates can be shared between servers running the same scripting engine. Internet applications written in this way basically consist of a library of scripts. The web server maps each request to a particular script at runtime.

In itself, just being able to control how the page is built is not enough to be able to build dynamic applications that incorporate data from external systems. Therefore, each of the different scripting languages provides some way of accessing such things as a database or of keeping track of who is currently accessing the site and using the online banking application.

By the end of 1997, it became clear to us that any further versions of our online banking solution needed to be based on one of these scripting technologies. Therefore, in 1998 we started to design and build the next version of our Internet banking platform—using ASP.

That solution allowed the dynamic creation of HTML pages using scripting templates. The customer's account information was integrated into the HTML pages by way of specific components that accessed the mainframe and returned the data needed. The way these components were written allowed them to be easily integrated into the scripting process.

On the whole, using ASP made it easier to write the new version of the online banking application, but unfortunately, other things were going on that gave us quite a few headaches.

Starting in late 1996 and continuing until well into 1999, Microsoft and Netscape fought the “browser wars.” This meant that new versions of both programs were released nearly every week (at least, it seemed like it). It didn’t really affect us until we actually had to write applications to support the different versions. When we started planning the online banking solution for our customer, the Microsoft browser version we were supposed to support was Internet Explorer (IE) 3.0. During the time the application was written, new versions appeared and had to be supported inside our solution. When the application went into production, the newest IE version was 5.0.

Our headaches were caused by the fact that each browser vendor had (and perhaps still has) a unique view of what correct HTML should look like. Even different versions of the same browser did not render HTML in the same way. The first versions of our application could not be displayed on some of the available browsers because of these differences.

There was only one way to get around this problem, and that was for the scripting inside the ASP pages to allow for these different versions. Luckily, browsers send a piece of information to the server, telling who they actually are. This information includes the name and version number. This information can then be interpreted by the server application.

However, this also meant that our ASP pages became riddled with browser-specific commands. Depending on the browser type or version, different HTML fragments had to be generated into the finished page. This caused the whole solution to become very hard to maintain and extend. Listing 1.1 shows what the ASP then looked like. In this case, a specific function was added to the generated page if Netscape version 4 was being used.

Listing 1.1 Sample ASP Code

```
<% if Session("browsername") = "Netscape" and
       Left(Session("browserversion"), 1) = "4" then %>
    function button_Print() {
          window.print();
        }
<% end if %>
```

Every time a new browser version appeared, it had to be incorporated into the scripts to be sure it received the HTML it could render. This approach works, but it is not easy to extend and maintain. It is also not very flexible, because each time you want to change something for a particular browser, you have to make sure there areno side effects.

Flexible Publishing

The scripting approach works well when you are serving information in only one format to one particular device or application. When you decide to serve the same data into a different format, such as Wireless Markup Language (WML) for mobile phones, then you are faced with the problem of rewriting the application to provide for this new format.

When the customer who was running our first Internet banking application decided to support mobile phones, they had a completely new application developed. This application was written with the specific goal of serving data, scraped out of the generated HTML pages into the WML format required by phones. Indeed, many content providers did exactly this when using mobile phones to surf the Net started becoming popular, especially in Europe and Japan. Because of the way applications were being developed—and, for the most part, still are today—each different format is thought of as requiring a separate application.

Another drawback—and perhaps even worse—is the fact that the scripting approach does not help you truly separate the layout design from the data you want to display. The same people responsible for laying out the graphical design of the HTML page are forced to know about the data and how to access it. Also, because of the way script pages are integrated into the software that hosts them, the same person has to worry about the architecture of the complete application.

As we saw from the way we were building applications based on ASP, the same person who authored the code to access the data from inside those pages was designing the pages. Apart from how the pages were filling up with the specific commands we mentioned earlier, it became increasingly difficult to maintain a common look and feel for the whole application, because the different authors were changing the look of the pages they worked on.

Some scripting languages support the use of libraries of reusable code—and this does help when large applications are built. However, when we looked at some of these libraries being used by ourselves and also by our customers, we found that not only was code being put into them, but the look and feel was contained in the code libraries as well. So, what in fact was happening was that portions of the complete page were being stored, making it just as hard to adapt the code for new formats.

Another problem is maintaining sites written in scripting languages. Imagine a mission-critical application that contains perhaps 200 separate script pages. As soon as it is in production, the application doesn't suddenly stop existing. It is constantly extended and bug-fixed. New versions are released as new functions in the application become available. How do you manage to rescript the application for a new format such as WML without affecting the stability you might have already achieved?

This is exactly the situation in which we found ourselves in 1999.

During that summer, our customer told us that they wanted to be the first bank in Germany to support mobile phones and Personal Digital Assistants (PDAs) using WML. Even though WML was written for mobile phones, the PDAs adopted this format as well because it was easier for the software to display than HTML. This made us take a step back from the way we had designed Internet solutions up to that point. At that time, supporting the various browsers in the different types of mobile phones was far harder than supporting the leading PC browsers.

We took a close look at this upcoming “standard” of WML and the “standard” devices that were slowly emerging. After we tested against a couple of the available phones and PDAs, it quickly became clear that the situation was far worse than we had imagined.

At that time, very few phones supported WML and the underlying Wireless Access Protocol (WAP) technology, but many mobile-phone vendors had announced their support in upcoming versions. In addition to those on the market, we obtained a couple of preproduction devices and tested those against the WML standard.

Because WAP was a hyped technology and everybody was rushing to jump on the bandwagon, there were large differences in how the WML format was displayed. Some phones displayed input fields on the same line as their label; other phones broke up the flow of a form by putting the label and input field on two separate lines.

The hype went so far that one mobile-phone vendor released its new model with a version of WAP that actually was never supported by the phone companies. Add to all this the difference in screen layout and size between mobile phones and PDAs such as the Palm, and we quickly decided that in no way did we want to fit all the WML code to handle all this into the completed, tested, and running ASP pages.

At roughly the same time (having had our interest ignited after visiting XML talks at the 1999 JavaOne conference in San Francisco), we started checking out the possibilities of XML and XSL technologies. The interesting thing about XML and XSL was the fact that they were being adopted by nearly everyone who was anyone. Microsoft had initial versions of these components available, and IBM and other Java vendors were hard at work on their own versions.

Although we had yet to get our hands dirty by actually implementing an application using XML and XSL, we could see that this might be a way to solve our problems.

So we decided to implement the WML solution using XML and XSL components from Microsoft. We started out by defining exactly which data we wanted to present. This was not too difficult because the WML part was to be integrated into the running online banking solution. The application already consisted of the components that provided the data. We then developed a single ASP page that accessed the data and built an XML format using the available parser. When the XML data was available, it was transformed into WML using the correct XSL style sheet.

(If you are not familiar with all these components, don’t worry; we will explain the details in Chapter 2.)

And it worked. In fact, it worked so well that we were able to hand over the time-consuming part of supporting the various mobile devices to our customer (one of the great secrets of the software business).

For the first time, we had built an application that separated the different areas of an Internet application—layout, data, and site management. Each area could be worked on by a separate person or instance, allowing development to happen in parallel and minimizing side effects due to changes.

Taking our specific example, the style sheets that formatted the data could be developed by someone who had no idea how the data (such as account balances) was accessed by the underlying components. For example, a stylesheet for a Nokia phone could be developed by one person, while someone else could work on the style sheet for a Siemens phone. Each style sheet could then take the devices' differences into account. Any device that came out in the future could be integrated easily by developing and deploying a new style sheet.

In the Cocoon project, these areas of development are called *concerns,* and Cocoon aims to separate these concerns (often called SoC, for Separation of Concerns) so that they can be worked on in parallel and without affecting each other. Going further than our first solution, Cocoon also allows the application's architecture (or design) to be separated from both data and design.

After the success of our first XML and XSL application, we decided that it would be a good idea to write a generic platform solution that would offer a more flexible way of writing Internet applications than we had been doing up to then. We also thought that there might already be something available that would help us do this. In order to determine what functionality the end result needed to provide, we had to take another step back from our everyday problems and evaluate how Internet applications were being built from an architectural point of view.

Application Architectures

When we started writing software for large financial corporations in the early 1990s, we wrote programs in C++ that ran on PCs (the clients). These programs connected to server systems that supplied the PCs with data they had obtained from a mainframe. This was the next step from when workstations connected to the mainframe and host emulations were used to access host applications directly.

The PC programs were what would today be called *thick clients,* providing both a rich user interface and decentralized business logic. The servers were mainly used to route the data from the mainframe to the client. They contained little or no logic themselves.

During the 1990s, more and more logic was transferred from the mainframe and client to the server. The clients became thinner, and the mainframes became dedicated data suppliers. This client/server architecture was in place when the Internet and its technologies started taking over the corporate IT world. This architecture was well-suited for the typical tandem of browser and web server we see today.

Today's Internet application architectures are typically organized into three layers or tiers. This is what you most commonly find in corporations that have migrated their client/server architectures to Internet technologies. The tiers consist of a client; a server, which provides some form of middleware application; and the mainframe or other legacy system, which provides the data.

So when you perform a function such as retrieving your account balance, that data flows from the legacy system through the middleware and is then displayed in a program running on the client you are using, such as a PC, a mobile phone, or a TV.

Clients

In many areas, the thick clients of the 1990s have been replaced by the browser. Because it is widely available on all platforms, the browser is an ideal publication front end for applications that run on the server. The server application sends pages formatted in HTML to the client, where they are then displayed. The first platform to support the browser was the PC or workstation. As the PC took over as the operating platform for other devices, such as the bank ATM, the browser moved with it. Even though you might not know it when withdrawing cash, the ATM might well be running Internet Explorer or Netscape.

The first way of accessing the Internet was through stationary devices such as the PC. A big drawback of these devices is the simple fact that it is not easy to take them with you. As the Internet became more of a way of life, often used to access up-to-the-minute information such as stock prices and news, ways of accessing that information on the go were needed. Portable devices such as the mobile phone and the PDA were already available, so it was natural that the Internet should become available on these devices as well. The mobile professional, out in the field, can use these devices to request information and send details of the customer he is currently visiting back to his office.

The formats these devices understand are more diverse, ranging from WML to cHTML. The standardization in this area is not yet as far advanced as HTML and the Internet browser. However, the formats used in mobile devices, because the development is more recent, tend to be based on XML. This makes these formats a lot easier to support in the XML-based architecture we will discuss in Chapter 2.

PCs/workstations and mobile phones are the two classes of Internet clients you are most likely to find at the moment, but there are many others. Some consumer devices offer Internet access from the comfort of your own couch. Televisions are available that allow you to surf the Net using specialized browsers while a TV program is playing in the background. This allows you to call up additional information on the program you are watching or take part in interactive games.

Even the car radio has become an Internet device. In tandem with a mobile phone, a car radio can call up on its display web pages that help you navigate around traffic jams or find the nearest restaurant. Obviously, a radio has display properties very different from a

PC, so the data the radio receives needs to be in a format it can display properly. Because visual information is distracting when you're driving, the information might also need to be provided in an audible format.

The Internet is available not only on devices you would associate with the web, but also on devices you would normally not expect. We have already talked about the fact that an ATM can run a browser just as well as a PC, but it can also use the Internet to communicate and retrieve data. ATMs used to be banking devices that had only one function—dispensing cash. Then other capabilities were added, such as accessing your account balance or making a transfer. In addition, the communication networks were standardized, and the protocols became Internet-based. Because banks are always looking for ways to attract customers, they quickly found out that the Internet could be utilized to display real-time information, such as stock quotes, while the money was being dispensed.

When we enabled the Internet for devices such as ATMs, we used available formats such as HTML and client programs such as browsers to display that information. This works as long as the device resembles a PC. The less the device looks and acts like a PC, the more difficult it becomes. This is the reason new formats have appeared and are evolving for the different Internet clients that are available.

As the number of different devices and formats grows, it is becoming increasingly important that an application present the same information to the user in a format the device can understand. If you access your online banking account from your home PC, from your mobile phone, and from the ATM, you expect to receive exactly the same information in all cases, but presented in a device-specific manner. If a salesman accesses customer data from his workstation and then from his PDA while on a train, he expects to see exactly the same data.

That salesman also wants to be able to print documents containing the same data but in a format such as Adobe's PDF. This is one area that web applications have always been reluctant to tackle, because the formats they can generate, such as HTML, are unsuitable for printing contracts and binding documents. PDF is a format that maintains a page's "print integrity." Banks consider it very important that any solution that can print information for the user must be able to print it in a format such as PDF.

One of the first Internet applications we installed was a solution that allowed sales-people to enter all the information they needed to provide a customer with financing for goods such as cranes, boats, and cars. After the data was entered, by way of a Java applet, the salesperson could send the contract to the browser as a PDF document so he could print it for the customer to sign. As soon as the application on the server received the data from the applet, it generated that data into a PostScript template. Then another program converted the PostScript file into PDF, and that file's link was returned to the browser. Not exactly cutting-edge. We were basically using one application to present the data inside the Java applet and another, separate application to convert and present that data as PDF.

Imagine the number of separate applications you would need if each could publish that data in only one specific format. You would have an application that accessed a database and formatted the data into HTML. Another application would access the same data and publish to WML for mobile phones. A third application would be needed to format the account balance for a speech format such as VoiceXML. As each new format and device were released, developers would be rushing to develop a new application to support it. Imagine the maintenance costs of all that software if someone decided to change the underlying data format.

This might seem to be a problem that has developed because of the influx of various Internet formats, but that is not the case. Looking closely at traditional applications we wrote for banks before the Internet came about, we discovered that many of these showed the same problems.

Imagine a corporation that stores its customer data in a large database. That database contains all the transactions for a particular customer over time. One department wants a reporting tool that publishes the data in a statistical format. A second department wants to receive the customer data in a printable format, such as PDF. A third department wants to write each customer a letter and therefore requires an address format. Normally, separate applications would be written for each particular function.

Often, when these applications were then migrated to Internet technologies, the same thing happened. Each traditional application made way for an individual web application, when in fact the better way would have been to implement just one application between the data systems and the clients. That application would be able to publish the data from the database into each format needed.

A middleware application like this is a typical use case for Cocoon. Indeed, flexible publishing to various output formats is one of its great strengths. Because Cocoon uses XSL style sheets to format the output, it can publish data to a variety of presentation standards, such as HTML, WML, PDF, and VoiceXML. It also allows you to add your own components so that you can publish to a specific format you might require.

Middleware

The middle tier of an Internet application architecture is often called *middleware* because it lies between the client and the data storage or legacy systems. Apart from publishing data to a particular format, middleware applications are also responsible for accessing data and integrating functionality that may be implemented on other systems.

One of the most common functions of a middleware solution is *data aggregation*. For example, imagine a “get my account data” function, typically found in an online banking solution. Although the end result of this function is your current account balance, a lot can go on in the background. When you click the “get account data” button in the web application, several things can happen. First, your customer data (who you are) is fetched from the first database. Next, a function is called on the legacy system. (This system,

typically a mainframe, stores your account balance.) Then, depending on your account balance, a system responsible for customer relationship management (CRM) might be called to indicate that a bank representative should call you with an offer of investing in some company. So a single function to you as a user might actually be a combination of several functions that go on in the background. The middleware hides this fact from you. An ideal solution also allows these steps to be reconfigured or changed without you noticing.

Another common example of this type of functionality is a middleware application that provides *content syndication*, which is commonly found on news sites. A news site accesses various sources of information and combines the data into a single layout, such as an HTML page. The different news sources can be databases, other Internet servers that offer their news for syndication, or a content management system in which journalists type in local news as it comes in. A good site will allow you to configure the news you are interested in and then will access only that data for you.

As you can see from these examples, a middleware solution provides the integration platform for diverse systems such as a database, a messaging system, a mainframe, or another web server, perhaps running another application. The interfaces and necessary formats for these systems can be either standardized, such as using SQL for database access, or proprietary, such as a corporation-specific protocol that allows access to an application running on a mainframe.

Cocoon is a solution that enables you to build such a middleware application. Apart from allowing the publication of data to various formats, it also consists of components that allow the integration of various systems, such as the ones just mentioned. For example, Cocoon has a component that enables you to integrate a database into a Cocoon-based solution. You will learn more about the specific components in Chapter 4, "Putting Cocoon to Work." The Cocoon architecture also allows you to combine these components to build an application containing functions such as the "get my account data" function mentioned earlier. Because the architecture is extensible, it allows you to add your own components. This might be necessary if you want to integrate a back-end system that is not supported by Cocoon out of the box. We will show you how to write your own components in Chapter 9, "Developing Components for Cocoon."

Back-End Systems

One of the major functions of a middleware solution is to hide the back-end systems from the clients. By this we mean that the client accesses the middleware to obtain the data and does not need to access the mainframe directly.

Back-end system is a term most often used in connection with mainframes or host systems. The term *legacy system* is also often used to describe mainframe applications that are specific to a particular corporation. When we are on-site at a company, such as a large bank, we seldom get to see or access these systems directly. It is our experience that the people who work on these systems are treated with a lot of respect, and their opinions

are valued. There is a simple reason for this. Often the applications running on these systems have been in place for many years. Many times, the original programmers no longer work for the company, so anyone who knows how to change the code is an important asset to the company.

A large middleware vendor once told the following story at a training course we attended: When a large German airline decided to migrate its mainframe applications over to a more modern software architecture, they discovered that only a couple of the original people who had written the application were still alive. In order to make sure no mistakes were made during the transition period, the airline flew one of the programmers in from the U.S. to watch over the process.

It is our experience that when a decision needs to be made as to whether the mainframe application needs to be changed—or whether the middleware can be adapted as a workaround—the middleware always wins. We have even had to adapt our middleware to compensate for errors in host applications. It just takes too long to change the mainframe solution, and the programmers are just too expensive.

During the transition period into the year 2000, banks and other large companies paid top money for programmers who knew how to program in, for example, COBOL. All the mainframe applications had to be checked and many rewritten because of the two-digit year problem. When the original programs were written, in some cases decades before, nobody thought the year 2000 would be a problem, because the software would be long gone by then.

Because these systems have been around for such a long time, they often are incompatible with one another or with modern application architectures that might use a standardized format to exchange data. Therefore, it is the job of the middleware to form a bridge between these systems so that the end result does not look as though it comes from completely different sources and in different formats.

Without using a solution such as Cocoon to build this bridge, an application written to access account information from a legacy system would not be able to integrate an XML feed of stock quotes.

Cocoon in the Middle

As explained in the previous sections, Cocoon is well-suited to building middleware solutions that shield clients from the back-end systems. Because corporations are unwilling to alter their applications and because of the existing protocols, a middleware solution must be able to integrate these proprietary systems as well as standard ones.

Cocoon allows new components, written for exactly this purpose, to be integrated into the given architecture to form a complete solution. These components can then access the mainframes, databases, and any other system using whatever protocol and format is necessary. The data they obtain can then be formatted into a common XML-based format

and merged to allow the presentation inside an application such as a browser. Figure 1.4 shows what a complete solution using Cocoon might look like.

Figure 1.4. A Cocoon-based, three-tier architecture.

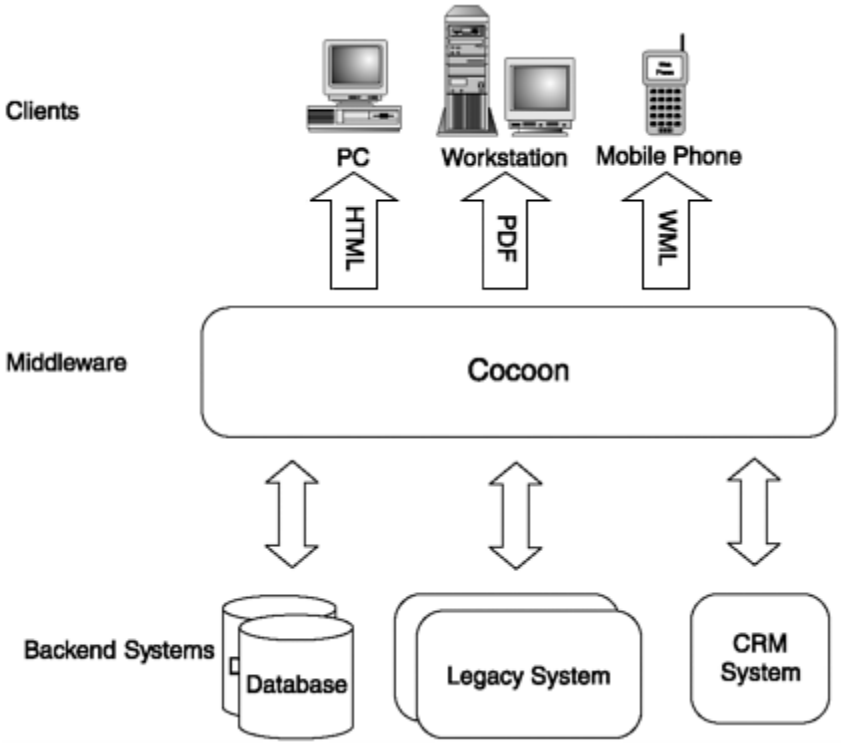


Figure 1.4 shows how Cocoon can access data from a variety of systems and then format it for presentation in the required output format. The Cocoon architecture enables you to merge the different data to give the client a single presentation format that hides the source of the data and its original format from the viewer.

Up to now we have looked at Internet application architectures from a more technical viewpoint, showing how different systems need to be integrated into a common middleware solution and how the data must be presented in a device-specific manner. However, there are also other challenges that a modern software solution needs to solve for it to be a success.

The Challenges of Building Internet Applications

We have already seen that publishing data to various formats and integrating diverse data systems are two of the major challenges facing a modern Internet application architecture. However, there are also other challenges that must be met. Apart from being able to integrate data, any new solution installed in a large corporation must also be able to support applications that might already be installed there. Because the Internet has speeded up the half-life of technologies, a solution must also be as platform-independent

as possible so that it can survive major changes to such things as the operating system and can run in as many environments as possible.

Being able to personalize the data the application sends to the client is perhaps one of the requirements we hear most often when talking to customers about new middleware solutions based on Internet technologies.

Personalizing Content

Anyone who surfs the Net, trying to find a certain piece of information, can quickly suffer from information overload and have difficulty finding exactly what he is interested in. This is also the case, although on a smaller scale, inside corporations on their intranets. The popular term for an application that allows you to control what the client receives, instead of sending everything, is *portal*.

A portal presents the available information in a way that allows the "user" to configure the presentation so that he receives only the content he really needs. The reason we put the word "user" in quotation marks is because the user might not be a person. It could also be a device. A portal that serves information to devices needs to select that information based on information about the requesting device—just as a person interested in the weather in San Francisco wants to receive only that information. He doesn't care whether it is raining in Paderborn, Germany (which often it is).

A portal is often a major part of a middleware application because it provides a personalized view of the data to the user. Portals can be built either as single pages of information or as pages that contain several blocks of information obtained from different sources—just as the personalized news site we mentioned earlier is considered a news portal.

So, apart from being able to integrate data from various sources, the middleware solution also needs to allow the user to tailor the information in different ways, such as moving information around, adding or removing information, and changing the colors or fonts. A portal also needs to be able to integrate complete applications as part of its content.

Integrating Applications

When we approach a customer about installing a new Cocoon-based middleware solution, we are often told that he already has several different web applications he wrote when he migrated his traditional client/server architecture to a web-based architecture. Any portal that is to be implemented must therefore be able to integrate not only data, but also these existing applications.

When Internet technologies began to take over the corporate IT world, large corporations rushed to develop new Internet applications to replace their traditional ones. However, this often resulted in different departments producing applications that did not have a

common HTML look and feel, and sometimes they also contained duplicate code needed to access the legacy systems.

For example, imagine a portal that is built to serve internal corporate data to employees. Each employee logs on to the portal using an ID and password. After logging on to the portal, the employee sees current data and can start an application that provides access to his retirement fund plan. This application was written before the portal was established and ran as a stand-alone web application. Now that the employee is logged on to the portal, he should be able to start the application without having to log on again. This capability is often called *single sign on*, and it must be provided by the portal.

Also, if the application already publishes its data to an Internet format such as HTML, a portal should be able to integrate that format as part of its content. If this is possible, the portal can provide a common look and feel across all applications, whether old or new.

Because corporations often already have an Internet infrastructure in place, on which they perhaps also run web-based applications, it is important that any middleware solution can run on that given platform.

Platform-Independent Solution

The IT world changes quickly—sometimes overnight. Although large corporations are not able to change their infrastructure as quickly as an individual, they are also known to throw out one operating system and change to another at the drop of a hat. For example, in the 1990s, corporations were willing to pay large sums of money to have their software ported from OS/2 to Windows NT. Because of the number of applications and the increasing speed at which new operating systems are sometimes released, by the time the applications were all ported to NT, all the clients were running Windows 2000.

Today, things are different, because firms have reduced the amount of money they are willing to invest in IT infrastructures. The lack of corporate investment presents an additional challenge to the software supplier: being able to supply software that will still run if the underlying platform changes. This is one of the reasons Java has been so successful on the server. This is also the reason applications such as application servers have become popular for hosting middleware solutions. Application servers built on top of Sun's J2EE architecture allow the integration and combination of modules such as servlets and Enterprise Java Beans (EJBs). These components are then interconnected to build the application that is needed.

So, instead of monolithic server applications being built, the current trend is toward a component-based architecture on the server. Because of this, a middleware solution must itself be built from components, must run in a variety of infrastructures, and must allow additional components to be easily integrated.

A common requirement in today's web-based architectures is for *offline page genera-tion*.Very often, corporations want to be able to generate a current view of their

web pages—as if they were taking a snapshot of their site at a given time. These snapshots are often deployed to other web servers to be served statically, perhaps for performance reasons. Another goal is to be able to put all the generated pages on a CD that can be handed out to customers or at training courses. So, apart from perhaps integrating the solution into an online scenario such as an application server, the application needs to provide alternative methods of being called (for example, via the command line).

As you can see, all these requirements result in the demand that the Internet application provide a very flexible architecture.

Flexible Architecture

Today's middleware applications are built using a combination of various components that can be reused repeatedly. Although Cocoon is written in Java and this book therefore focuses on Java components, the same is also true of Microsoft-based architectures.

Gone are the days when software companies received contracts to build complete and singular solutions. Now the goal is to build applications from standardized components and add a little specialized "glue" here and there. The glue can be a configuration file that is adapted to integrate the new component. It could also be something like a script that defines in which order the components are to be called to process an incoming request.

You can compare this method of building applications to cooking a Mexican meal. The "components" are things such as meat and vegetables. You can make a variety of things from these components. The "glue" we mentioned would be the spices and the habanero chiles you use to make it a Mexican dish and not something else.

Any customer who is interested in installing an Internet application architecture will want to be able to drop in available components as easily as possible. In order for this to be possible, the architecture must define clear interfaces that these components can then adhere to. Using the documentation of these interfaces, the component builder can build a new module to, say, integrate a proprietary system and then add it to the middleware solution by perhaps just adapting a configuration file.

Apart from defining the interfaces to the components, the solution must also make clear how these components are called, when they are called, and what is expected of them. This is often dealt with in the solution's documentation and—in the case of Cocoon—is enhanced by secondary literature such as this book. A book on Cocoon must also help the reader decide whether the solution as a whole is suitable for his specific needs and challenges.

Using Cocoon to Meet the Challenges

This chapter dealt with the various challenges a modern Internet application architecture must address in order to be a success:

- Flexible publication of data to various formats
- Integration of diverse data sources
- Personalization of content
- Integration of applications
- Platform-independent solution
- Flexible architecture

There are probably also other challenges that arise from specific situations, such as what type of application to build or the exact type of data sources to be integrated, but these challenges are the ones we hear most often from our customers when confronted with a new middleware solution.

Another challenge that you have perhaps missed up to now is that of the solution's cost. Some solutions, such as commercial application servers, meet several or all of the challenges just listed. However, they come at a cost.

Cocoon is a Java solution that is freely available today in source or binary form. It runs in a wide variety of given Internet infrastructures, such as on web servers, and it offers a highly extensible architecture. In addition, it offers the possibility of being called from the command line via an alternative interface.

After you've installed it, Cocoon offers components for integrating data sources and for flexible publication to various formats. It also allows you to personalize what the client receives and to integrate other web applications you might already have.

All this is important, but the fact that Cocoon is completely XML- and XSL-based is perhaps what makes it stand out the most from the commercial offerings you might be familiar with. Why you should worry about deploying a system that uses these technologies is something we will discuss in the next chapter.

Chapter 2. Building the Machine Web with XML

In the fast-paced world of Internet software architectures, where technology changes quickly and new de facto standards are sometimes born overnight, any new application architecture must be able to meet the challenges laid out in the Chapter 1, “An Introduction to Internet Applications.” Before any corporation invests in a new application architecture, it first compares it to whatever it is currently using and sees how the new offering can solve its problems.

Most of the available Internet applications were originally designed to publish data in a specific format—HTML. However, an HTML-based architecture is not suited to providing for the variety of devices and necessary formats required by today’s growing number of Internet devices and applications.

The XML family of standards was defined to alleviate the deficiencies of HTML. It is now the language of choice for computer systems that want to exchange data with each other over a network. The Apache Software Foundation hosts a number of projects in which industry-strength, open-source XML components are being built. Apart from projects that are developing single components, the Cocoon project provides a complete XML architecture for building XML solutions. In many cases, you can do this without writing a single line of code.

To see why XML offers so many advantages for building Internet applications, we must first take a look at HTML and see why it is not well-suited for a web in which machines communicate with each other.

HTML Applications

Most Internet applications you use every day—such as your favorite news page, the search engine that helps you find what you’re looking for, or the online banking application that tells you how much money is in your account—were built to present information in a format that was invented in the early 1990s.

Tim Berners Lee invented HTML in 1991. He originally designed HTML to present linked information on a computer network in a form suitable for human viewing. When

the first web was established at the European Particle Physics Laboratory, the server ran on a NeXT machine, and a simple browser program allowed the web pages to be accessed from a variety of platforms.

HTML has since become *the* language for presenting information on the web for humans to read. When you look at a page of HTML, you can determine what the information means from the way it is formatted and what other information is presented with it. But when you need to send the same information over a network so that it can be processed by a software component, you need to take a closer look at whether HTML is the right format to use.

The Meaning of Data

Because HTML was designed to present information to humans, it has many drawbacks when that data needs to be interpreted by machines. One of the major deficits is that as soon as the data is formatted into HTML, the original meaning of the data is lost.

When you look at HTML pages, you can determine the meaning of what you see by how the data is formatted or by other information that is presented with it. You use the visual context of the HTML page to understand the data's meaning.

For example, consider an HTML table. The data in that table might have originally come from a variety of sources, in which each column of data had a specific meaning or type. Listing 2.1 shows a simple HTML table.

Listing 2.1 A Simple HTML Table

```
<table>
    <tr>
        <td><b>Matthew</b></td>
        <td>1964</td>
        <td>36</td>
        <td>43</td>
        <td>183</td>
    </tr>
</table>
```

A proper HTML table would probably have an additional row with the headings for each column. As you can see from this example, leaving off the headings makes it difficult to determine what the information means (even if you understand what 1964 means, what does 183 refer to?) and what relationship each column might have originally had to a previous one. Each piece of information is formatted with the *same* `<td>` tag. This is fine when *you* view the table, complete with titles and so on. But how would a machine be able to interpret the table and decide what each column meant?

If you took HTML as the "base" format and sent your sample table to a component for processing, how would that component know what each piece of data meant? The difficulty lies in the fact that HTML mixes content with the layout.

How HTML is formatted takes into account how the visual representation of the data should look to a human viewing it. In the preceding table, notice how the name is formatted with the <b> tag, for bold. Every browser that presents that data knows how to interpret each HTML tag. A machine or application *can* interpret data formatted in HTML, as long as the goal is to present the information in a visual and predefined way (in this case, as HTML defines the presentation). If a browser encounters the <b> tag in an HTML page, it will always emphasize the enclosed text by making it bold.

By using HTML as the base format, and because of the way a table holds no information on what the data actually means (apart from any textual information you care to add as a heading), you lose the semantics of the underlying data. If a software component received the preceding HTML table with the goal of sending it on to a remote machine for processing shoe sizes, how would that machine be able to determine which of the columns contains a (European) shoe size and which column contains an age?

This is the basic problem with formats that are designed to combine data and layout information in one presentable screen format. Although this might seem to be a problem specific to HTML, this is not the case, as you will see in the next section.

Extracting Data from Screens

There *are* ways for machines to interpret visual data and extract the information from them. In fact, the technique for doing this goes back to the time before HTML had even been thought of.

In Germany, one of the earliest online banking networks was set up by the German post office monopoly in June 1980. It was called BTX for Bildschirmtext (screen text). BTX ran on a mainframe and offered various services via dial-up phone lines. The killer applications of BTX were the online banking applications provided by the different banks and hosted on the BTX system. It was a great success, because you could use a computer for banking transactions from the comfort of your own home. This was long before Internet banking, or even the popular Internet.

In 1997, when we wrote the first Internet banking solution (discussed in depth in Chapter 1), we had to provide a system that could interface with the banks' running BTX system. The screens of the BTX banking application were based on the Conference Europeenne des Administrations des Postes et des Telecommunication (CEPT) standard, a way of defining screen data in which the layout is standardized in 24 rows and 40 columns. Each screen the BTX system sent to the client was filled with the appropriate data (such as your account number) and also contained attributes to determine how the information should be presented. The screens also included fields where data had to be entered in order to send that information back to the mainframe.

The way to access the screens was for a software component to read a complete page into a buffer and then to navigate to, say, line 1, column 6, and read exactly seven characters (the customer's account number). Then the component would access the data at some

other fixed position in the screen buffer and extract the account balance, and so on. This technique is called *screen scraping*.

Our solution included a component that could scrape the screens and access the needed information. About three months after we installed the solution, things started going wrong. The bank changed the screens' layout (remember, the BTX application was still running and being used in the old fashion), and our screen scraping failed abruptly. We got around that by making the scraping scriptable in the application—and by telling the customer, "If you change the screens, you'll need to change the script."

There was just no other way to do it, because what each piece of information actually meant was lost in the screen representation. You couldn't just scan through the page until you came to the information labeled "account number" and extract it, regardless of the position. There was no label.

It is the same situation with HTML. You can scrape each HTML page and extract the information if you know exactly where in the page the information you are interested in is. Of course, you then hope that no one changes the HTML page in the future.

We used this example from an older, non-Internet-oriented technology to show that the problem of mixing layout and content is not new—and it has nothing to do with technology. In order to avoid these problems, you need a completely different approach—both in technology and in how you plan your application—as to how you will separate the content from the layout information. Just as the CEPT standard made it hard for software components to easily access the data contained in the screen, HTML formatted pages make it difficult to access the data contained in them.

As HTML and the web became popular, browser vendors added ways of allowing more information about the page to be authored into the HTML information, using such things as *metatags*. In the end, however, HTML still remained a format for displaying data to humans.

XML Arrives on the Scene

By 1996, it was obvious that HTML had too many limitations to make it the language of the *machine web,* a web where computers could communicate with each other in an open and cross-platform fashion. The companies that wrote browsers also began to extend HTML so that they could exploit the advantages of their own offerings without having to rely on a committee to standardize the rapidly changing format. Therefore, in 1996, the World Wide Web Consortium (W3C) started a project to define a new technology that would move the web into the machine web age. (The W3C is a neutral forum formed by Tim Berners Lee, where companies meet to discuss how the web should progress and define new protocols or technologies that enable this.)

This project was the beginning of what we now call XML. XML has grown from being just a way of defining data to allowing the definition of that data to be published in a

standardized way. It also allows flexible transformations from one format to another and forms the basis of the various XML components that allow applications to be built.

Extensible Markup Language (XML)

In 1998, the W3C released version 1.0 of the Extensible Markup Language (XML) recommendation. XML was created by a group of companies, including Microsoft, Adobe, Sun Microsystems, and many others. These companies made up the W3C XML Working Group. XML was defined as an "open, human-readable format that does for data what Java does for programs" (refer to http://www.w3.org/Press/1998/XML10-REC).

One of the most noticeable things about this new format was the fact that competing companies designed it jointly. This was perhaps *the* key factor that made the technology become so successful so quickly.

Although XML 1.0 defined what tags and attributes are, there is now a whole family of standards that make up XML. Technologies such as XLINK, XSL, XPointer, XFragments, and XML Schemas exist today to allow applications to be built on top of these standards.

There are already a number of books on these subjects, so we refer you to those for a deeper look at these technologies. However, because many of the XML family members have made their way into Cocoon, we will introduce you to the most important ones as we progress. We will also provide you with the background information needed to understand key Cocoon concepts and the examples provided in this book.

So, what is XML, anyway?

XML is one of those buzzwords that vendors and company managers tend to use in a fashion that shows they don't really grasp what XML actually is. Here are some statements we have heard when talking to customers about XML solutions:

- "Our data format is XML. Can your system interpret that?"
- "We don't want a system that uses XML. It's too slow."
- "If we send our data in XML, everyone can read it."
- "XML was invented by Microsoft. We don't use Microsoft software."

Obviously, Microsoft was only one of the companies involved in defining XML, and of course XML has nothing to do with the software you use, but what about the other statements? Apart from the fact that many companies took part in the definition of XML, its success is also due to its being an open and extensible way of defining a data format. After it is defined, a strict rule set about the data can be published so that other interested parties can design software to understand the format. You can also define your format to be globally unique so that it doesn't get mixed up with someone else's definition.

As you can see from the preceding statements, XML's openness is something that many people find hard to understand at first.

Building an Open Format

XML was defined as a way of describing data in an open, readable, and structured fashion. XML itself is not a format you can actually use in your own application. XML is a rule set that tells you how you can define an XML-based format for your own use.

If you want to send your customer data in an XML format, you have to define that format yourself, or use a standardized format if one is available. One of XML's key advantages is the fact that many formats have already been defined and their specifications published. We will discuss this in more detail later in this chapter.

XML data is open. By this, we mean that you can send your XML data from an application written in C++ on a UNIX machine to an application written in Java on a Windows system and, if the systems have the necessary components, they will be able to read and process the data. Although designed as a way for machines or applications to communicate with each other, XML is also a format that you can read without the aid of software.

By supplying a set of rules with your data, the remote machine or application will even be able to "understand" and check what you are sending. After you have defined the rules of your format, you can exchange them with other systems you want to communicate with. You can also publish these rules so that applications that don't even exist yet can understand your data at a later date.

However, XML does not *force* you to be open. In fact, you can define a format based on XML and make it just as proprietary as the binary format you were using before. The XML family provides the "tools" to make that format open and cross-platform or cross-company. Whether you actually use them is still up to you. This is one point that is often missed when companies build their first XML applications.

What's Your Format?

The following example shows the problems that can occur when XML is used only as a way to define a data format. It also shows why you need to take into account other members of the XML family of standards when designing an XML application. The example shows that as soon as you've decided on an XML format and written the software, there are further steps you should follow to make full use of all the advantages of XML.

At the headquarters of a large bank, the department responsible for customer relationships wants to build a new XML-based application. They first define the format to represent a customer, as shown in Listing 2.2.

Listing 2.2 Customer XML Format

```
<customer id="">
    <name/>
    <address/>
    <account>
        <id/>
        <since/>
        <balance/>
    </account>
    <importantcustomer/>
</customer>
```

As you can see from this example, an XML format is always readable. This does not mean that the data contains information on how it should be presented, but that it *can* be read without relying on a machine to interpret it. This can be very important when designing and testing XML applications.

The format is made up of tags. Using logical names makes the information easier to understand for humans reading the data. Each tag can have *attributes* (such as `id` being an attribute of `<customer>`) and *children* (other tags that are logically "inside" or enclosed by the parent tag). As opposed to HTML, the tags only delimit the data. They do not tell the application that reads the data what it should do with that particular piece of data. The `<b>` tag in HTML is an example of the difference. A browser that receives this tag will always format the enclosed text in a bold font. The `<b>` tag acts as an explicit command for the browser.

XML data is structured in a logical fashion. As you can see, the example has a logical entity called "customer." Customer has a name and address and contains information on an account. The account is also structured in a logical fashion, containing the ID, when the account was created, and the current account balance.

Note that XML does not force you to define your data in a logical fashion. You can still make a mess of defining your data if you want to. Instead of using descriptive names for the tags in this example, we could have chosen meaningless combinations of letters and numbers if we wanted to. For example, the customer format shown in Listing 2.3 conforms to XML.

Listing 2.3 Bad Customer Example

```
<poiu xx="">
    <xxc/>
    <ysq1/>
    <dfr>
        <blag/>
        <ABC1234567/>
        <fgh/>
    </dfr>
    <ghjgj/>
</poiu>
```

It should be obvious from this example that using descriptive names is the better option.

After you've defined the format, the next thing to do is to write the software to process the data and to make sure the format is documented.

Publishing the Format

After deciding on the customer format, the bank's customer relationships department builds the server software that can understand and interpret exactly the defined format (after all, they know exactly what each tag means). To reduce the amount of data sent, they define that two things are optional in their format: <importantcustomer>, with the default being "no," and the <account> tag, in which all the children can be left out if the customer has not yet purchased anything. They also define that the <address> tag is mandatory because the old legacy system needs it—but the word "none" can be used to define that no address has yet been entered.

When the server application is complete, the department builds a web front end that is able to understand the defined format and can also handle the implicit options that were chosen. Because the same people wrote both server and client, there is no need for the optional tags to be documented. After the software has been tested, it is installed, and it functions without problems for some time.

A few weeks later, a foreign branch of the same bank decides that it can integrate its Java-based system into this new XML system. So the branch asks for samples of the data to be sent. They receive the examples shown in Listing 2.4 from the implementing department.

Listing 2.4 Sample Customers

```
<customer id="1234">
    <name>Carsten</name>
    <address>Delbrueck, Germany</address>
    <importantcustomer>no</importantcustomer>
</customer>

<customer id="1235">
    <name>Matthew</name>
    <address>Paderborn, Germany</address>
    <importantcustomer>yes</importantcustomer>
</customer>
```

The foreign branch looks at the data they receive and builds a system that can send and receive this format. They build the system, test it against the test data they received, and are happy.

They then go into production and wait for the first customer data to be sent from headquarters. Imagine their surprise when the first customer data comes in (see Listing 2.5).

Listing 2.5 A Real Customer

```
<customer id="AYX1234">
    <name>Christopher</name>
    <address>none</address>
    <account>
        <id>1234AD</id>
        <since>1997</since>
        <balance>-24,98</balance>
    </account>
</customer>
```

What's this? The customer ID is not numerical, the address is "none," there is an extra `<account>` tag with lots of additional tags. And where is the `<importantcustomer>` tag? The format that the server actually sent to the new Java client looks quite different from the sample data that was first sent. If the new program was written to handle exactly the type of data the examples had, it would not work when it receives a customer with all the additional information. After all, nobody told the programmers of the client application about the optional tags.

This is where an important factor of XML-based data comes into play—document *definitions*. In the form of Document Type Definitions (DTDs), these definitions are a logical description—or rule set—of the data. Listing 2.6 shows what the DTD of the sample data looks like:

Listing 2.6 Customer Format as DTD

```
<!ELEMENT customer (name, address, account?, importantcustomer?)>
<!ATTLIST customer id CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT account (id, since, balance)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT since (#PCDATA)>
<!ELEMENT balance (#PCDATA)>
<!ELEMENT importantcustomer (#PCDATA)>
```

Using a standardized notation, it is possible to author a rule set that defines how the customer data should look. For example, note how on the first line the tags `account` and `importantcustomer` have a `?` after them. This means that they are optional. Also notice how `account` is defined to consist of `id`, `since`, and `balance` on the fifth line.

Why is a DTD needed? In this example, you can see that it was obviously not enough for the foreign branch to receive a few examples of data, because they had no way of knowing whether the information contained in that data was complete.

However, if they had received the DTD and then built their software to that specification, rollout day would have been a success. Also, as you will see later, some software components are available that can use this information to check whether the data they receive is correct.

Another way of thinking about these types of definitions is that you are writing a contract. When you give the definition to someone else, you are basically saying, “I will send my data in exactly this format. You can depend on it.” Sending your data in a different format then means that you are in fact breaking the contract.

DTDs are now being replaced slowly by XML Schemas. One of the disadvantages of DTDs is the fact that they are not defined in an XML language. Listing 2.7 shows what Listing 2.6 looks like as an XML Schema.

Listing 2.7 The Customer Format as a Schema

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="customer" type="CustomerType"/>
    <xsd:complexType name="CustomerType">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="address" type="xsd:string"/>
            <xsd:element name="account" type="AccountType"
    minOccurs="0"/>
            <xsd:element name="importantcustomer" type="YesNoType"
               minOccurs="0" default="no"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string" use="required"/>
   </xsd:complexType>
   <xsd:complexType name="AccountType">
       <xsd:sequence>
           <xsd:element name="id" type="xsd:string"/>
           <xsd:element name="since" type="xsd:gYear"/>
           <xsd:element name="balance" type="xsd:decimal"/>
       </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="YesNoType">
      <xsd:restriction base="xsd:string">
          <xsd:enumeration value="yes"/>
          <xsd:enumeration value="no"/>
      </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

As you can see from this example, XML Schemas allow you to be more exact when defining your data. In this example, you can explicitly define the types of the various data items.

So a key to XML’s *interoperability* is the publication (whether inside an organization or globally) of the rule set that defines the data you will be sending. Imagine how important this becomes if you want to send the same customer information to completely different companies around the globe.

Global Definitions

The Internet is a network that links companies around the world. How do you know that no other company has also defined a <customer> model based on XML with a completely different meaning and structure? What happens if you send your <customer> to that company? Clearly, this is a problem. In order to allow the interoperability of different definitions like this, the concept of *namespaces* was defined as part of the XML standard.

In general, namespaces are used to uniquely define the meaning of the elements and attributes in an XML document. They are used to make sure that your definition of, say, a tag called <customer> does not get mixed up with someone else's definition of a tag called <customer>. So, to make sure this works, you mark your element with your namespace. It is basically your way of saying, "This is my definition of a customer."

In XML, a namespace is identified by a Unique Resource Identifier (URI). In most cases, this is a web address, such as http://www.s-und-n.de/customer. Namespaces usually are defined using the special xmlns attribute inside the XML document. This attribute is often used in combination with a prefix, such as xmlns:customer, xmlns:xsl, or xmlns:xsd. The name following the colon is a placeholder for the URI. Using this prefix for the elements applies the namespace to that element. As you can see in Listing 2.8, the customer tags now have our namespace, so there is no danger of this definition becoming mixed up with someone else's.

Listing 2.8 Customer Example with Namespace

```
<mycust:customer id="AYX1234" xmlns:mycust
     ="http://www.s-und-n.de/customer">
     <mycust:name>Christopher</mycust:name>
     <mycust:address>none</mycust:address>
     <mycust:account>
          <mycust:id>1234AD</mycust:id>
          <mycust:since>1997</mycust:since>
          <mycust:balance>-24,98</mycust:balance>
     </mycust:account>
</mycust:customer>
```

DTDs, XML Schemas, and namespaces are often treated the same way as software documentation: In a software project, these are the last things to be completed. However, it is our experience that this is the wrong approach. Because of the way XML can be extended by adding new tags, any discrepancies between the way you thought you defined the data and the way it is actually being used might not appear for some time. After you have defined your data, the next step is to transform it into something presentable. This is where XSL and XSLT play a role.

Extensible Stylesheet Language (XSL) and XSL Transformations (XSLT)

In the preceding section, we showed how data such as a customer can be defined using XML, and we presented the rules that govern how XML is built. The next step is to transform the XML data into a different format—either another XML data format or something presentable such as XHTML by adding layout information.

When you send your customer data to a company that already uses an XML-based application, wouldn't it be great if it were possible to easily transform your `<customer>` data into their `<customer>` format?

For this purpose, a set of standards was defined by the W3C accompanying XML. The Extensible Stylesheet Language (XSL), which has been a W3C recommendation since October 2001, consists of two big parts: XSL Transformations (XSLT), for transforming XML documents, and XSL Formatting Objects, for specifying document formatting.

This is exactly where XSLT comes in. XSLT has been a W3C recommendation since 1999 (which is why it is often thought of as being XSL, when in fact it is only part of XSL). It defines a language that allows the transformation of one XML format into another (or any other text-based format). We will take a closer look at the document formatting aspects of XSL later in this book. For now, we will take an introduc-tory look at XSLT.

Although the following gives you a short introduction to and example of XSLT, this is not an XSLT book. So we suggest you read up on XSLT using the available literature and web sites. Cocoon makes extensive use of XSLT, so you will need some knowledge before you start building your own Cocoon-based application. However, the following gives you enough background to be able to understand our examples.

XSLT is a set of commands for transforming XML. You build up your stylesheet using these commands and then save the complete stylesheet so that it can be used later. The stylesheet is in itself an XML document, so the rules of XML authoring apply to stylesheets as well. Listing 2.9 is a simple example of a stylesheet that transforms the customer data used earlier into XHTML format.

Listing 2.9 Simple Stylesheet Example for the Customer Data

```
<xsl:stylesheet version="1.0"
              xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
              xmlns="http://www.w3.org/TR/xhtml1/strict">
<xsl:template match="customer">
<html>
  <head>
    <title>Our Customer</title>
  </head>
  <body>
    <center>
      <h1>Customer details</h1>
      <p>ID: <b><xsl:value-of select="@id"/></b></p>
      <p>Name: <b><xsl:value-of select="name"/></b></p>
      <p>Address: <b><xsl:value-of select="address"/></b></p>
```

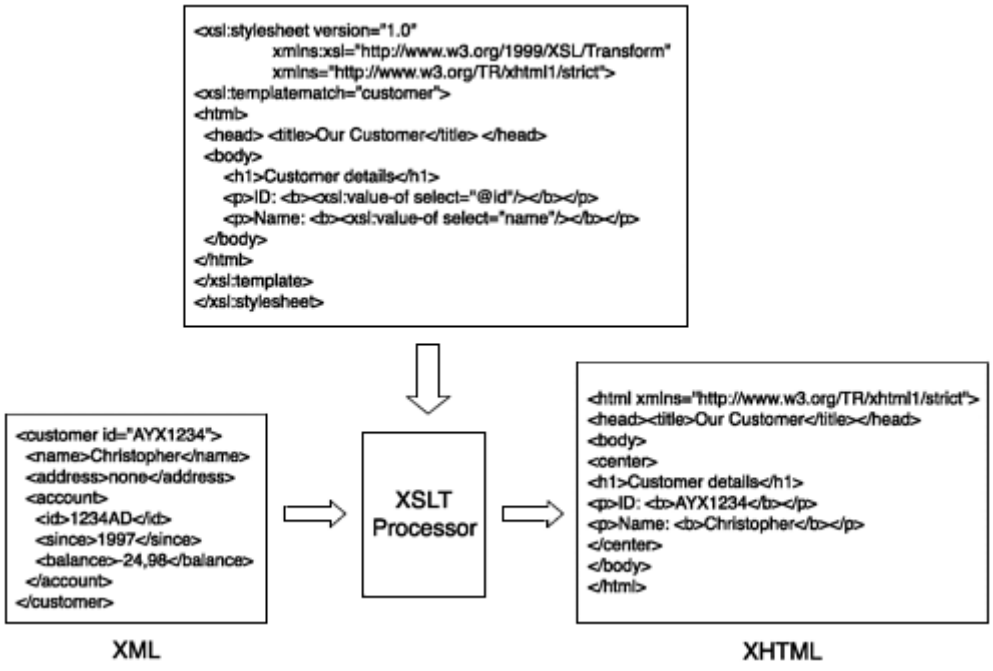
```
     <p>Account since: <b><xsl:value-of select="account/since"/></b></p>
     <p>Account balance: <b><xsl:value-of select="account/balance"/>
     </b></p>
    </center>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

As you can see, XSLT makes extensive use of namespaces. This example contains the namespace that refers to the XSLT definition itself and also the namespace that refers to the definition of XHTML.

Each style sheet starts and ends with `<xsl:stylesheet>`. Inside this tag can be one or more `<xsl:template>` tags. The `<xsl:template>` tag encloses the rules that are to be applied to a particular tag of the source XML document. So that you understand this, we need to briefly explain how the processing of XML with a style sheet works.

Figure 2.1 shows that the XML data is processed by an XSLT processor. You do not need to write this component yourself, because there are already XML components available that you can use in your own applications. We will talk more about them in the next section. The XSLT processor transforms the XML data into a "result" document using the style sheet.

Figure 2.1. XSL processing flow.



To do the transformation, the processor goes through the XML data (starting at the root) and searches for templates that match the tags in the XML data. Whether or not the template matches the XML data is defined by an attribute, conveniently called `match`,

that is attached to the `xsl:template` tag. The example has a match that is triggered when the `customer` tag is found. If a corresponding template is found, the processor goes through all the tags contained in the style sheet inside that particular template. For each tag inside the template, the processor does one of the following:

- If the tag in the style sheet has the XSLT namespace, it is executed.
- If the tag does not have the XSLT namespace, it is output to the result document as is.

The tags in the style sheet that have the XSLT namespace act as commands for the XSLT processor. One command might be to take a specific tag from the original XML data, and another command could make the processor jump to another place in the style sheet to continue processing. In all, you can use many commands when authoring your style sheets. In this way, XSLT does resemble a programming language.

Because our style sheet contains XHTML tags, these are output to the result document unchanged. Inside these XHTML tags, an additional XSL command (`xsl:value-of`) selects the value of a particular tag from the XML data and inserts it into the output. All in all, this transformation results in a complete XHTML document that contains the data from the original XML document.

This is how all style sheets are used to produce a particular output format. This can be a format for presentation, such as XHTML or WML, but it can also be a format needed to exchange information between firms. Therefore, it would be possible to use a style sheet to transform a sample definition of a customer into someone else's.

If every XML application had to build the XSLT processor itself, XML would never have become the standard that it is today. One of the advantages of using XML to define your data format or style sheets is the fact that these components already exist.

Using Standard Components

Up to this point, we have not yet talked about the software you need in order to build XML applications. Because XML is a standard means of defining data formats, it makes sense to use standard software components to access XML data or to process style sheets. Before we take a look at the specific XML components that are integrated into Cocoon, we will provide some background on the most important XML component—the parser.

The XML parser plays a major role in any XML application because it allows the other software components to access the XML data and manipulate it if they need to. XML parsers come in two different "flavors," according to how they allow access to the XML data—DOM and SAX.

The term Document Object Model (DOM) refers to the fact that the parser builds a complete representation of the XML data (often called a *document*) and holds the complete structure in memory.

The DOM is a W3C recommendation that defines programming language-inde-pendent interfaces to access the data in memory. Examples include getting the root node of an XML document, moving through the document structure, and perhaps modifying the content of a specific tag. The current version of this recommendation is called DOM Level 2. In addition to the first draft (DOM Level 1), the second version is namespace-aware. Namespaces allow the definition of an XML format to be unique. This is important, as you saw earlier in this chapter when we discussed the potential problems of someone else already having defined an XML format for a customer that would conflict with yours.

Accessing the XML data via DOM functions is quite simple. Because the parser stores the complete structure for you, there is no need to keep a copy of the data in your own component. However, a major drawback of DOM is that the document representation requires a lot of memory to be available, so DOM does not scale well for large XML documents.

The event-based approach (referred to as the Simple API for XML [SAX]) is completely different. The parser reads the XML data, and each time an element is encountered, an event is fired that the application can respond to. The application component that receives the event can then decide whether the tag and its content are important and processes the data if this is the case. An important fact to remember is that the SAX parser does not hold the XML data as a tree in memory, so any application component will need to do this itself if it needs access to larger sections of the XML document at a later time.

The SAX model is a recommendation not hosted by the W3C, but it has reached the same acceptance. It is also programming language-independent and defines a set of interfaces dealing with the various events occurring during XML parsing. Similar to the DOM Level 2 standard, the current SAX specification, which is also namespace-aware, is called SAX-2.

When going into detail about Cocoon, we will often mention the terms DOM and SAX. We always mean DOM Level 2 and SAX-2 because Cocoon is also name-space-aware. Namespace-aware does not imply that you *have* to use namespaces, but that you *can* use them. As suggested earlier in this chapter, we advise you to define a namespace and a DTD for your XML documents in your productive web application.

Cocoon is based on the SAX model, because it is faster and uses less memory compared to DOM. However, most applications still need to use a DOM-based approach inside certain components, because there are times when the application will need to navigate (and have access to) a memory-based representation of the complete data.

Apart from presenting the XML data to the application, the XML parser can also check an XML document for validity. Using a DTD, the parser can verify an XML document. The parser can detect whether the elements belong to the defined language and whether they are used in a semantically correct fashion. However, for prototyping and testing (and

throughout most of our Cocoon examples), you will probably not want to have your XML document validated by the parser.

The XML parser plays a very important role. Together with the XSLT processor, it forms the base of any XML application. Cocoon is shipped with the XML parser Xerces and the XSLT processor Xalan, both of which are Apache projects.

Xerces: An XML Parser

The XML parser is the component that can transform a stream of XML data into something that can be processed by another application component. The parser provides the interfaces and functions by which the data can be accessed.

Because Cocoon is implemented in Java, the Java API for XML Processing (JAXP) from Sun, a standardized interface for parsers, is very important here. If your Java parser implements this standard, chances are you will be able to drop it into Cocoon and use it instead of the provided parser if you want to. A JAXP-compliant parser implements both the DOM and SAX parsing models.

Adhering to a standard interface is important when you look back at one of the key requirements presented in the Chapter 1. If your customer is running a Cocoon-based solution and someone creates *the* high-performance, low memory footprint, coffee-making XML parser (OK, perhaps not the coffee-making), your customer would expect you to be able to drop that parser into Cocoon.

Although a variety of different XML parsers are available, we will introduce you to the XML parser used in Cocoon—Xerces. Xerces is itself an open-source project run under the Apache umbrella. Xerces originally started out as an IBM XML technology named XML4J and was then donated to Apache in 1999. This is something seen quite often in open-source projects: A company starts developing an application or component and then donates it to the open-source community.

The Xerces (named after a blue butterfly) parser supports the JAXP model and therefore both the DOM and the SAX models. It is available in a variety of implementations. Cocoon uses the Java implementation.

So, when you install Cocoon, you are also installing several other components, such as Xerces. In most cases, and especially when building your first applications with Cocoon, you will probably not encounter the parser on its own. This is one of the great advantages of using Cocoon as a base for your XML solution:You do not have to worry about having to program the XML parser. Cocoon does it all for you.

In order for Cocoon to be able to transform the XML data into an output format, you also need an XSLT processor.

Xalan: An XSLT Processor

Another key Apache project is Xalan. Named after a rare musical instrument, Xalan is the component that allows XML data to be processed with XSLT style sheets.

The Xalan project originally started out as a software component written at Lotus (LotusXSL) and was then donated to the Apache Software Foundation in 1999.

Xalan conforms to the JAXP standard, which deals with not only XML parsing but also with XML processing. The JAXP standard defines some basic interfaces to transform XML documents using style sheets.

Xalan also provides an XPath processor that can be used without XSLT to access XML data based on queries. The XPath engine is important whenever you deal with DOM. You can then use the XPath engine to access distinct nodes in your XML document for searching or modifying. Think of XPath as the SQL of the XML world.

Cocoon uses Xalan for style sheet processing. It is not the only solution that makes use of these freely available Apache components. Many other companies and products use them.

Key Components

Xalan and Xerces are two components that are widely used in XML-based architectures. They are found in specific customer solutions or inside standard applications such as Internet application servers. Here is a brief list of companies that use one or both of these components (based on poll results taken at JavaOne 2001 and posted to the general@xml.apache.org mailing list in October 2001):

- Lutris: Enhydra, EAS
- Software AG: Tamino products
- BEA web server
- VeriSign: trust services
- Iona: web server
- ATG: Documentum
- Orion application server
- Open Market web server
- Attachmate solutions
- Nuance VoiceXML
- Computer Associates

Just as these commercial companies profit from new versions of Xalan and Xerces, so does Cocoon. Instead of the efforts of many being split between several different versions of the same component, these efforts are combined to create one version of each component. The companies that then utilize these components can free up their resources for other things.

Because of the way Cocoon integrates these components, it removes the chore of having to integrate them into an application yourself. You can therefore concentrate on building the XML application.

Building XML Applications

Now that we have talked about what components are available, we will look at why XML applications are well-suited to meeting the requirements we discussed in the first chapter, such as multichannel publishing, personalizing the information we want to display, and integrating other data sources.

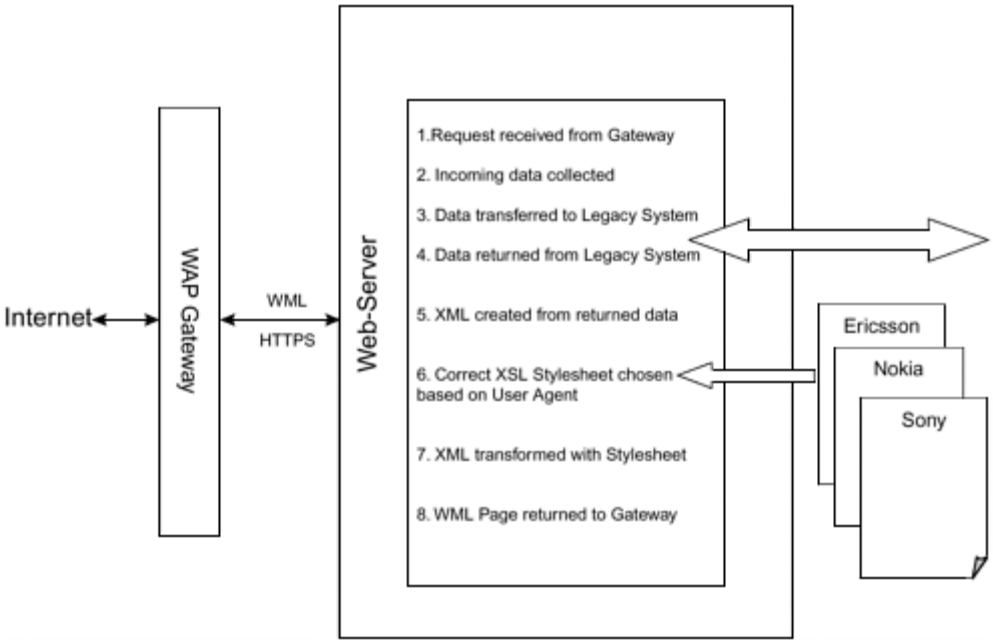
Multichannel Publishing

As you saw in the customer example earlier in this chapter, XSL enables you to publish XML data in the format of your choice, such as XHTML. The same XML data can also be formatted into, say, Wireless Markup Language (WML) using the correct style sheet.

Due to this flexibility, it is quite easy to see how it would be possible to build a simple application that can respond to certain data, such as the type of browser, and then select the correct style sheet for that device based on a given configuration. Using this concept, it is also possible to add multichannel publishing to an existing application.

Figure 2.2 shows how you can add multichannel publishing capabilities (using XML and XSL) to your existing application. This is exactly how our first WML application, written in ASP, worked (as we discussed in the first chapter). Using this concept (and the Microsoft versions of the XML parser and XSLT processor), we were able to extend a running solution and allow it to publish to a mobile phone or PDA.

Figure 2.2. Integrating multichannel publishing.



In our example, the request came into the web server by way of a WAP gateway. This is the program that converts the WAP protocol to Internet-oriented HTTP. As soon as the request reached the ASP script, the data was extracted and sent to the legacy system using existing components that were already part of the running application. So the new ASP script basically wrapped around those components to convert the returned data into WML. When the data, such as an account balance, was returned, the ASP converted that into an XML format. Based on the browser-specific piece of information called the user agent, which was mapped to a specific vendor and device, the ASP script selected the correct style sheet to be used and then passed the data and the style sheet to a Microsoft component for processing. The component transformed the data into WML, which was then returned to the mobile device by way of the WAP gateway.

As new mobile devices became available, the only change necessary was to add to the system a new style sheet to support that device. Of course, a default style sheet was applied if an unknown user agent was encountered.

This example of adding multichannel publishing to a given application shows how it is possible to flexibly format the same data into a specific output format depending on a certain piece of information. However, a successful XML solution must also allow for the easy integration of various data sources so that content is available for the various publication formats.

Integrating Data Sources

Integrating diverse data sources is one of the key functions of a middleware solution. Until the advent of XML, middleware solutions had to be able to send and receive proprietary, often binary, data to a variety of systems.

Many systems such as databases and even mainframes have already made the transition to Internet-based XML architectures. They offer XML interfaces to their data and therefore allow for easy integration into solutions such as Cocoon.

XML Services

Companies that provide a data-oriented service use the Internet to allow access to that data. These services, ranging in size and color from a complete news feed to a local weather service, are commonly used inside portals to provide the content.

These services increasingly publish their data in an XML format, making it extremely easy to integrate the data into your own application. You'll read more about this when you build a sample news portal application starting in Chapter 5, "Cocoon News Portal: Entry Version."

Some of these formats are even standardized, meaning that a solution built to display the news from one source can present the headlines from a completely different source with no changes needed. Because both news services use the same format (such as News Markup Language [NewsML]), the same style sheet can be used to format the data into HTML.

Commercial services such as Reuters offer news in an XML format at a cost. Noncommercial offerings such as Moreover.com provide news headlines and articles that can be integrated into noncommercial applications and sites for free. Companies such as the German company Onvista provide real-time stock quotes (at a cost) in XML for access over the Internet. Many banks are already integrating these services as part of their offerings to their customers.

As real-time information such as weather, quotes, and news becomes more popular, the number of these services will grow and become one of the key sources of XML data over the Net. In addition, every corporation has a lot of information already stored in databases. So they are another key data source.

XML Databases

Another popular source of enterprise data is the database. Most database vendors offer XML interfaces to their systems, allowing the XML data to be mapped to the underlying relational database model they use. This form of integration is becoming popular because the existing database does not need to be replaced. Instead, it can be integrated into a modern middleware architecture using the new XML-based interface.

On the other hand, we are also beginning to see vendors bring out XML databases, meaning that there is no direct mapping to the traditional table view of other databases. Databases that offer this type of support lend themselves to integration into XML middleware solutions, because they often allow XML concepts such as XPath to be used to access the data directly from the database.

Companies such as Microsoft and Oracle have extended their databases and application programming languages to provide for XML data. Although they don't change the base structure of the database, these extensions allow the easy integration of existing databases into an XML architecture.

The Tamino database, a product of Software AG, is a commercial native XML database. This means that XML documents are stored in the database without first being mapped to tables. XML query languages such as XQL can be used to access the XML inside the database, allowing the XML architecture to be extended from the middleware into the database. The open-source XML database Xindice is another example of a native XML database that allows access to XML data via a programming API, command line, or CORBA.

Apart from having information in databases, most companies have some form of back-end system that is also a source of data for a middleware solution.

Mainframes

Although we traditionally think of a mainframe as perhaps not being the first place you would find an XML solution, XML interfaces are often available. Combined with the flexibility of data transport over HTTP you have an ideal data source that can be integrated into an XML architecture.

Vendors of mainframe software such as SAP have extended their products to provide access via XML data and standard Internet protocols such as HTTP. Corporations that traditionally wrote their own programs, in such programming languages as COBOL, are now offering the same data in XML structures. Access to these systems is becoming more standardized through the use of messaging systems such as IBM's MQSeries and the utilization of standard protocols.

This allows the different applications running on the mainframe to be accessed via the same interface. This greatly reduces the cost of integrating these systems. Often corporations will already have written web applications that access data on the mainframe, so a common middleware solution also needs to integrate these applications.

Integrating Applications

Integrating an application is more difficult than just connecting to a given data source, because in most cases the application already contains a presentation layer and logic that controls the application's flow.

Depending on the type of application, there are various ways data sources can be integrated into an XML-based architecture. If the application is very "data-centric" (such as a report generator), the data sources can be separated from the existing application and integrated into a new XML-based application.

If the application exists as, say, an HTML-based web solution, it is also possible to integrate these types of applications into an XML system. The HTML can be converted to the XML format XHTML, manipulated using style sheets, and then displayed with the same look and feel as the new applications, written using Cocoon.

Chapter 11, "Designing Cocoon Applications," introduces building of actual applications with Cocoon and describes some of these concepts in more detail. Because of the amount of data and applications available, it is important that the XML middleware allows the presentation to be personalized.

Personalizing Information

It goes without saying that everyone has a personalized view of the world. Our brains are trained to take in the information that is important to us and to ignore what is not.

This is one of the reasons you are sometimes so overwhelmed with what you see in the browser window when surfing the web, especially when you have no way of selecting the information you are interested in. Apart from the content of the page, factors such as the color or the positioning of the data can influence whether you linger on that particular page or move on to somewhere more interesting.

It is becoming increasingly important to personalize the information you present to someone who is using your application or viewing your web page. Apart from allowing this to happen based on criteria you might already have stored somewhere, you also need to allow personal configuration by the user. He might never return to your web page if you chose blue as the background color and he hates blue.

Using XML to design your data structure and style sheets to format that data into the desired layout makes it a lot easier to allow for the personalization of the data you present.

You can provide different style sheets for the various browsers. You can design each style sheet in a way that allows it to interpret data such as the browser's language or a color preference stored in your internal customer information.

The advantage of this method is that you do not need to alter the data structure just because the user prefers yellow over the blue you chose. He might prefer to surf your web site with his mobile phone instead of his PC—and you can provide for that as well by just changing the style sheet you use for presentation.

In this book, you will build a sample application. This application, as well as some of the examples we will show you, allows the personalization of information using Cocoon. However, none of the advantages we just mentioned would be any good if the solution you built was locked in to a particular operating system or IT environment.

Platform-Independent Solutions

One of the major advantages of XML is the fact that it is not bound to a single platform, nor does a single vendor control it. XML data can be read without the aid of a machine. It can be interpreted by applications written in different programming languages and running on completely different systems.

Add to this the availability of components for nearly every platform you can think of, and the transport of XML data over networks using common protocols such as HTTP, and you have a solution that is truly independent.

The integration of XML interfaces into a growing number of systems is increasing this independence. Now, due to new protocols such as the XML-based Simple Object Access Protocol (SOAP), the middleware can connect to systems over the Internet that might be far removed from its location—and it can do this in a very flexible manner.

Flexibility

When using XML, you are not only independent, but also very flexible. For example, if your application (or one component) is interested in only customer names, for example, then you can easily extract this information from the XML document.

For example, using a style sheet, you can filter the XML document and obtain a new XML document containing only the customer names. Or you can use the XPath engine to get the names using a DOM representation.

In contrast to other formats, especially binary formats, you don’t have to worry about all the other information contained in the document. You can simply ignore it.

Ignoring information reveals another important feature of XML: extensibility. Imagine that you first designed your customer with only a name, mailing address, and email address. Your web application runs well on top of this model until you decide to store the fax number of each customer to send him “printed” bills.

If you had not chosen XML but a proprietary format, you would have to extend the format and also all the components dealing with your format. If you exchanged this format with other companies, they would have to update their components, too. And, of course, you also would have to deal with the old model’s not having a fax number.

With XML this is much easier. You just add an optional fax number element to your document definition. The component requiring the fax number can easily test whether it is available, and all other components still run unchanged. So it is easy to build and extend your XML solution to meet changing requirements.

Building an XML Solution

Now you have all the basic information on XML and XSL. You know there are components available that let you build a solution that will give you the flexibility to publish to nearly any format you want.

But what about the components that integrate that SQL database into the same architecture? Where is the component that lets you do user authentication in your portal against an LDAP directory? These are key things you need when building a complete solution.

Granted, XML is becoming increasingly popular as a way of integrating data sources, either new or old, into modern XML middleware solutions. And, as more and more traditional solutions migrate to an XML-based model, it will become increasingly easy to integrate them.

However, what about all the customers who do not yet have an XML-based platform, or XML-based mainframe architectures? The middleware solution needs to be able to handle these as well. It needs to be able to handle them in a way that lets them be replaced without having to change the middleware solution.

And what about Xalan and Xerces? How do you integrate them into your application?

Obviously it would be possible to do all this yourself. And indeed, for smaller, dedicated applications, it might well be a better alternative to integrate the needed component directly into your application. The first XML application we wrote was an extension to a given application. It used XML and XSL to publish given data into the WML format for mobile phones. It made sense to just use the XML components, because the other parts of the application were already in place. However, this solution also limited the flexibility of XML and XSL to just this one area, so all the other parts of the application remained unchanged.

When it comes to building complete middleware solutions such as a portal, a web site based on XML and XSL, or a reporting system that can publish to PDF or WML, there is an easier way than integrating all the components yourself. Apache Cocoon combines all the needed components into a ready-to-run architecture and therefore removes the need to integrate the components into your own application. Using Cocoon, it is even possible to build complete XML applications without writing a single line of Java code.

Apache Cocoon

Apache Cocoon is an open-source XML publishing framework that is being developed by a group of enthusiasts all over the world. Although Cocoon started off as the project of only one person, it has grown into an industrial-strength framework for XML applications. Its major advantage is that it is free. This means that you do not have to pay anything to use the software or obtain the source code. Cocoon comes with all the

components you need to build different types of XML applications. You can also extend the solution with your own components if you need to.

The Project

The Apache Cocoon Project (http://xml.apache.org/cocoon) was started in 1998 by Italian student Stefano Mazzocchi because he was frustrated by HTML's limitations when redesigning the Apache web site. He decided to use XML and XSL as the basis of the new software he wrote because it would allow him to separate the different parts of designing a web site (content, layout, site architecture, and logic) between several people without their interfering with each other.

Instead of writing all the necessary components in the Cocoon project, Mazzocchi decided to use software that already existed or was being developed. Because Cocoon uses many other software components from different Apache projects, such as Xalan and Xerces, it also influences these projects and is itself influenced by them.

During the last part of 2001, about 2,000 people subscribed to the Cocoon project mailing lists. Cocoon has a large following, even though the newest Cocoon version has only just been released. Several large firms are helping develop Cocoon. This is a sign that the project has a lot of strength and will hopefully continue for a long time.

The core Cocoon development team consists of about eight people. These developers have the right to check in new code and, in doing so, change the code base. Many "fringe" developers support the project by submitting new components or helping with bug fixes.

All the developers work for free to provide new software, documentation, and support. The complete Cocoon software is available under the Apache Software License, one of the most common open-source licenses.

Open Source

The term *open source* was coined in 1998, shortly after Netscape released the source code of its Netscape browser. Before that, the more common name for freely available source code was "free software." Even though the name changed, the ideals remained pretty much the same. The goal is to provide the source of a particular application or module so that people can modify that code and, in doing so, add value to the software.

The more governing factor when it comes to open source is that of licensing. The license under which the source is released governs what can be done with the software. So this is what you need to consider when deciding whether a particular project or software is right for you.

The Apache Software Foundation authored the Apache Software License, which governs all projects that sail under the Apache flag—or, rather, feather (see

http://www.apache.org). This includes the Apache Cocoon project. The Apache Software License basically allows you to do what you want with the source code of a particular component or project. You can use that code in your own product and still sell that product under a commercial license.

Being able to do this is very important for many companies that are interested in using Cocoon inside a commercial environment. Although this might seem to contradict the open-source movement, remember that many companies support open source by allowing developers to work on open-source projects as part of their paid time.

Using Cocoon

Using the available components we have discussed in this chapter will help you build your own application that can harness the power of XML and XSL. However, you still must do a lot of integration work to get the components into your application. What is really needed is a complete framework that

- Is completely built on XML and XSL
- Is not limited to a specific operating system
- Allows easy integration into existing Internet architectures
- Encapsulates the necessary components, removing the need to integrate them into an application architecture yourself
- Allows the integration of standard data sources such as databases and external HTTP servers
- Allows data to be presented in a personalized way
- Offers an extensible architecture, allowing the integration of additional components you might need to build for your environment

Summary

This chapter looked at why HTML is not an ideal format for designing the “machine web” and how the XML family of standards has allowed data (and layout) formats to be defined in an open and cross-platform way. XSL lets you design a way to publish your data in various formats in a very flexible manner. Because it is standardized, XML is supported by components that can be integrated into new applications or as extensions to given solutions. An XML-based application can meet the many challenges facing today’s Internet solutions. The Apache open-source project Cocoon provides an extensive framework for XML applications. Cocoon contains the basic XML components and also provides ways to integrate various data sources and control how the data is published in a specific format.

The first thing you need to do to use Cocoon is install it. You will do this in the next chapter.

Chapter 3. Getting Started with Cocoon

Now it's time to get your hands dirty and actually install Cocoon to find out what it contains. Installing Cocoon is actually very easy; this chapter contains all the details you need.

For simplicity, we assume that you will be installing Cocoon onto the same system on which your browser is installed. This means that, in effect, the system is then both server and client at the same time. In case the setup is different (such as if you want to install Cocoon on a standalone server), the address you give, such as to access the samples, needs to be adjusted from `localhost:8080` to the server's actual address. However, this is the only difference from installing everything onto one system.

We will take a look at what is needed before you start the actual installation. Then you will see how to install the servlet engine and then Cocoon. After everything is running, the included samples provide some insight into what you can actually do with the software.

Prerequisites for Installing Cocoon

For most installation scenarios, there are only two prerequisites for installing Cocoon.

First, a Java JDK must be available on the system. In writing this book, we used version 1.3.1 of the Sun JDK. The JDK can be downloaded from the Sun web site (www.sun.com). In case you already have an older version of the JDK installed, you need to make sure that the version is at least 1.2.2.

The most common installation environment is to run Cocoon as a servlet in a servlet engine. If one is already installed on the target machine, you can skip the sections on using Apache Tomcat. If no servlet engine is running, follow the step-by-step explanation of how to obtain and install Apache Tomcat.

If Cocoon is being installed on a UNIX system, there is an additional prerequisite: X-Windows must be installed in order for the installation to work as we describe here. If you don't have X-Windows, have a look at the Cocoon FAQ (on the CD or online at

http://xml.apache.org/cocoon/faq.html) to see how you can get Cocoon running on a "headless server."

Step-by-Step Instructions

This section provides step-by-step instructions on installing Cocoon into an Apache environment. We will be using the Apache Tomcat Servlet Engine. The required components can be downloaded from their respective web sites or copied from the CD.

Using Apache Tomcat As the Servlet Engine

The first step in getting a running version of Cocoon onto your system is to install Apache Tomcat, the servlet engine provided by Apache. As you've seen in previous chapters, and as is most common in today's Internet architectures, an Internet server normally receives incoming HTTP requests via a web server. Depending on the web server's configuration, the requests are served directly by the web server itself or are passed to a servlet so that the response can be generated dynamically.

A common setup for Cocoon is to have the web server serve the static content (such as images) and to have Cocoon process the requests to generate the dynamic HTML documents.

For this installation guide, we will stick with the Tomcat-only installation. Refer to the Tomcat documentation available on the web if you want to connect Tomcat with a web server. This method of installation means that you will send your requests directly to Tomcat. You can do this because Tomcat comes with its own little web server. This makes life much easier, especially when you're just starting out with servlets and Cocoon.

Obtaining Tomcat

You can download Tomcat from the Apache web site or copy it from the companion CD. The Tomcat home page is located at http://jakarta.apache.org/tomcat/index.html. To obtain a version, follow the links listed there until you arrive at a page where you can download a binary version of Tomcat. Refer to the documentation there to determine the exact version of Tomcat you need.

This section covers the installation of Tomcat version 3.3a. It should be easy to adapt the following steps to different versions as they become available. However, we have provided this version of Tomcat on the CD so that it can be used as a starting point. The steps given here use the binary file jakarta-tomcat-3.3a.zip. We have provided additional binary formats on the CD, and more are available from the web site.

Installing Tomcat

The next step is to unpack the downloaded file into a directory. Use the root directory of the Windows system (C:\) and unpack the zip file below that.

As a result of unpacking the zip file, you now have a directory called C:\jakarta-tomcat-3.3a with several subdirectories.

Setting Up the Environment

You now need to configure your environment so that, for example, the JDK can be found when Tomcat is started. To do this, you need to set some environment parameters. How you do this depends on what system Tomcat is running on (Windows, UNIX, Mac OS, X Window).

The environment variable `JAVA_HOME` needs to be set up to point to the root directory where the JDK was installed. Check to see which directory contains the JDK, and then enter the following in a shell or a DOS window. Refer to your operating system's documentation for the exact syntax.

```
set JAVA_HOME=c:/jdk131
set PATH=%JAVA_HOME%\bin;%PATH%
```

It's a good idea to enter these lines into a script or batch program so that you don't have to enter them each time you want to start Tomcat.

Starting Tomcat

The \bin directory of the Tomcat distributions contains scripts and batch files that can be used to start and stop the servlet engine. So as soon as the environment has been set up, you can start Tomcat by entering the following:

```
[Unix] bin/startup.sh
[Windows] bin\startup
```

The first time you start Tomcat, it takes longer before the servlet engine is ready to process any requests. If everything goes as planned, the following output is logged to `stderr`, which is by default the window Tomcat is started in:

```
2002-02-19 13:04:24 - Http10Interceptor: Starting on 8080
2002-02-19 13:04:24 - Ajp12Interceptor: Starting on 8007
2002-02-19 13:04:24 - Ajp13Interceptor: Starting on 8009
```

If this output appears, you know Tomcat is running. You can then access the start page by entering `http://localhost:8080` into your browser. For most installations, the default Tomcat configuration works as described. Depending on the setup of the system you installed Tomcat onto, there might be situations in which you need to alter the port

number (8080 in this case) by changing the Tomcat configuration. This is explained in the Tomcat documentation.

As soon as your request is processed, you should receive the Tomcat start page as HTML. So, now your servlet engine is running and you can install Cocoon.

Installing Cocoon

Installing Cocoon is as easy as copying a file into a directory, because that's all there is to it. We have provided the Cocoon 2.0 distribution on the CD; everything you need is contained there.

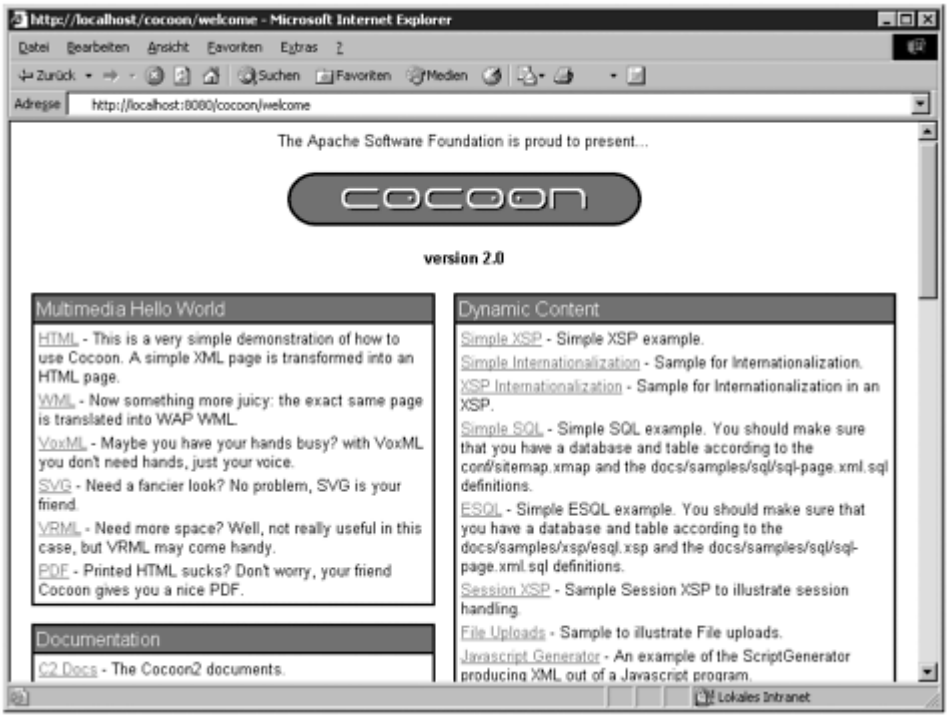
Open the binary distribution (for example, the file cocoon-2.0-bin.zip) and extract the cocoon.war file to the webapps directory in Tomcat. Other files with a WAR extension are already in that directory.

Next, stop and then restart the servlet engine. You can do this by first calling the shutdown script (or batch file) in the \bin directory and then calling startup again (as described in the preceding section).

Tomcat recognizes the new file and unpacks it below the webapps directory. After this has happened, Cocoon is installed and ready to be used.

If you then use your browser to go to `http://localhost:8080/cocoon/welcome`, you should see the Cocoon welcome page, as shown in Figure 3.1.

Figure 3.1. The Cocoon welcome page.



Congratulations! Cocoon is installed and running. As you can see from the figure, there are some interesting things you can try out immediately. You will see some examples after we look at problems that can occur when you install and view the Cocoon samples.

Common Problems and Finding Help

Unfortunately, not every installation goes as planned. Even though the installation is not too difficult, a variety of potential problems can occur. If an error occurs (for example, the browser does not show what it should be showing), the first thing to check is any additional output that Cocoon makes to either the console or the logs.

The logs can be found in the subdirectory \WEB-INF\logs. Three different logs are contained in that directory. Each log contains messages that are specific to a certain area of Cocoon:

- `components.log`: Contains certain component information (XSLT processor/ memory store).
- `root.log`: Contains servlet errors, and certain Avalon components log here.
- `cocoon.log`: Serves as the main log file. It contains the most important information.

You can configure the detail of information contained in the logs, as you will see in Chapter 6, “A User’s Look at the Cocoon Architecture.”

Most of the difficulties encountered when you install Cocoon can be solved quite easily. The following sections list the most common problems and provide some pointers on solving them.

Because the Cocoon distribution we reference in this book has been available for some time and comes complete with functioning samples, most problems at this stage are due to conflicts in the system configuration. We provide some information at the end of this section on obtaining more help if none of the following clears up the encountered problem.

HSQL Errors

The HSQL database is integrated into Cocoon. It starts automatically when the servlet engine starts Cocoon. If there are any problems with starting the database, the messages are logged to the console. Here are a couple of tips that help you get around error messages concerning HSQL:

- Delete the file db.backup, which is found in the subdirectory \WEB-INF\db.
- Make sure that there are not several instances of HSQL running at the same time. If there are, alter the configuration so that each instance listens on a different port.

If none of this helps, it might be a good idea to remove the entry for HSQL from cocoon.xconf and see if this clears up the problems. Of course, this then means that none of the database examples will work.

Communication Between Cocoon and the Browser

After Cocoon is installed and has been started, you use a web browser to view the documents. Several potential problems can occur in this area:

- Make sure the browser can connect to the server where Cocoon is running. This might mean altering the proxy settings in the browser.
- The browser might cache the result of the request. If so, you need to refresh the browser in order to get the correct output.
- An error is often the result of a typing mistake. Unfortunately, Cocoon does not make this very clear in its error messages. The first thing you should check is that you entered the information into the sitemap correctly. Then check the XML file and the XSL file.
- Some browsers have problems displaying an HTML page after directly viewing XML data (and vice versa). In this case, opening a new browser window should fix the problem.

Hopefully this information will solve any problems you encounter using a browser to access Cocoon.

Xerces Conflicts

Cocoon comes with its own version of the Apache XML parser Xerces. Some servlet engines also have their own version of the parser. This can cause conflicts when you start Cocoon.

The basic solution is to make sure the servlet engine also uses the Cocoon version of Xerces. Often you can do this by replacing the servlet version with the Xerces version from Cocoon's WEB-INF\lib directory.

Using a Different Servlet Engine

Unfortunately, a variety of servlet engines are available in various versions. This does not make it easy to document how to install a servlet such as Cocoon. Even worse, there can be quite a few differences between versions.

As Cocoon becomes adopted by more and more people, it is being installed into new servlet engine environments. Luckily, a comprehensive guide to installing Cocoon into different servlet engines is available in the Cocoon documentation on the CD or online at http://xml.apache.org/cocoon/installing/index.html.

What to Do if Something Is Still Wrong

If none of the preceding information helps, and Cocoon is still not functioning correctly, there are several ways of getting help:

- Check out the documentation provided on the CD for information that might help (in particular, the installation guide and FAQ).
- Check out the same documentation from the Cocoon web site. It might be more up-to-date than the CD version.
- Check the archives of the Cocoon mailing lists to see if someone else has had the same problem.
- Post a question to the Cocoon user list to see if someone can help (but check the archives first!).

All the web links for these sources of information are contained in Appendix C. If everything works as planned, you can look at the samples that come with the installed version of Cocoon.

Accessing the Samples

Your browser should be showing the Cocoon welcome page. As you can see, it is full of examples that give you a good idea of Cocoon's capabilities. Let's take a look at some of the examples that can be selected from the welcome page.

Multimedia Hello World

The first block of examples shows the various ways of presenting "Hello World." There are examples that present these words in HTML, WML, PDF, and other more exotic

formats. Depending on the browser you are using and the software you have installed on your system, you might not see all the examples in your browser.

Some of the examples, such as HTML and PDF, work on most systems. For other examples, such as WML and VoxML, you might need to install additional software before you get the expected result.

Documentation

This section allows you to access the complete Cocoon documentation. This is not so much a sample as a potential lifesaver. We have also included the Cocoon documentation on the CD so it can be referenced without requiring a running Cocoon installation.

News Feeds

This section shows some examples of fetching XML news feeds over the Internet and then formatting them into an HTML layout. This is similar to what we will describe later in this book when you build your own news portal.

Dynamic Content

These samples show how content that is not static can be generated using different methods, such as by accessing a database or using JavaScript or XSP to generate the content. Some of these examples require additional components to be installed first.

Sample Forms

The Cocoon installation comes complete with the open-source database HSQL and some sample tables. The forms in this section show how data in different tables can be manipulated using Cocoon.

System Pages

When Cocoon is running, it is important to be able to obtain information about the system. These examples show what is possible in this area. They allow you to see details such as which version of the Java virtual machine is running on the server and how much memory is available.

One example that you should look at closely is the sample that generates an error page. It shows how Cocoon formats and presents any error that can occur when the document pipeline is processed. This document might become a common sight as you start adding your own pipelines.

Completing the Sample Tour

This completes our look at some of the examples that come with Cocoon. We suggest that you also take a look at the other examples to get a feel for what you can do with

Cocoon. Don't worry if it is not yet clear how all this works. We will delve into the details as we progress. We will also find out how Cocoon works internally, what else you can do with it, and how you can perhaps extend the framework with your own components. All this information is contained in the following chapters.

But before we move on, one additional piece of information is perhaps best introduced now, even though it is something you'll probably better understand after finish-ing this book and trying out the examples. This is the information you need in order to obtain a newer version of the software than we have provided on the CD. We used version 2.0 of Cocoon when writing this book. By the time you read this book, a newer version of the software is sure to be available. It would be pretty mean of us not to provide a guide on how to get the newer version.

Obtaining a Newer Version of Cocoon

The version of Cocoon on the companion CD is the released version 2.0. Because Cocoon is constantly being developed, newer versions containing additional components or changes will become available and can then be downloaded from the Cocoon web site.

Depending on whether a newer binary version of the software is required or whether you perhaps want to build your own version of Cocoon from the source, you need to look at the two different ways of finding what you need.

Downloading Binary Releases of Cocoon

The Cocoon project currently does not yet support the idea of "daily builds." This term is used in other projects to describe a daily binary release of the software. This release is commonly generated automatically, and the binary version of the project is then copied to a location that can be publicly accessed.

Binary versions of the Cocoon software are limited to actual releases. This means that it is possible to download a version of the software that the project team has put through certain tests and then decided that it is stable enough to be publicly announced. New versions are announced on the Cocoon mailing list.

After the software is announced, you can download it from the Cocoon web site or from any mirror site. It is easy to obtain a released version.

Navigate to http://xml.apache.org/cocoon/dist/ and download one of the files there. Basically, the software on the web site is packaged in the same manner as that on the CD version. So it should be easy to install a new version of Cocoon, because we provided a description earlier. It is best to delete an older version before deploying the new WAR file from the downloaded distribution.

Remember that some things might have changed in a newer version of Cocoon. Therefore, you should study the documentation that goes with the release to find any differences from the version we used for this book.

If you are feeling more adventurous and you want to build an up-to-the-minute version of Cocoon, you need access to where the source code is stored.

Building Your Own Version of Cocoon from Source

Before we explain how to download and build the newest version from source, here are some words of warning:You should do this only if you have extensive knowledge of Cocoon and detailed Java know-how. In addition, the version that can be downloaded might not be compilable or even function properly.

If this scares you off from trying, you might want to move on to Chapter 4, "Putting Cocoon to Work," where we look at using Cocoon's various concepts and components to build the first examples. However, if you are feeling brave, read on for details on the first steps you can take to become a part of the Cocoon project. The project is always looking for new committers, and you need to be able to compile the newest version of Cocoon in order to become one.

In order to be able to compile the Cocoon source, you need to have a JDK set up correctly. You must make sure that the environment variable `JAVA_HOME` points to the JDK's root directory.

Access to the Cocoon source is provided by way of CVS (Concurrent Versioning System). The CVS server is hosted by Apache. You need a CVS client in order to be able to access the server and download the source. More details on CVS can be found at http://www.cvshome.org/.

Quite a few different CVS clients are available. Which one you use depends on the operating system and personal taste. For the following explanation, we assume that the command-line version of a CVS client is being used and that it can be started from the command line. First you need to open a shell or command-line window, navigate to where on the local drive you want the source to be installed, and then type the following commands:

```
cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login
(The password is: "anoncvs")
cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic -z3 checkout
     xml-cocoon2
```

This creates a directory called xml-cocoon2 where the Cocoon source is checked out to. The actual checkout process can take a while, and the Cocoon server sometimes suffers from heavy loads. When the checkout is complete, all the Cocoon source is on the local drive.

In order to update the source to a newer version at a later date, you need to first change to the xml-cocoon2 directory and then enter the following:

```
cvs -z3 update -d -P
```

This command updates the previously checked-out source.

Next, you need to compile the source. To do this, you need to enter a command in the shell window:

```
[Unix] ./build.sh -Dinclude.webapp.libs=yes webapp

[Windows] .\build.bat -Dinclude.webapp.libs=yes webapp
```

If it's successful, the compilation process creates a cocoon.war file in the subdirectory \build\cocoon. This file can be deployed as described at the beginning of this chapter.

Some of the components in Cocoon are compiled only if additional third-party libraries are available. More details on this and additional information on building Cocoon from source can be found in the installation document on the CD or online at http://xml.apache.org/cocoon/installing/index.html.

On We Go

Hopefully, you now have Cocoon up and running on your system. You might want to take some time and explore the examples contained in the distribution before moving on to the next chapter. There you will see how Cocoon works and how you can use the different components and concepts to build your own examples.

However, if you are like us, you will probably want to go straight to the next chapter and get started. That's fine too. It's your book, after all.

As you will see, Cocoon is capable of many different things, such as publishing XML data in various formats, integrating data from databases, integrating scripting languages, and accessing external data sources such as news feeds. As you move through this book, you will build sample applications that range from the standard "Hello World" example to a full-blown news portal, so there's lots for you to explore. Let's get started!

Chapter 4. Putting Cocoon to Work

Now that you know how to install and start Cocoon, it's time to start building your first XML/XSL applications with it. However, before we get into the details of how to build, say, a personalized picture gallery on the web, we first need to explain in detail how Cocoon works and what components and concepts you can use to build these applications. But you don't need to start your Java programming environment. You don't need to develop any new components to get the examples in this chapter up and running. As installed, Cocoon already has many components you can use, so there is no need to develop any new ones yet. You will be doing that in Chapter 9, "Developing Components for Cocoon." But for now, you can use the components provided.

We have split the description into two parts. This first part contains an overview of the Cocoon components and concepts you will use the most to build Cocoon-based applications. The second part is in Chapter 6, "A User's Look at the Cocoon Architecture." It contains more-advanced components and architectural concepts. After we have introduced these components and concepts, we will provide some examples that show how to put these components to work. These examples range from a simple "Hello World" web page to the personalized picture gallery we mentioned before.

In all our examples, we assume that you have installed Cocoon as we described in the preceding chapter. To repeat the most important details, you installed Cocoon into a context called cocoon, and your servlet engine is accessible under the name localhost and via port 8080. A request in the form of `http://localhost:8080/cocoon/document` is therefore routed to Cocoon. If you have a different configuration, you need to adapt the examples in this chapter to your environment.

Before writing your first Cocoon application, you need to understand what makes it so different from other solutions, such as a web server that reads a static HTML file from the filesystem and returns it to the browser. Because the Cocoon concept is completely different, we need to look at Cocoon as a whole before examining the details.

Cocoon: The Big Picture

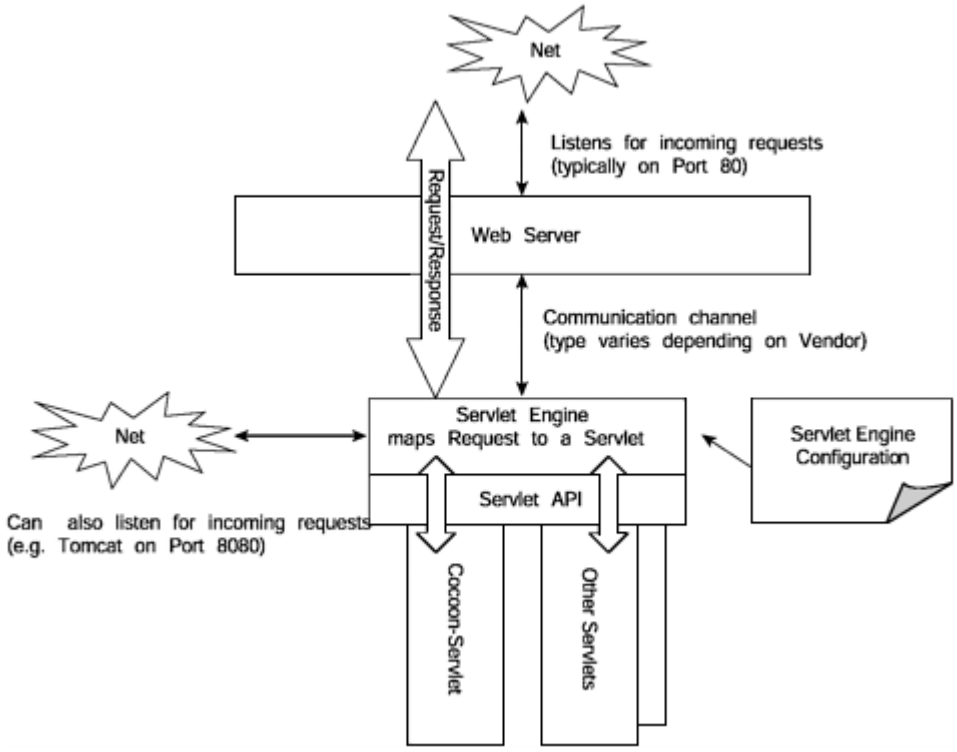
Before we get into the details of the components contained in Cocoon, it is important to understand the Cocoon architecture from a higher level. Also, now is the time for us to define what we mean when we talk about “Cocoon.”

The Cocoon software is made up of many components. Some of these components were developed inside the Cocoon project, and others are external components, developed elsewhere, that have been integrated as part of the larger software bundle you installed in the preceding chapter. Apart from the software, you also installed an application that contains the standard Cocoon examples, the documentation, and all the necessary configuration files.

It is important to remember that these are two separate things. If you build your own application using the Cocoon software, you don’t need the Cocoon sample application, and you also probably don’t want to put the Cocoon documentation on your web site. Because you will build your own sample applications in this book, we use the term “Cocoon” to describe the *software* and the *configuration files* you will edit to add your own application functions.

Due to how the software was written, Cocoon can be used in various ways. The most common way is by deploying it as a servlet inside a servlet engine, which in turn might be connected to a web server. This is shown in Figure 4.1. Another way of using Cocoon is through a command-line interface (CLI). This is described in detail in Chapter 6. Because of this flexibility, Cocoon could also be hosted in different environments, such as an Enterprise Java Bean (EJB) (although currently Cocoon has no code that lets you do that).

Figure 4.1. Cocoon: The big picture.



Cocoon offers a high degree of portability, because it can be run as a Java servlet. The specific code that allows this conforms to version 2.2 of the servlet API and therefore allows Cocoon to be run in environments such as a standalone servlet engine or a Java application server based on Sun's J2EE architecture.

Figure 4.1 shows how requests are routed through the web server and into a particular servlet. As shown in the figure, you can have several servlets running in a servlet engine at the same time. Configuration files that go with a servlet engine define which servlet should handle which request. These files are normally described in the servlet-engine documentation. When you installed Cocoon in the preceding chapter, this configuration took place, so the servlet engine we use as an example, Apache Tomcat, is already set up correctly.

Because Cocoon is most often used in this environment, this chapter first provides some background information on servlets and then takes a specific look at how Cocoon handles the requests it receives. We use the term "Cocoon servlet" to indicate that we are discussing Cocoon running in a servlet engine.

Requests and Responses

Every web page you see in your browser is sent over the Internet using Hypertext Transfer Protocol (HTTP). This application-level protocol is stateless and is based on the concept of requests and responses. Because the protocol does not contain a way to maintain a state, such as being logged in to a portal, additional mechanisms such as cookies are used to keep track of what you are doing.

In HTTP, clients—such as your web browser or your WAP phone—send requests for a particular piece of information to the server and in return receive that information. What the server returns could be an HTML document, a binary file such as an image, or data in an XML format. This request-response cycle is also the basic way Cocoon works. The incoming requests are routed via the web server through the servlet engine to the Cocoon servlet. The request is then processed by Cocoon, and a response is generated. This response takes the same way back: from Cocoon, to the servlet engine, to the web server, and then back to the client.

Every request the web server receives contains several pieces of information sent by the client. The most important part is the Uniform Resource Locator (URL), which specifies the document the client asks the server to send. URLs are absolute specifications, such as `http://yourserver:8080/cocoon/welcome`. This URL is used in the HTTP protocol to send the request to the specified server.

Today, URL is an informal term used with popular protocols such as HTTP and FTP. It has been superceded by the more general term Uniform Resource Identifier (URI), which describes any addressing scheme that uses strings to identify resources. We will use the more-correct term URI.

After the request reaches the server, the now-unimportant information, such as the server name, is stripped from the URI/URL. This results in a relative definition: `cocoon/welcome`. When using Cocoon, you have to deal with only this relative piece of the address.

In addition to the URI, the request contains further information, such as the user agent, which is text that the client program, such as a browser, sends. This text can be used by the server to detect the type of client used. The next couple of pages take a closer look at how Cocoon can act on this type of information when generating documents.

The server generates a response to every request. This response consists mainly of the document the client requested (such as a specific HTML page). In addition, the response contains a MIME type, which denotes the document's type, such as HTML or PDF. The browser uses this information to present the request in an appropriate manner.

When using Cocoon to build your own web applications, it is important to remember and understand the request-response cycle. Cocoon can act only if it receives a request. It cannot start doing something by itself. If no request comes in, nothing will happen on the server or in Cocoon.

Now that you have seen that Cocoon needs a request to get going, we will take a closer look at two terms that are often confused when talking about the Cocoon servlet—*servlet context* and *Cocoon context*.

Contexts Everywhere

One of the first things that can cause some confusion when you run Cocoon as a servlet is the term *context*. Therefore, we need to sort out the different ways this word is used to make the differences clear.

A servlet engine can run several different servlets at the same time. It therefore needs some way of differentiating between them. It uses *servlet contexts* to do this. You can think of each servlet context as being a separate web application. If several servlets are running in the servlet engine, these are independent applications that do not interfere with each other. Of course, these applications *can* communicate with each other and exchange information if they want to.

The concept of individual servlet contexts allows you to run several Cocoon applications at the same time on the same server. As an example of why you would want to do this, imagine a system hosting several portals for different firms. Each firm has its own look and feel, its own user base, and a pool of data sources that need to be integrated. Writing a separate Cocoon-based application for each portal and running them in separate servlet contexts makes them completely independent from one another. So, when one portal needs to be changed, such as if a new data source is added, the other portal is unaffected.

In order for this to work, each servlet has its own separate space on the hard disk. Just as a web server needs a physical starting point for serving HTML pages on the hard drive, a servlet has its own physical starting point, too. This is called the *Cocoon context*.

Usually all servlets are installed in the same directory. A servlet engine searches on startup automatically in this directory, which is often called webapps for web applications. If you install Cocoon as a web archive (WAR) file, this archive is unpacked to the webapps directory, and a directory named cocoon is created below this directory. The cocoon directory is the Cocoon context that is used as the root to generate documents. You can think of the Cocoon context as being its physical address on the server.

Cocoon is usually installed in a directory called webapps. After it is unpacked, you can delete the WAR file, cocoon.war. The following shows a typical directory structure in Tomcat 3.3 after Cocoon has been unpacked into its directory. This diagram does not show all the directories, so don’t worry if you have more.

```
\tomcat
    \work
    \bin
    \lib
    \logs
    \webapps
        admin.war
        \admin
        cocoon.war
        \cocoon
```

```
cocoon.xconf
sitemap.xmap
\WEB-INF
\stylesheets
\resources
\protected
```

The complete structure is made up of directories that belong to the sample application and directories that contain the software (remember that the WAR file contains both the Cocoon software *and* the examples).

The directories that belong to the examples contain the XML files and the stylesheets that are used to transform the data into the generated documents you receive in your browser.

Generating Documents

This book uses the term *document* to define the result of a request that is processed by Cocoon. This result can be an HTML page, a PDF document, WML formatted data, or some other format that Cocoon can generate. Using stylesheets to format the data in the output format you need lets you generate a variety of document types that can then be served to the appropriate applications for displaying.

When you build an application with Cocoon, instead of having a static file, you define a function that results in a particular document being generated and returned to the application that sent the request. Therefore, you could look at a web site built with Cocoon and not find a single HTML page on the filesystem.

Because of how Cocoon can generate the documents as and when they are requested, there is no need to add a descriptive extension to the request (such as ".html").

When the site administrator defines the function inside Cocoon that is responsible for handling the generation of a particular HTML document, he must configure that function so that it returns this format. How this is done is described in the following sections. As soon as the function has been configured to return HTML, accessing it results in the contents being delivered in the defined format. If the site administrator defines the same function to return a PDF document instead, that is what is returned. Therefore, the type of document (such as HTML or PDF) has nothing to do with its name.

One of the first things customers say when we tell them how Cocoon generates the documents dynamically is, "That must be slow."

It's true that if the documents *were* to be generated each time you requested them, Cocoon's performance wouldn't be as good as it is. However, apart from being able to serve some types of files, such as pictures, without generating them first, Cocoon also has a built-in configurable caching system that makes sure documents are served up as fast as possible. How the caching system works is explained when we talk about tun-ing your Cocoon installation in Chapter 6.

The dynamic generation of documents makes web site maintenance a lot easier for the administrator than it would be if all the documents were stored in their different formats. If the site administrator decides to change the result of `http://myserver/hello` from HTML to PDF, you don't need to call a different link. In fact, because of how a PDF viewer is integrated into today's browsers, everything is automatic. All you notice is that you receive a PDF document instead of an HTML page.

However, if you want to, you *can* add the descriptive extension to the document name. Nothing in Cocoon prevents you from calling a function that returns the HTML document hello.html. But this is not how you normally configure documents in Cocoon. We also do not recommend that you call your documents something like foo.xml. Your user will be worried about receiving an XML document, when in fact he will be receiving an HTML document generated from XML. A simple descriptive name is far easier for the user to remember.

Cocoon takes the concept of dynamic document generation one step further. Using components that are discussed later in this chapter, you can configure Cocoon to automatically generate the correct format on demand. This means that if you type the address into your HTML browser, you get the HTML version. If you access the same address using your WAP phone, you get the WML version. This is because the function can automatically determine at runtime what format to return.

By *automatically*, we do not mean that Cocoon will magically provide the correct HTML format. You still have to provide this yourself using a stylesheet. However, as soon as the function is set up, Cocoon can automatically select the correct format, depending on the used device or browser. Therefore, you don't have to remember separate URIs for each format. The fact that you can provide separate stylesheets for each format also means that there is no need to set up a single script to handle all the formats you might require. Remember how we told you about the difficulties of catering to the different browsers inside a scripting language such as ASP earlier in this book?

A document is not limited to just a visual representation of the data. Using Cocoon, you also can generate formats such as VoiceXML. This is a format that is used to describe data in a way so that it can be spoken to the user. In other words, if the correct software is installed on the client, you actually hear the document.

It is important to note that the document name can be more than just a logical content name; it can consist of any of the values that are allowed in URIs, including paths and parameters. For example,
`http://myserver/userdocuments/information?subject=howto` is a valid Cocoon document name.

The logic that decides how the content should be processed can be influenced by the parameters in the URI. Additional parameters such as the current browser you are using or even the time of day can also play a role in the generation.

This section has talked a lot about how the site administrator can set up a function to return a specific generated document. A complete application built with Cocoon can consist of many functions, each generating a particular document and returning it to the browser, for example. All the available functions are contained in the central Cocoon configuration file—the sitemap.

The Sitemap

The sitemap is the heart of Cocoon. It contains all the vital organs, such as the component declarations and the configured functions that a Cocoon-based application provides. The default location of the sitemap file, sitemap.xmap, is the root directory of the Cocoon context.

In Chapter 6, we will explain how to configure the number of sitemaps and their names and locations. However, this chapter uses only the default settings. Before we proceed, you might want to open your directory browser and find the file sitemap.xmap.

The sitemap file in itself is an XML document. You can edit this file with any tool suitable for editing XML documents or use your editor of choice. If you decide to edit the XML file with a non-XML editor, just make sure you do not forget to close XML tags. If you forget to do so, you will get an `Internal Server Error` when you try to access the document.

Why? Even though the sitemap is a file you can edit, Cocoon converts it into a Java class at runtime and then loads it. So, instead of constantly accessing the sitemap.xmap file from the hard disk, Cocoon imports all the configuration details. Obviously, any mistake in the XML file will hinder Cocoon in building the Java class.

The sitemap consists of two main areas: a library of available components and a document definition area. The document definition area uses the configured components to describe how a document should be generated. These two logical areas are split into several sections. As you can see in Listing 4.1, the sitemap consists of five sections that build the global structure.

Listing 4.1 The Global Sitemap Structure

```
<map:sitemap xmlns:map="http://xml.apache.org/cocoon/sitemap/1.0">
  <map:components/>
  <map:views/>
  <map:resources/>
  <map:action-sets/>
  <map:pipelines/>
</map:sitemap>
```

The sitemap uses its own namespace, which is introduced with the prefix `map` and is defined as http://xml.apache.org/cocoon/sitemap/1.0. This prefix is used throughout the sitemap. It is common to refer to the elements inside the sitemap using this prefix. This means that you would write `map:sitemap` instead of `sitemap`.

The rest of this chapter deals with the two most important sections of the sitemap: the components and the pipelines. Using these two sections, you will be able to build your first Cocoon applications. The other sections are described in more detail in Chapter 6. To show you how the sitemap works and how you can use it to build your own documents, we will use an example that you are probably familiar with.

A Closer Look at the Sitemap

The best way to introduce a new concept such as the sitemap is to start with an example you're familiar with and then explain how it is built using the Cocoon architecture and, in particular, how the sitemap is configured to achieve this.

We will introduce the different types of components that are available and how they can be combined to build pipelines. Each pipeline then results in a document being generated, as you saw earlier.

The Hello World Example

Nearly all books start with the typical Hello World example. Because we don't want to break this convention, we will do exactly the same. You will add your first Cocoon document—a simple function that produces the HTML output `Hello World`.

Open the sitemap with your XML tool or editor and locate the `map:pipelines` section. Remember that the sitemap is an XML file and that everything is contained inside XML tags, so you are looking for `<map:pipelines>`.

Because of the XML structure of the sitemap, each separate document entry is contained in the `map:pipelines` section.

As soon as you have found the correct position, add the code shown in Listing 4.2 so that they are underneath (or, in XML terminology, inside) the `map:pipelines` entry.

Listing 4.2 The Hello World Page

```
<map:pipeline>
  <map:match pattern="helloworld">
    <map:generate src="helloworld.xml"/>
    <map:transform src="helloworld2html.xsl"/>
    <map:serialize/>
  </map:match>
</map:pipeline>
```

Save the sitemap. For the moment, don't worry about the details of what you are doing. Everything will be explained after you see your first page.

Now, save the XML document shown in Listing 4.3 to the Cocoon context directory (probably webapps/cocoon).

Listing 4.3 The Hello World Document in XML

```
<?xml version="1.0"?>
<document>
  <text>Hello World</text>
</document>
```

Make sure you save the file using the name helloworld.xml. Do not forget to include the first line. Cocoon is very strict as far as how your XML document should look. Also make sure all your tags are closed (this is XML, remember?).

Finally, also add the stylesheet shown in Listing 4.4 to the Cocoon context directory.

Listing 4.4 The Hello World Document Stylesheet

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
              xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="document">
  <html>
    <body>
      <h1><xsl:value-of select="text"/></h1>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Again, make sure you enter the XML exactly as it appears here. If you make a mistake, you will not see the correct result. Save the file as helloworld2html.xsl.

Let’s take a quick break to double-check your work so far. Having followed the previous steps, your directory structure now should look something like this:

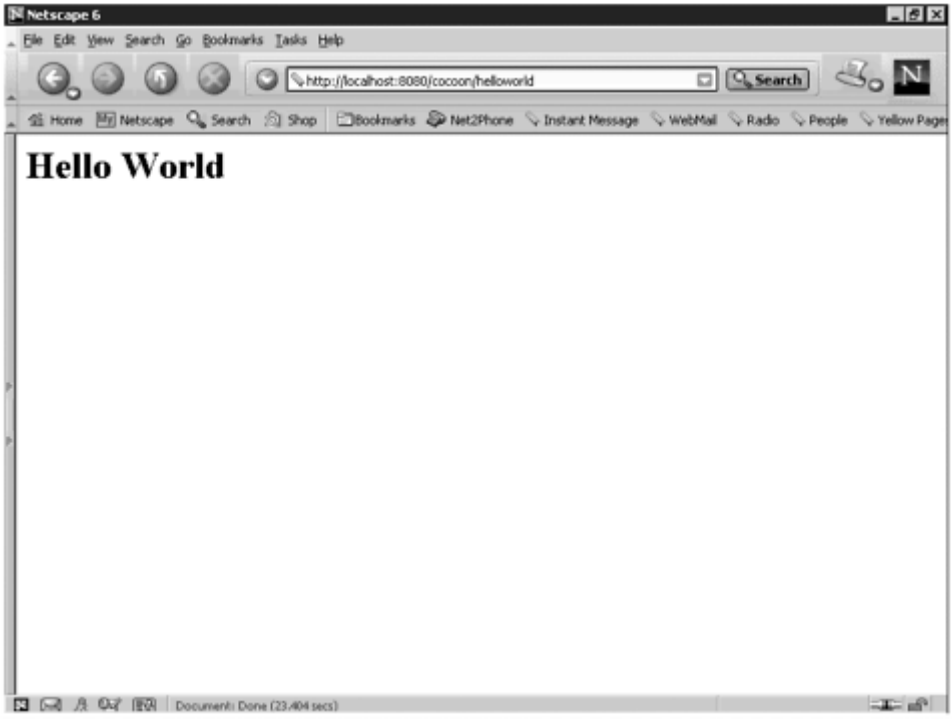
```
\tomcat
    \work
    \bin
    \lib
    \logs
    \webapps
        admin.war
        \admin
        cocoon.war
        \cocoon
            cocoon.xconf
            sitemap.xmap
            helloworld.xml
            helloworld2html.xsl
            \WEB-INF
            \stylesheets
            \resources
            \protected
```

Because you are running Cocoon as a servlet, you now need to start the servlet engine. This automatically instantiates the Cocoon servlet. Next, open your favorite browser and point it to `http://localhost:8080/cocoon/helloworld`. (See the preceding chapter for more details about starting and accessing Cocoon.)

Notice that there is a slight delay before Cocoon responds with the generated page. The reason for this is that the sitemap is transformed into a Java class and is then loaded by Cocoon. This transformation happens whenever Cocoon detects a change in the sitemap.

Your first self-created Cocoon document should look like Figure 4.2.

Figure 4.2. The Hello World example.



If you see an error page instead of a page that looks like Figure 4.2, there might be a mistake in one of the edited files. Also, the files need to reside in the directory structure explained earlier. If they were saved to a different directory, Cocoon will not be able to find them.

Cocoon can be configured to transform the sitemap in the background, so after adding a new pipeline to the sitemap, you might need to call the link a couple of times before you see the result.

You can also restart your servlet engine to make sure that Cocoon is loading the correct version of the sitemap. The CD that accompanies this book includes the Hello World example in a form that lets you integrate it into your Cocoon installation without having to type it in yourself.

Without yet going into further details, the Hello World example reads static information from an XML document (content) and applies a stylesheet (layout) to create an HTML representation of the content.

This simple example utilizes the most common sitemap components to generate the document. As you can see in this example, you have to define the steps that need to happen in order to have the document generated as requested. All this takes place in the sitemap, so you will spend a lot of time there.

But before you get carried away and think you already know all there is to know about generating documents with Cocoon, you must learn a bit more about sitemap basics. We will introduce how pipelines work and will show you the different types of components available.

Sitemap Components

The sitemap has two very important sections. One section contains the pipelines, and the other contains the available components that can be used to build them. We will look at the pipelines section later in this chapter. We will start with a general look at the component types.

Cocoon provides seven different types of components. Each component type has its own area in the sitemap, located inside the `<map:components>` tag:

```
<map:components>
  <map:generators/>
  <map:transformers/>
  <map:serializers/>
  <map:readers/>
  <map:selectors/>
  <map:matchers/>
  <map:actions/>
</map:components>
```

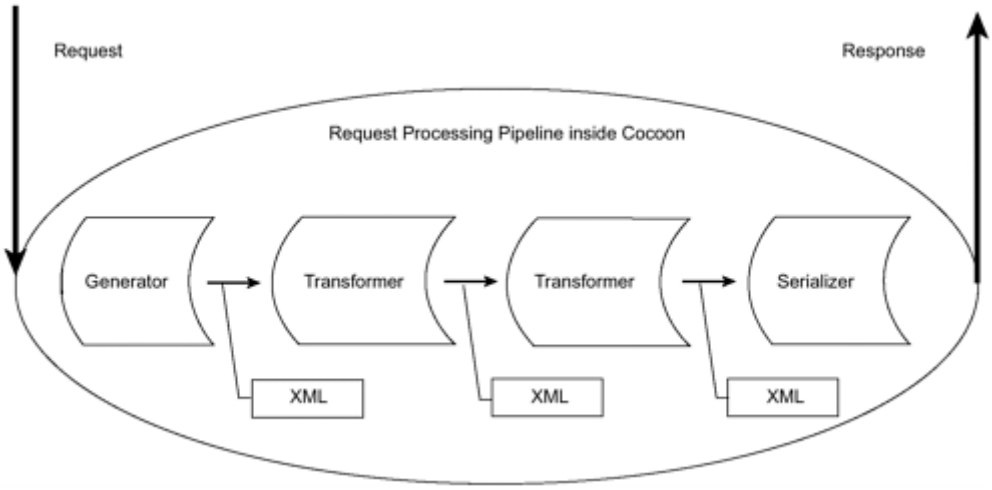
Because the sitemap is made up of nested XML tags, each subsection defines the available components for that type (all the different transformer components are defined inside the `<map:transformers>` tag). Each component has a unique name and is associated with a Java class that implements the component.

In this chapter, you will use only the available components, as opposed to writing your own, so you do not need to change the components section. Therefore, you can use it as a reference to see which sitemap components are available and what their names are.

Cocoon provides a variety of components that can be used to build pipelines. A *pipeline* is a processing chain that tells Cocoon what do to when a document is requested. A pipeline can range from the very simple to the very complex. You saw a simple pipeline in the Hello World example where an XML file was read and transformed into HTML

using available components. A more-complicated pipeline would include additional components that perhaps connect to a database to retrieve data or send an email. Figure 4.3 shows how a Cocoon pipeline is built.

Figure 4.3. A Cocoon pipeline.



The most common way of building pipelines is to define an XML processing chain that consists of three component types: a *generator*, one or several *transformers*, and one *serializer*. Because each document is generated from an XML format, you need a component that starts the process by sending the initial XML data into the pipeline. The generator does this.

Generator

Each pipeline in Cocoon must start with a single generator. Most often, the generator fetches data from a data source, converts it (if necessary) into an XML format, and makes the XML available for further processing in the chain. A simple example is a generator that reads an XML document from the hard drive, as is the case in the Hello World example.

Data that is accessed by a generator does not need to be in an XML format. If it's not, the generator simply performs all the necessary steps to create XML from whatever data format it receives. Examples include transforming HTML to XHTML and getting emails from a mailbox and then generating XML from the text of the emails. It is important to remember that when the generator has finished its job, the format that is exchanged between components in the pipeline is XML.

As soon as the generator has created or read the XML format, it passes the tags to the components next in line in the pipeline for further processing. The next type of component to come into play is the transformer.

Transformer

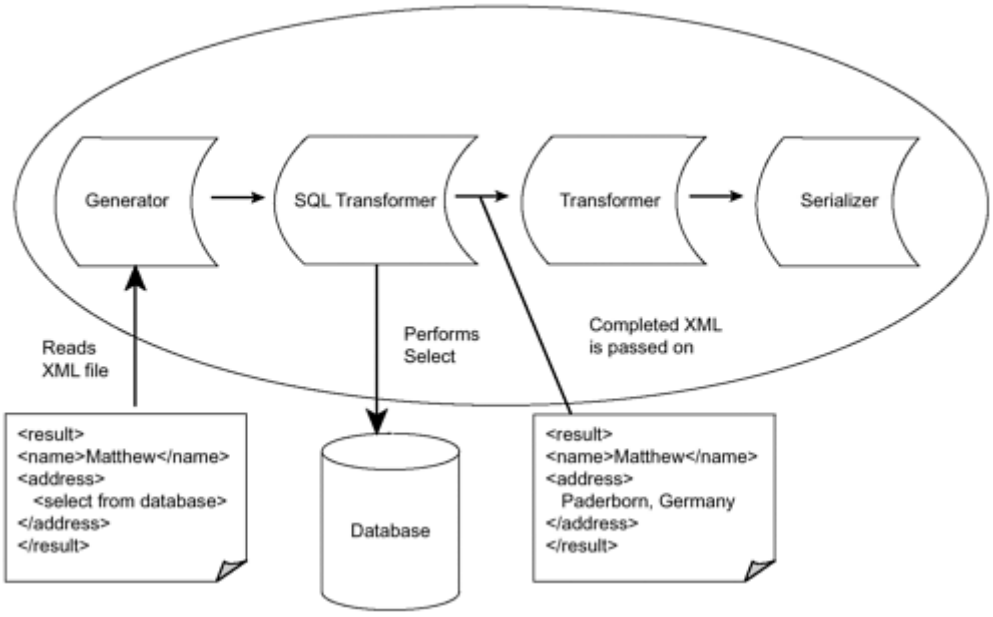
The transformer is an optional component that can be used in the pipeline after the generator. As the name states, the transformer manipulates the XML data that the generator sends.

A transformer receives XML data from the generator or another transformer (there can be several transformers in a pipeline). One of the most common transformers used in a Cocoon pipeline is the xslt transformer. The xslt transformer uses XSL stylesheets to transform XML data into an output format such as XHTML. In the Hello World example, the generator reads a simple XML file containing just the text you want to display. The transformer then transforms the text into XHTML using the stylesheet you wrote.

The XML data that the generator reads or creates is not limited to plain data you would see in the end document. It can also contain tags that a transformer can act on. Therefore, a transformer does not always have to transform the whole document. It can also focus on just the information (or XML elements) it needs to perform its task. If the XML data contains tags that are specific to the current transformer, it evaluates that information and acts on it to get additional content or to manipulate the XML data in some other way.

Figure 4.4 shows a pipeline in which a sql transformer acts on specific tags. The generator reads an XML document specifying a SQL statement for a particular database. This statement can consist of the database's name and the table to query. The transformer then processes the XML, extracts the SQL query, and performs the fetch against the database. The data received from the database is then transformed into XML and is inserted into the XML document for further processing. The data replaces the original command in the XML tree.

Figure 4.4. A Cocoon pipeline containing commands and data.



As soon as the transformer has finished its work, it can pass the XML data to the next component in line, which may be another transformer. This means that the file originally read by the generator can contain specific tags for several transformers. So, apart from having commands for the sql transformer, the XML file could also contain tags for a mail transformer. As each transformer in line is called and processes its specific commands, the end result is built as the command tags are replaced.

It is also possible for a component to dynamically generate commands for a component later in the chain. This means that it is not necessary for all the commands to be in the file the generator reads. For example, the sql transformer performs a query against the database and, as a result, receives data it can use to form a command for the next transformer in line, such as a mail transformer.

Let's look at this process in more detail. The mail transformer understands commands that tell it who to send an email to (the address) and what to put in the email (the subject). Using the file generator, the sql transformer, and the mail transformer, you can build a pipeline that sends an email to the administrator if the database is inaccessi-ble. In order for this to happen, the sql transformer could automatically generate the commands for the mail transformer into the XML tree if a connection to the database is not possible. If all goes well and the sql transformer can get the information from the database, no command is generated, and the mail transformer has nothing to do.

Instead of the sql transformer generating the commands, it is also possible to use a stylesheet and the xslt transformer, located between the sql transformer and the mail transformer, to do this. Just as a transformer can generate tags that are then sent to the next component, so can a stylesheet.

Of course, if you are more familiar with Cocoon, you are probably mumbling something about there being no mail transformer in Cocoon. You're right, but you can develop one, as you'll see in Chapter 9.

Commonly, the pipeline builds the content starting at the generator and then by using transformers. The last step in the pipeline is creating the layout. This is typically done using the xslt transformer. At this point, you are still inside the pipeline processing, so you are still dealing with XML. The consequence is that the laid-out content is also XML. For example, in the case of HTML, it would be XHTML.

If the format you want to return to the client application is any non-XML format, such as PDF, the result of the last transformer is an intermediate format, which is then processed by the serializer. For PDF, this intermediate format is called XSL:FO.

Serializer

Every pipeline must end with a serializer component. To be more precise, every *path* through a pipeline must end with a serializer. As you will learn later, the path through the pipeline can be influenced. Therefore, each pipeline can have one or more exits—each

with its own serializer. For now, and for the Hello World example, it is enough to remember that each pipeline ends with a serializer.

A serializer receives the generated and transformed XML and serializes it into the format required by the end device or program. How the serialization takes place is determined by the type of serializer used. A serializer that is responsible for serializing HTML to the browser may manipulate the received XHTML format so that it conforms to the HTML standard (such as by removing a closing `<br/>` tag).

In addition to doing this, the serializer can add specific information to the document so that the application can display the document correctly. For example, a browser needs information such as the MIME type in order to determine whether it can display the data itself or if it needs to start an external viewer.

If the XML format is related to the end format (such as XHTML to HTML), the serializer does not have that much work to do. However, the process is different if the end format differs completely from the XML-based format. Two examples of this are generating PDF documents and JPEG images. In both cases, Cocoon can produce these formats from an XML-based format (XSL:FO and SVG, respectively) using the appropriate serializers.

Now that you know how to build a pipeline using the different component types, you need to be able to configure Cocoon so that a request you send will be forwarded to your pipeline for processing. This is done using a matcher.

Matcher

The main function of the matcher is to allow document requests to be matched against a pipeline. Imagine matchers as being the locked door to a particular pipeline, allowing entry only if the key fits. In this case, the key is the request. Armed with the received request, Cocoon tries each door in the sitemap to see which one can be opened.

To put it another way, the matcher acts like a giant `switch/case` clause or like lots of `if` clauses in a programming language. If a configured matcher can match the request (or part of the request, as you will see later), Cocoon passes the request to the enclosed pipeline for processing. If the matcher does not fit, Cocoon tries the next one in the pipeline until the end is reached. If no matcher fits the request, Cocoon generates an error.

The matcher normally used is the *wildcard* matcher, which tests the incoming URI, or the document name, against a particular pattern. In the Hello World example, this matcher is used to test the document name against the static pattern `helloworld`.

The value that is tested by a matcher is not limited, which means that it can be any value sent with the request but also the current time or any other piece of system information. However, it is most common for the matcher to test against the requested document name. As you dig deeper into Cocoon, you will learn more about matchers and how they work.

Now that you've learned about the different types of components available, the next step is to learn how these different components are used inside the sitemap. So it is time to take another look at the sitemap. You might want to open the sitemap with your editor and follow along as you read.

The Sitemap Pipelines

After the components section (`map:components`) in the sitemap, the most important section is the `map:pipelines` section. This section defines all the processing chains for your application. If you look at this part of the sitemap, you will see that inside the `map:pipelines` tag are several `map:pipeline` (without the `s`) tags. And inside a `map:pipeline` are the `map:match` tags. This is slightly complicated, so we need to explain this in more detail.

The first distinction we need to make is to define the term *pipeline*. Each document can actually be thought of as a pipeline through which XML flows from one component to another. Indeed, this is the term used in Cocoon to describe figuratively the chain of components. However, this term is also used as the name for the XML tags. These two definitions are not quite the same.

Each pipeline (each chain of components that produces the document) is enclosed by a `map:match` tag. This combination can be thought of as an individual function or virtual URI that exists inside the Cocoon application. Because Cocoon provides the mechanisms for creating virtual URIs, this means that the URI space does not necessarily have to match the filesystem space on the server. (In other words, an address `http://localhost:8080/cocoon/hello` does not mean that there is a file called hello on the server.)

A collection of documents (one or more) is then enclosed by a `map:pipeline` tag. This allows each collection to have its own error-handling routines. We will explain this shortly. You can also use one or more `map:pipeline` sections to separate your document pipelines into distinct groups. This can make maintaining large applications easier.

All the `map:pipeline` sections are then enclosed by a `map:pipelines` tag. To distinguish between the different uses of the word *pipeline*, we will use the term *pipeline* for the processing chain and `map:pipeline` for a section in the sitemap containing one or more pipelines.

The `map:pipelines` section describes the processing flow. Each time a request comes into Cocoon, this section is processed from top to bottom until the request has been completely processed. The component used most often to control this flow is the matcher, which acts as the gateway to each pipeline. A very simple flow can be created by specifying a match directive for each possible document, as shown in Listing 4.5.

Listing 4.5 A Separate Match for Each Document

```
<map:pipeline>
    <map:match pattern="news" type="wildcard">
        <map:generate src="news.xml" type="file"/>
        <map:transform src="news2html.xsl" type="xslt"/>
        <map:serialize type="html"/>
    </map:match>
    <map:match pattern="products" type="wildcard">
        <map:generate src="products.xml" type="file"/>
        <map:transform src="products2html.xsl" type="xslt"/>
        <map:serialize type="html"/>
    </map:match>
</map:pipeline>
```

The flow through such a sitemap is straightforward. Cocoon processes one match directive after the other until a match is successful. If this match is successful, the XML processing pipeline is then created according to the instructions inside the `map:match` element. This pipeline is then executed, and the flow through the sitemap finishes.

In general, the flow through the sitemap is stopped whenever a serialize directive (or, more technically, a `map:serialize`) is reached. The flow is not finished just because a match is successful! To show this, Listing 4.6 is the same as Listing 4.5, but it uses separate `map:match` elements around each component.

Listing 4.6 A Separate Match for Each Component

```
<map:pipeline>
    <map:match pattern="news" type="wildcard">
        <map:generate src="news.xml" type="file"/>
    </map:match>
    <map:match pattern="products" type="wildcard">
        <map:generate src="products.xml" type="file"/>
    </map:match>
    <map:match pattern="news" type="wildcard">
        <map:transform src="news2html.xsl" type="xslt"/>
    </map:match>
    <map:match pattern="products" type="wildcard">
        <map:transform src="products2html.xsl" type="xslt"/>
    </map:match>
    <map:serialize type="html"/>
</map:pipeline>
```

As this listing shows, you can use the match to surround a complete pipeline or an individual entry in the pipeline. Although it is possible to surround an individual entry, we do not recommend that you write your pipelines like this. You should use the match to enclose the *complete* pipeline.

This completes our first theoretical overview of the sitemap, the various component types available, and how a request is processed using the information inside the sitemap. Next we will expand on this and fill in more practical details so that you can then build your first real examples using Cocoon.

Getting Practical

Now you know the theoretical basics of using Cocoon: the components and the pipelines of the sitemap. To get more practical, we will start with a very small sitemap that contains only what you have learned so far, and that includes the Hello World example.

But before we start, here's a piece of advice that we mentioned when we talked about the naming and location of the sitemap: Cocoon provides defaults for most configuration parameters. We advise you not to change these defaults at the moment and, indeed, not to do so before you are more comfortable with the sitemap. The default settings were chosen to make life easier for someone starting with Cocoon, so we will stick with them for the moment. In Listing 4.7, it is easy to find the default values, because they are labeled as such.

Listing 4.7 The First Sitemap

```
<map:sitemap xmlns="http://xml.apache.org/cocoon/sitemap/1.0">
    <map:components>
       <map:generators default="file">
          <map:generate name="file"
     src="org.apache.cocoon.generation.FileGenerator"/>
       </map:generators>
       <map:transformers default="xslt">
          <map:transformer name="xslt"
src="org.apache.cocoon.transformation.TraxTransformer"/>
       </map:transformers>
       <map:serializers default="html">
          <map:serializer name="html"
src="org.apache.cocoon.serialization.HTMLSerializer"
                          mime-type="text/html"/>
       </map:serializers>
       <map:matchers default="wildcard">
          <map:matcher name="wildcard"
src="org.apache.cocoon.matching.WildcardURIMatcherFactory"/>
       </map:matchers>
    </map:components>
    <map:pipelines>
       <map:pipeline>
          <map:match pattern="helloworld">
             <map:generate src="helloworld.xml"/>
             <map:transform src="helloworld2html.xsl"/>
             <map:serialize/>
          </map:match>
       </map:pipeline>
    </map:pipelines>
</map:sitemap>
```

The first section in this listing is `map:components`. You then see four of the available component types: generator, transformer, serializer, and matcher. These are the component types you need to build the pipeline to generate a document with the name helloworld. Each component type definition contains one component. Looking at the `map:transformers` section as an example, you can see that a transformer is defined with

the name "xslt". The actual pipeline, built from these components, is defined in the map:pipeline that is inside the map:pipelines section.

Each sitemap component type is defined by an element that has the same name as the type of the component (for example, a generator is defined inside a map:generator tag). This tag is then nested inside an element that uses the plural of the type as its name. For example, all available generators are defined using an individual map:generator element, which is then inside the map:generators element. Cocoon knows only about generators listed in this section, so these are the only ones you can use in your pipelines. This is true for all other component types as well.

The component definitions also have more in common. Each component has two attributes, src and name. The name is a unique name for this component type. It is used later in the pipelines to identify exactly which component should be used. The source is the Java class that implements the component. In addition to these two attributes, all serializers have a third attribute in common: the mime-type. This information is used to define the output format for the client to display the generated document correctly.

For every defined component type there is always one component that is the default component. This means that you can leave out the details of this component in your pipeline if you want Cocoon to use the default component. This default component is defined by the attribute default in the section for the component type. The value of this attribute must be the name of a defined component. For example, as you can see in Listing 4.7, the default generator is the file generator. The standard distribution of Cocoon defines a set of default components. For the four components we are currently looking at, you can see the defaults in the listing. This listing uses the default components for each component type. This set is very useful and common. So if you build only simple pipelines, you will find that you do not have to specify the component name because the default component is exactly the component you need.

Although at first glance this seems like a very helpful feature, it has potential dangers, too. For example, should you need to change the default component of a particular component type, you have to be sure to change every usage of the old default component beforehand! Otherwise, none of your pipelines will work, because they expect a different component to be the default. Another problem with default components is exchanging sitemap entries. When exchanging with someone else pipeline definitions that use default components, you need to make sure that the defaults are the same in both cases.

So, be especially careful when changing the default component settings. We suggest that you avoid this whenever possible. This book uses the default components defined by the Cocoon distribution.

In order to use a sitemap component inside a map:pipelines section, simply add an element where the name is a verb derived from the component type. For example, for a generator, use the element name generate, for a transformer, use the name transform,

and so on. This makes it easier to remember the name of the XML tags you need to edit into your pipelines when you want to use the particular components.

To specify which sitemap component of that type is to be used, you can specify the attribute `type` for each component, as shown in the Listing 4.8 pipeline example. This listing is equivalent to the `map:pipeline` in the preceding sitemap. However, we have now added the `type` attribute to each component, although you are in fact using only the default components.

Listing 4.8 Using Explicit Type Definitions

```
<map:pipeline>
    <map:match pattern="helloworld" type="wildcard">
        <map:generate src="helloworld.xml" type="file"/>
        <map:transform src="helloworld2html.xsl" type="xslt"/>
        <map:serialize type="html"/>
    </map:match>
</map:pipeline>
```

It is a matter of style whether you use the default types or whether you always define the type explicitly. We suggest that when you write your first pipelines, you explicitly add the type, even though you will probably be using the default types. As soon as you feel more at home in the sitemap, you can omit the `type` attribute for the default components. This chapter uses the standard notation, which is to omit the `type` attribute for default components. If you are unsure what the default component is, just refer to the beginning of the sitemap, where all the components are listed.

After you have defined your sitemap and listed the components you want to use, the next step is to give them something to do. For example, in the case of the generator, you need to tell it which XML file to read in.

Resolving Resources

Most sitemap components have something in common: They all read a resource. By resources, we mean such things as XML documents, XSL stylesheets, script files, and images. For all components, such a resource is usually defined by the source attribute inside a pipeline; it is abbreviated as `src`. Be careful not to confuse this with the attribute of the same name used when the component is *defined* in the components section.

In the Hello World example, an XML document named helloworld.xml is read by the file generator and an XSL stylesheet named helloworld2html.xsl is read by the xslt transformer.

All resources are defined through URIs, which are resolved by Cocoon. This includes making relative URIs absolute for the purpose of reading resources. If the URI is relative, Cocoon assumes that this URI points to a file on the local hard drive and resolves it to the current sitemap context. This is usually the same as the Cocoon context (the directory

where Cocoon is installed). For example, the URI `helloworld.xml` is resolved to a file called helloworld.xml, which is located in the Cocoon context directory. However, if there are subsitemaps beneath your main sitemap, this location might differ. Because you will learn more about subsitemaps in Chapter 6, don't worry about the sitemap context for now. Just remember that relative URIs are resolved with respect to the Cocoon context for the (main) sitemap.

If the URI is absolute, it is resolved using the standard mechanisms. For example, `http://databaseserver/datas/index.html` is fetched using HTTP from the given address.

In addition to the standard protocols, such as reading from a file or reading from another web server via HTTP, Cocoon offers some additional protocols that, for example, allow you to use the result of one pipeline as the input for a generator. These protocols and their uses are explained in detail in Chapter 6.

Now that we have looked at a small sample sitemap, explained how components are defined, and looked at how they can access files as input, we will now show you the main components you will use to build your pipelines.

Common Components

We have used the Hello World example to explain the most important concepts without actually explaining what the components do exactly. Now is the time to change that and tell you what the components actually can do. You will see how you can read in an XML file as the starting point for a pipeline, how you can transform that XML into a format fit for viewing, and which serializer you can use to send HTML back to the browser. We will also list additional components you can integrate into your own pipeline. This chapter does not present all the components. As with the rest of our tour, we will take this step by step. Additional components are explained in Chapter 6. You will also find a complete list of the available components in Appendix A, "Cocoon Components."

The generator is the starting point for each pipeline. Because the file generator can read XML from a variety of sources, we will look at it first.

The File Generator

The most common generator is the file generator. It reads an XML document and inserts the content of this document into the pipeline. The document can either be stored on the server's local hard drive or fetched from any URI.

It would really have been better to name this generator `URIGenerator`, but the name *file generator* has a historical explanation. It was the first generator developed for Cocoon, and it could read only files from the local hard drive. Later versions of the generator were extended to support any form of URI, but the name never changed.

In fact, all protocols supported by the Java Developer Kit (JDK) can be used to fetch XML documents, such as HTTP and FTP. In addition, Cocoon offers several ways of adding your own protocols to the system. The file generator can use all these protocols. For more information on adding new protocols, see Chapter 9.

Listing 4.9 shows how the file generator is defined in the `map:components` section of the sitemap and gives two examples of using it in your own pipeline. Note that this listing and the following ones do not show complete sitemaps. You can refer to your own sitemap or to the earlier Hello World example to see the exact syntax for each part.

Listing 4.9 The File Generator

```
<map:generators default="file">
    <map:generate name="file"
                  src="org.apache.cocoon.generation.FileGenerator"/>
<map:generators/>
...
<map:generate src="localfile.xml"/>
<map:generate src="http://newserver/latestnews.xml"/>
```

The file generator is the default generator, so you do not need to specify the `type` attribute for the `map:generate` element. The `src` attribute defines the location of the XML document. As explained in the previous paragraphs, just the default generator setting should be changed only with great care.

The html Generator

The html generator reads an HTML document from the local filesystem or from any URI. It acts much like the file generator, except that it reads HTML documents and converts them to XHTML using the open-source solution JTidy. JTidy is used because when the generator passes control to the transformers, the data must be in an XML format. JTidy checks the HTML tags and, for example, makes sure that each opening tag (such as `<br>`) also has a closing tag.

The html generator can be used for legacy documents that already exist in HTML or to integrate existing web applications into your Cocoon application. Listing 4.10 shows how the generator is defined in the sitemap and presents two examples. Note that the `type` attribute is used because the html generator is not the default transformer.

Listing 4.10 The html Generator

```
<map:generators>
    <map:generate name="html"
                  src="org.apache.cocoon.generation.HTMLGenerator"/>
</map:generators>
...
<map:generate src="localfile.html" type="html"/>
<map:generate src="http://oldserver/legacy.html" type="html"/>
```

Therefore, when using the html generator, you *have* to specify the `type` attribute with a value of `html` and also remember the `src` attribute, which defines the HTML document's location. This document can either reside on the filesystem or be accessible via HTTP or some other protocol.

Apart from loading an XML or HTML file as the input for the pipeline, you also might want to access other available data and present that. The following generators provide an easy way to do that.

The Directory Generator and the Image Directory Generator

The directory generator and the image directory generator read the content of a directory on the local hard drive and generate an XML representation of this content. Listing 4.11 shows how the two components are configured in the sitemap and how they can be used in a pipeline.

Listing 4.11 The Directory Generator and the Image Directory Generator

```
<map:generators>
    <map:generate name="directory"
                src="org.apache.cocoon.generation.DirectoryGenerator"/>
    <map:generate name="imagedirectory"
        src="org.apache.cocoon.generation.ImageDirectoryGenerator"/>
</map:generators>
...
<map:generate src="stylesheets" type="directory"/>
<map:generate src="stylesheets" type="imagedirectory">
    <map:parameter name="depth" value="2"/>
</map:generate>
```

The node of the generated document is normally the `directory` element. A directory node can contain zero or more file or directory nodes. A file node has no children, whereas a directory node can have the same sorts of children. Each node contains the following attributes:

- `name`: The name of the file or directory.
- `lastModified`: The time the file was last modified, measured as the number of milliseconds since 00:00:00 GMT, January 1, 1970. This measurement is based on the Java implementation for the calculation of time.
- `date` (optional): The time the file was last modified, in a more human-readable form.

All generated elements have the namespace http://apache.org/cocoon/directory/2.0. The root directory node has the attribute `requested` with the value `true`.

The following parameters can be specified in the pipeline for the directory from which the information is to be generated:

- depth (optional): Sets how deep the directory generator should delve into the directory structure. The default value is 1, which means that only the contents of the starting directory are returned.
- dateFormat (optional): Sets the format for the date attribute of each node. This format is taken directly from standard Java. For more information, take a look at the java.text.SimpleDateFormat class. If no format is specified, the default format is used.
- root (optional): The root pattern.
- include (optional): A pattern describing the files to be included.
- exclude (optional): A pattern specifying the files to be excluded.

For example, the directory generator has produced the XML shown in Listing 4.12 from a directory called stylesheets.

Listing 4.12 Sample Output for the Directory Generator

```
<dir:directory xmlns:dir="http://apache.org/cocoon/directory/2.0"
             name="stylesheets"
             lastModified="999425490000"
             date="02.09.01 12:11"
             requested="true">
   <dir:directory name="sites"
                lastModified="999425490000"
                date="02.09.01 12:11"/>
   <dir:file name="dynamic-page2html.xsl"
           lastModified="999425490000"
           date="02.09.01 12:11"/>
   <dir:file name="simple-xml2html.xsl"
           lastModified="999425490000"
           date="02.09.01 12:11"/>
</dir:directory>
```

The image directory generator extends the directory generator. The generated XML contains more information if the directory contains images. The following attributes are added for images:

- width (optional): The width of the image if it is an image file.
- height (optional): The height of the image if it is an image file.

If you want to use the directory generator, the type of the generate directive is simply directory. For the image directory generator, this is imagedirectory.

After you have generated some information about the directories on your system, another helpful pipeline to write is one that provides some details about the request you sent to Cocoon.

The Request Generator

The request generator uses the current request to produce XML data. When a document is requested from Cocoon, the device sends a request to Cocoon. Besides containing the name of the requested document, the request also contains additional information, such as various connection parameters and the type of browser used.

This generator converts some of the information contained in the request into structured XML. In contrast to the other generators described so far, the output of this generator is not static. If the same document is processed, the file generator reads the same XML document on each call and produces the same output. The output of the request generator might be different on each call, because the data in the request will change.

If you want to use the request generator, the `type` attribute of the generate directory must be set to `request`, as shown in Listing 4.13. It shows the configuration in the sitemap and its use in a pipeline.

Listing 4.13 The Request Generator

```
<map:generators>
   <map:generate name="request"
                 src="org.apache.cocoon.generation.RequestGenerator"/>
</map:generators>
...
<map:generate src="optional_source" type="request"/>
<map:generate type="request">
   <map:parameter name="test" value="yes"/>
</map:generate>
```

One sample usage shows the specification of an `src` attribute for the generator. The value of this parameter is included in the output as an attribute of the root element. As shown in Listing 4.14, which is the result of the second sample usage, the request generator uses the namespace http://xml.apache.org/cocoon/requestgenerator/2.0 for the tags it generates.

Listing 4.14 Sample Output for the Request Generator

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- The root element is request. The target attribute is the requested URI
     and the source attribute is the optional source attribute of the sitemap
     entry for this pipeline. -->
<request target="/cocoon/request" source=""
         xmlns="http://xml.apache.org/cocoon/requestgenerator/2.0">
 <!-- First the headers: -->
 <requestHeaders>
   <header name="accept-language">de</header>
   <header name="connection">Keep-Alive</header>
   <header name="accept">image/gif, image/x-xbitmap, image/jpeg,
    */*</header>
   <header name="host">thehost.serving.cocoon2</header>
   <header name="accept-encoding">gzip, deflate</header>
   <header name="user-agent">Browser User Agent</header>
```

```
    <header
     name="referer">http://thehost.serving.cocoon2/cocoon/welcome</heade
     r>
  </requestHeaders>

  <!-- All request parameters: -->
  <requestParameters>
    <!-- Create a parameter element for each parameter -->
    <parameter name="login">
      <!-- Create a value element for each value -->
      <value>test</value>
    </parameter>
  </requestParameters>

  <!-- All configuration parameters: -->
  <configurationParameters>
    <!-- Create a parameter element for each parameter specified in the
     pipeline
        for this generator-->
    <parameter name="test">yes</parameter>
  </configurationParameters>
</request>
```

As you can see in this output, the XML document has three different sections: `requestHeaders`, `requestParameters`, and `configurationParameters`.

The first section contains all headers sent by the browser or device. This includes the user agent, which can be used to determine the type of device.

The request parameters are all the parameters sent with the request. For each parameter, a new subtree starting with the element `parameter` is created. The attribute `name` contains the name of the parameter. For each of this parameter's values, a separate subtree with the element `value` is created. This `value` element contains the value as a text node.

The last section contains all parameters defined for the request generator in the pipeline. Each parameter has its own `parameter` element. The attribute `name` holds the parameter's name, and the value is enclosed by the parameter's tag. The second pipeline in the example uses such a parameter, called `test`. The generated XML document contains one entry for this parameter with the value `yes`.

To obtain more information on your system, Cocoon provides a generator that you can use to generate the most important facts and figures.

The Status Generator

The status generator uses Cocoon's current status or configuration to produce XML data. Like the request generator but in contrast to the other generators discussed so far, the output of this generator is not static, because the output contains volatile information such as memory usage.

Listing 4.15 shows the configuration in the sitemap and the use of this generator in a pipeline. Remember that the status generator is not the default generator, so you need to add the `type` attribute.

Listing 4.15 The Status Generator

```
<map:generators>
   <map:generate name="status"
                 src="org.apache.cocoon.generation.StatusGenerator"/>
</map:generators>
...
<map:generate type="status"/>
```

When specifying the format of the XML it provides, the status generator uses the namespace http://xml.apache.org/cocoon/status/2.0. As you can see from Listing 4.16, the root element has the name `statusinfo`. It contains two attributes, `host` and `date`, which contain the server's host name and the current date, respectively.

Listing 4.16 An Example of the Status Generator

```
<?xml version="1.0" encoding="UTF-8"?>
<statusinfo date="16.07.2001 16:46:20" host="myhost"
          xmlns="http://apache.org/cocoon/status/2.0"
          xmlns:xlink="http://www.w3.org/1999/xlink">
  <group name="vm">
    <group name="memory">
      <value name="total"><line>11788288</line></value>
      <value name="free"><line>2778208</line></value>
    </group>
    <group name="jre">
      <value name="version"><line>1.3.0</line></value>
      <value type="simple" href="http://java.sun.com/" name="java-vendor">
        <line>Sun Microsystems Inc.</line>
      </value>
    </group>
    <group name="operating-system">
      <value name="name"><line>Windows 2000</line></value>
      <value name="architecture"><line>x86</line></value>
      <value name="version"><line>5.0</line></value>
    </group>
    <value name="classpath">
      <line>classes</line>
      <line>lib\ant.jar</line>
      <line>lib\jasper.jar</line>
    </value>
  </group>
</statusinfo>
```

All other information is grouped by two elements: `group` and `value`. A group collects several pieces of information about one specific topic. The topic is defined by the attribute `name` on the group. In Listing 4.16, the group of information belonging to the

Java virtual machine (vm) contains information on the memory situation, contained in the group memory.

Each individual piece of information is contained in a value tag, with the attribute name describing the information. Looking again at the memory group, you can see two pieces of information—the total memory and the currently free memory.

This component is useful for viewing your system's current status and for debugging if something goes wrong. So, as an exercise, you might like to write a small pipeline that includes the status generator and then view the information about your system.

This completes your first look at generators. Now we will move on to transformers and to the transformer you will use the most in your pipelines.

The xslt Transformer

When writing your pipelines, the transformer you will use most often is probably the xslt transformer. This is the transformer that can use a stylesheet to transform, for example, an XML data format into an output format such as XHTML. You can also use a stylesheet to manipulate the XML data in some other way, such as adding or removing tags. Remember, a stylesheet transformation does not need to result in a format suitable for viewing! The result can just as well be some other format that is more suitable for further processing. Think back to our earlier description of the pipeline containing the sql transformer and the mail transformer. There you saw how to use a stylesheet in the pipeline between the two transformers to generate the necessary commands.

The xslt transformer is defined in the sitemap just as the generators you saw in the previous sections. However, the transformers reside in their own section, map:transformers, as shown in Listing 4.17.

Listing 4.17 The xslt Transformer

```
<map:transformers default="xslt">
    <map:transformer name="xslt"

     src="org.apache.cocoon.transformation.TraxTransformer"/>
</map:transformers>
...
<map:transform src="localstylesheet.xsl"/>
<map:transform src="http://stylesheetserver/styles/style.xsl"/>
```

This sitemap excerpt shows that the xslt transformer is the default transformer, so you do not need to specify the type attribute for the map:transform element when you use this transformer in a pipeline. The src attribute defines the location of the XSL stylesheet. As explained in the previous paragraphs, the default transformer setting in the sitemap should be changed only with great care.

So far, we have looked at a few different generators that provide the data that then flows through the pipeline. Using the xslt transformer, you can then transform the data into a format such as one suitable for viewing.

We will extend our look at the xslt transformer and the other available transformers in Chapter 6. Now it is time to find out how you can convert a format such as XHTML into something that the browser can display. For this you need a serializer.

The html Serializer

When you start building your first examples with Cocoon, you will use an Internet browser to view them. The Internet browser requires the data to be in HTML so that it can be displayed. Therefore, you need to add the html serializer as the end point of your pipeline. The term *html serializer* might lead you to believe that the serializer can generate HTML from some other format, but this is not the case.

The document itself must already contain valid XHTML. The html serializer forms HTML from this XHTML. Because XHTML is an XML language, it is well-formed, so all elements are closed correctly, and so on. In contrast, HTML is derived from SGML and is not as restrictive. For example, some elements, such as the `br` element, are not closed. That's exactly the job of the html serializer. It converts the "perfect" XHTML to the weak HTML by removing the closing `br` element.

Before you can use the html serializer, you must apply a stylesheet, using the xslt transformer, to the XML document to lay out the data in XHTML. Then the serializer can serialize that into HTML.

The output of the html serializer is text containing the HTML. In addition, the serializer sets the document's MIME type to `text/html`. Listing 4.18 shows the configuration of the html serializer and how to use it in a pipeline. Because it is the default serializer, you do not need to add the `type` attribute in the pipeline.

Listing 4.18 The html Serializer

```
<map:serializers default="html">
   <map:serializer name="html"
                mime-type="text/html"
              src="org.apache.cocoon.serialization.HTMLSerializer"/>
</map: serializers>
...
<map:serialize/>
```

Like most other serializers, the html serializer can be configured in various ways. You can find a full description in Appendix A. But one configurable parameter is worth mentioning here: the encoding (see Listing 4.19).

Listing 4.19 Setting the Encoding of the html Serializer

```
<map:serializers default="html">
   <map:serializer name="html"
                mime-type="text/html"
               src="org.apache.cocoon.serialization.HTMLSerializer">
      <encoding>ISO-8859</encoding>
   </map:serializer>
</map: serializers>
```

Using the `encoding` parameter, you can define the character set used for the used for the output document. The default is Unicode (UTF-8). Especially for non-English sites, where many special characters are used, setting the encoding to the correct value is very useful. This allows the client to display special characters correctly.

The setting of the encoding is applied to every use of the serializer. It is not possible to override this setting, so your whole application uses this encoding.

Although we looked at the html serializer first, because newer versions of browsers support the display of XML data directly, you could use a different serializer, the xml serializer, to send the data back to your application.

The xml Serializer

The simplest serializer possible is the xml serializer. It simply generates text output from the XML document and sets the MIME type to `text/xml`. It is configured in the sitemap the same way all components are, as you can see in Listing 4.20. Note the use of the `type` attribute when using the serializer in the pipeline, because it is not the default serializer.

Listing 4.20 The xml Serializer

```
<map:serializers>
   <map:serializer name="xml"
                mime-type="text/xml"
               src="org.apache.cocoon.serialization.XMLSerializer"/>
<map:serializers/>
...
<map:serialize type="xml"/>
```

Most browsers available today are able to display XML data. This brings us to the first way of debugging pipelines. Should your pipeline not produce the output you are expecting, one way of checking the process flow is by removing the stylesheet transformation and then having the xml serializer send the XML to the browser, where it is displayed as XML. To do this, just add the `type` attribute with a value of `xml` to the `map:serialize` in the pipeline (you are changing the used serializer from HTML to XML).

Because the transformation step is now omitted, the XML you receive in the browser is the XML data that the generator read or generated. So, if the XML is as you expected, there is probably something wrong with the stylesheet that was being used to transform the data. You can correct the stylesheet and add the transformation step back into your

pipeline. Don’t forget to change the serializer type from XML back to HTML or whatever output format you were generating.

This brings us to the end of the description of the most common components used in a pipeline. The next component we need to look at more closely is the component that tells Cocoon that your pipeline can process a certain request. This is done using a matcher.

The Wildcard Matcher

As explained earlier, the default matcher is the wildcard matcher, which matches the incoming URI against a pattern. This matcher uses common wildcards, so the pattern doesn’t need to be static, like `helloworld`. You can define patterns that match against several document names, such as all the document names that have the same prefix.

Let’s look at some of those patterns. An asterisk matches flat parts in the URI, meaning that zero or more characters up to the occurrence of a slash. The slash is used as a path separator. Using two asterisks matches hierarchical parts, which means zero or more characters including slashes.

For example, assume that the document name in a request is products/books/ Cocoon and that you have a matcher with the pattern `"products/*/*"`. This pattern matches the request. In the pipeline surrounded by this match, you need to know the current values of the asterisks. To get those values, Cocoon uses a mechanism called *value substitution*. It allows you to use placeholders in the pipeline that will then be replaced with the actual values when the pipeline is called. Therefore, the matcher adds two keys/placeholders for value substitution, `1` and `2`, with the values `books` and `Cocoon`. You will learn more about value substitution later.

If you had chosen the pattern `"products/*"`, that would not match, because the asterisk matches only flat parts, and `"books/Cocoon"` is hierarchical. The pattern `"products/**"` *would* match, so the matcher would provide the key `1` with the value `books/Cocoon`. Because the matcher is a component just like the others you have seen in this chapter, it must ➥be configured in the sitemap, as shown in Listing 4.21.

Listing 4.21 The Wildcard Matcher

```
<map:matchers default="wildcard">
    <map:matcher name="wildcard"
src="org.apache.cocoon.matching.WildcardURIMatcherFactory"/>
</map:matchers>
...
<map:match pattern="news"/>
<map:match pattern="products/**"/>
<map:macth pattern="**.product"/>
```

Wildcard matching, based on common wildcards, is very powerful. However, it’s also easy to get wrong when designing your pipelines. When you start using pattern matching,

it is very likely that at some point you will request a document but get a different response than what you expected.

Let's look at Listing 4.21. There are three matches. The first one uses the static pattern `news`. This matches only if a document called news is requested. The second match matches everything that starts with products/. The third match matches everything that ends with .product. If a document called products/cocoon.product is requested, the second and third matches are successful.

In these situations, the order of the matches in the sitemap is important. The rule of thumb is that the first matcher wins, because only one can.

In many cases, this situation occurs because of using patterns that are too general. The pattern matches more documents than you want. For example, the apparently simple pattern `"**"` matches *every request*. If you have a request that will match more than one pattern in your sitemap, only the first pattern will be processed. The second pattern that would also have matched is never reached! Of course, this is true only if the first match leads to a serializer, which stops the sitemap processing.

You have to be very careful when defining patterns. A pattern should only include the documents that will really match. And if you add new documents to the sitemap, you must make sure that there is not already a pattern that matches the new document name.

Now that we have finished our first look at the four different component types we started out describing using the sitemap example with the Hello World example, now is perhaps a good time for you to go back to that example and have another look at it in light of the explanations we have given you in the preceding sections. Don't worry; we will still be here when you come back. Then we will show you how Cocoon can reuse components by allowing their configuration.

Configurable Components

One of the goals of a component-based architecture is to be able to reuse components instead of writing a dedicated component for each purpose. This is also true of the Cocoon architecture and the components included. For example, Cocoon has an xml serializer that simply serializes the processed XML to a text stream containing the XML. However, if you want to output different XML languages, such as WML, XHTML, or VoiceXML, these languages have different document types, different MIME types, and perhaps different settings that might influence the layout (such as indenting). Therefore, it might seem natural to have a separate serializer for each format, but there is an easier way. Cocoon offers a more modular and reusable approach by allowing the xml serializer to be configured with different parameters.

For example, the wml serializer is implemented using the xml serializer (see Listing 4.22). In contrast to the simple xml serializer, the wml serializer gets its own configuration. Instead of having a separate component, you can use parameters such as `mime-type` and

doctype-public to add the configuration you need so that a device such as a mobile phone will recognize the format and be able to display it.

Listing 4.22 The wml Serializer

```
<map:serializer name="wap"
                mime-type="text/vnd.wap.wml"
                src="org.apache.cocoon.serialization.XMLSerializer">
    <doctype-public>-//WAPFORUM//DTD WML 1.1//EN</doctype-public>

    <doctype-system>http://www.wapforum.org/DTD/wml_1.1.xml</doctype-sy
    stem>
</map:serializer>
```

Using the MIME type, the device can detect that the document is WML, and, using the document type, the document you send can also be verified. When the device receives the information about the document type, it can load the definition and check that the data it received conforms to that description. This can be achieved by the parser loading the DTD file from the address you configured in the example, and then using the DTD to validate the data.

Any configuration information that is added to a component, such as the serializer, when it is defined in the appropriate section in the pipeline is automatically available when the component is used in the pipeline. Many Cocoon components can be configured this way. Appendix A contains more details on the various possibilities.

Configuration parameters are valid for all instances of the component, but what about being able to configure a component when it is used in the pipeline? You do this by adding parameters when the component is used.

Parameters

Each sitemap component can have additional parameters in a pipeline. These parameters can be used to configure the component for this pipeline or to control the behavior of the specific instance.

A parameter is added to a component using the element map:parameter. This element has two attributes: name and value. To pass a parameter called "host" with a value of "myhost.com", you would write <map:parameter name="host" value="myhost.com"/>. Listing 4.23 shows how to pass the parameter use-request-parameters to the xslt transformer when it is used in a pipeline. For example, this parameter allows the stylesheet to access the request parameters while in the process of transforming the XML data.

Listing 4.23 Defining Parameters for a Sitemap Component

```
<map:generate src="helloworld.xml"/>
<map:transform src="stylesheet.xsl">
```

```
    <map:parameter name="use-request-parameters" value="true"/>
</map:transform>
<map:serialize/>
```

Many of the components have specific parameters you can use in this way. Appendix A contains a list of components and their parameters.

Adding parameters to a component when you use it allows very flexible pipelines to be built, which is fine as long as they work as expected. Now is the time to introduce the Cocoon concept that helps you when things do not function as they should: error handling.

Error Handling

Hopefully, up until now, things have been easy. By this we mean that everything has worked as expected, and no errors have occurred. But what if an exception *does* occur? As with any other system, there are pitfalls and mistakes you can make. Here are a few:

- The requested pipeline does not exist.
- A configured generator cannot read the XML data.
- The XML file to be read does not contain valid XML.
- The configured stylesheet is missing.
- The configured component does not exist.
- The sitemap does not contain valid XML.

All these errors result in Cocoon's being unable to process the request and an error being returned. This error (such as `sitemap handler is not available`) is displayed in the browser, along with additional information about the error. This information is generated by the configured error-handling routines.

Each `map:pipeline` section can define its own error handling. In addition to the flow definition inside a pipeline section, you can define an error handler to react to any error that might occur during the pipeline processing.

This error handler is defined by the tag `map:handle-errors` and is itself an XML-based pipeline. It is processed each time a critical error occurs, such as an HTTP timeout to get external content, a Java exception inside a component, one of the errors just explained, or any other error.

This error pipeline consists of only transformers and a serializer. No generator is necessary, because the "generated XML" is constructed by the reason for the error. Thus, this pipeline automatically has a generator called the error generator. You must not define a generator for the error pipeline.

Apart from this, the pipeline can be built just like any other, using any sitemap components you want, including matchers and also other components we have not yet

talked about. In Listing 4.24, the pipeline uses a stylesheet to transform the error into XHTML, and the default serializer (HTML) is then used to return the error message.

Listing 4.24 An Example of an Error Handler

```
<map:handle-errors>
    <map:transform src="stylesheets/error2html.xsl"/>
    <map:serialize status-code="500"/>
</map:handle-errors>
```

Whenever an error occurs during document processing, internally a Java exception is raised. Cocoon catches this exception, and the error generator converts it into an XML document.

An XML processing chain is built up by the error generator, and all sitemap components are defined in the `map:handle-errors` element. Each `map:pipeline` section can define its own `map:handle-errors` pipeline.

The error handler can now display the exception in any format by using stylesheets to transform the XML, or it can display the same static error page indicating to the user that something has gone wrong (see Listing 4.25).

Listing 4.25 Sample Output for the Error Generator

```
<?xml version="1.0" encoding="UTF-8" ?>
<error:notify error:type="error"
     error:sender="org.apache.cocoon.sitemap.ErrorNotifier"
       xmlns:error="http://apache.org/cocoon/error/2.0">

   <error:title>Error creating the resource</error:title>
   <error:source>org.apache.cocoon.ProcessingException</error:source>
   <error:message>Failed to execute pipeline.</error:message>
   <error:description>org.apache.cocoon.ProcessingException:
      Failed to execute pipeline.: java.lang.RuntimeException:
      Problem in getTransformer:Error in creating Transform Handler
   </error:description>
   <error:extra
    error:description="exception">org.apache.cocoon.ProcessingException
    :
      Failed to execute pipeline.: java.lang.RuntimeException:
      Problem in getTransformer:Error in creating Transform Handler
   </error:extra>
</error:notify>
```

The XML document generated by the error generator looks similar to this output. All elements and attributes use the namespace http://apache.org/cocoon/error/2.0. The root element has the name `error`. Inside this element are several pieces of information concerning the reason for the error, such as a title and the source. All this information can then be transformed using a stylesheet.

Remember that each `map:pipeline` section can have its own error handler. As explained earlier, this error handler can be customized like any other XML processing pipeline except the implicit error generator, so it is up to you to define how this error is displayed by defining the stylesheet and the sitemap components.

For web applications, it is common to set the status-code of the response to `500`. This tells the browser that an error occurred. You can set this code by using the attribute `status-code` of the serializer.

If no matching document definition is found in the sitemap for the request, Cocoon automatically returns to the browser with a status code of `404`. This code indicates that the requested document does not exist. You can refer to the standard sitemap for examples of how the error handling is used.

Now that you have seen how error handling works, you are well-prepared to tackle some examples. These first examples use what you have learned so far.

Basic Examples Using Cocoon

Now, what you've been waiting for: It's reader participation time. The following examples show you how to put Cocoon to work and demonstrate its capabilities. We will start by building a pipeline that converts someone else's HTML to look as though it is your own. You will then build a picture gallery and extend it so that you can present the pictures in a personalized fashion.

As is the case with all the examples in this book, these examples should give you some ideas for your own applications and also point you in the right direction for when you think about what you want to do with Cocoon. We encourage you to adapt the examples for your own use and perhaps change or add things here and there to see how Cocoon reacts.

Someone Else's HTML

Integrating data sources is one of Cocoon's major strengths. A typical data source is a web server that can already serve HTML pages. This example takes the HTML generation from the Hello World example one step further. This time you will let someone else generate the HTML page for you. That's right—it's web-napping time. You will access someone else's web page and then use it in your own Cocoon application.

Of course, we have permission to use this example, and you should get permission as well before including a remote web page in your Cocoon installation.

Listing 4.26 shows the fragment for the sitemap.

Listing 4.26 A Sitemap Fragment

```
<map:pipeline>
    <map:match pattern="fiery">
       <map:generate
     src="http://www.flatbooks.com/frameset/con_fier.htm" type="html"/>
       <map:transform src="fiery.xsl"/>
       <map:serialize/>
    </map:match>
</map:pipeline>
```

As you can see, it is pretty straightforward. We give the html generator the link to access in order to obtain the web page. Then we add the xslt transformer with the appropriate stylesheet, as shown in Listing 4.27.

Listing 4.27 The Stylesheet to Alter the Retrieved HTML

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

<xsl:template match="table">
<html>
<center>
     <h1>Fiery Food - "found" by Cocoon</h1>
     <h3>Original <a
     href="http://www.flatbooks.com/frameset/con_fier.htm">here</a></h3>
</center>
<body bgcolor="yellow">
     <xsl:apply-templates/>
</body>
</html>
</xsl:template>

<!-- This gets rid of "dirt" in the page -->
<xsl:template match="script">
</xsl:template>
<xsl:template match="title">
</xsl:template>

<xsl:template match="td">
     <xsl:value-of select="."/>
     <p/>
</xsl:template>

</xsl:stylesheet>
```

Save the stylesheet in the Cocoon context directory and name it fiery.xsl. Restart Cocoon and point your browser to `http://localhost:8080/cocoon/fiery` to see someone else's information in a completely different layout, as shown in Figure 4.5.

Figure 4.5. A stylesheet alters the layout.

Cocoon accesses the web page and then formats the HTML as XHTML using JTidy. Next, the stylesheet extracts the information from the XHTML format and adds a few HTML tags to give the information a different look.

As you can see, extracting data from HTML pages is tricky, because it is not easy to see what each piece of data means when it has been formatted as, say, a table. Nevertheless, using the html generator, Cocoon provides a convenient way of including available HTML pages in an application that might otherwise be completely built with XML and XSL.

You can use this example and experiment with other web pages. Just change the link you give the html generator to the page you want to access. Remove the stylesheet and change the serializer to type `xml`. The XHTML representation of the page is sent to your browser. As soon as you have it, you can design the XSL stylesheet to fit.

Picture Gallery

This example uses the imagedirectory generator to build a picture gallery from a directory of images.

The first step is to create a directory called gallery below the Cocoon context directory. Copy some JPEG or GIF images into this directory. Next, add the fragment shown in Listing 4.28 to your sitemap.

Listing 4.28 A Sitemap Fragment

```
<map:pipeline>
    <map:match pattern="gallery">
        <map:generate src="gallery" type="imagedirectory"/>
        <map:transform src="gallery.xsl"/>
        <map:serialize type="html"/>
    </map:match>
    <!-- Gallery Images -->
    <map:match pattern="gallery/**">
        <map:read src="gallery/{1}"/>
    </map:match>
</map:pipeline>
```

Notice that this example contains two `map:match` sections. The result of the first `map:match` is an HTML page that presents a thumbnail representation of the pictures stored in the directory. Because the browser loads these pictures through Cocoon, you need to make Cocoon aware of this.

This is done by the second `map:match`. Here you tell Cocoon that a request for anything below the gallery directory should be returned using the default reader. Don't worry that you don't yet know what a reader is. We will explain it later. For now, just remember that it basically just reads a file and returns it to the browser as is.

Next is a stylesheet that formats the output of the imagedirectory generator as a simple HTML page (see Listing 4.29).

Listing 4.29 A Gallery Stylesheet

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

<xsl:template match="/">
<html>
<body>
<center>
<h1>The Gallery</h1>
</center>
<xsl:for-each select="dir:directory/dir:file">
     <img src="{/dir:directory/@name}/{@name}" width="100"
     height="100"/>  
</xsl:for-each>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

The first thing to note is the use of namespaces in this stylesheet. The imagedirectory generator uses a distinct namespace for all the tags it returns, so the stylesheet must also declare the namespace and use it when referencing the tags.

In order to separate the pictures from one another, we opted to simply stick two spaces between them using the XML space ( ) notation.

Save the stylesheet to the Cocoon context directory and name it gallery.xsl. Restart Cocoon and point your browser to `http://localhost:8080/cocoon/gallery` to see a thumbnail gallery of your pictures, as shown in Figure 4.6.

Figure 4.6. The gallery.

Now that you can put your favorite pictures on the web using Cocoon, wouldn't it be great to add some form of personalization to your gallery?

Personalized Picture Gallery

You have set up your family picture gallery on the web. You have told all your friends and relatives where to go to see all the great pictures. And then people start calling you and complaining about the colors you used.

The first person who calls wants a brighter color. You start changing it, and when you are finished, your uncle calls to say he wants darker colors. And that's the point when you start thinking about personalization. Let the user choose which color he sees rather than hard-coding it and changing it every day.

Cocoon offers several ways of adding personalization. The next example uses a simple way of doing this. We will show you some other ways when we expand our examples in the following chapters. This brings up a point worth stressing: Due to the flexibility of the

Cocoon architecture, there are often several ways of integrating a certain functionality. So, even though we will show you one way, you can be sure that there are others.

Let's look at the problem of choosing colors. First, you'll add a small color chooser to the document. This is a list of links in which each link displays the gallery in the corresponding color. Rather than defining a separate pipeline for each color, you will define only one pipeline and then use the color as a parameter.

To achieve this, you will use a request parameter called `color`. Then you can access the gallery through `http://localhost:8080/cocoon/mygallery?color=red`. If no parameter is specified, you will use a default color.

The `color` parameter in turn is used in the stylesheet to set the background color. As explained, some sitemap components can be configured using parameters inside the pipeline. The xslt transformer is one such component. When you specify the `use-request-parameters` parameter, all request parameters are then available in the stylesheet (see Listing 4.30).

Listing 4.30 A Sitemap Fragment for the Personalized Picture Gallery

```
<map:pipeline>
    <map:match pattern="mygallery">
        <map:generate src="gallery" type="imagedirectory"/>
        <map:transform src="mygallery.xsl">
            <map:parameter name="use-request-parameters" value="true"/>
        </map:transform>
        <map:serialize type="html"/>
    </map:match>
    <!-- Gallery Images -->
    <map:match pattern="gallery/**">
        <map:read src="gallery/{1}"/>
    </map:match>
</map:pipeline>
```

As shown in this sitemap fragment, the pipelines look nearly the same as in the picture gallery, except that your document is now named mygallery and the stylesheet is called mygallery.xsl. This stylesheet has access to a request parameter if an XSL parameter with the same name is declared on the top level of the stylesheet. So the stylesheet looks like Listing 4.31.

Listing 4.31 The Personalized Picture Gallery Stylesheet

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

<!-- Make the request parameter available -->
<xsl:param name="color"/>
<xsl:template match="/">
```

```
<html>
    <body>
       <xsl:choose>
          <xsl:when test="$color=''">
             <xsl:attribute name="bgcolor">#FFFFFF</xsl:attribute>
       </xsl:when>
       <xsl:when test="$color='red'">
          <xsl:attribute name="bgcolor">#FF0000</xsl:attribute>
       </xsl:when>
       <xsl:when test="$color='black'">
           <xsl:attribute name="bgcolor">#000000</xsl:attribute>
       </xsl:when>
       <xsl:otherwise>
           <xsl:attribute name="bgcolor">
               <xsl:text>#</xsl:text><xsl:value-of select="$color"/>
           </xsl:attribute>
        </xsl:otherwise>
    </xsl:choose>
    <center><h1>MyGallery</h1></center>
    <xsl:for-each select="dir:directory/dir:file">
       <img src="{/dir:directory/@name}/{@name}"
            width="100" height="100"/>  
    </xsl:for-each>
    <br/>
    <a href="mygallery?color=red">I want red</a><br/>
    <a href="mygallery?color=black">I want black</a><br/>
    <a href="mygallery?color=AAAAAA">I want grey</a><br/>
    <a href="mygallery">I want white</a><br/>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

The stylesheet is also very similar to the one we used in the previous example. In addition, it declares a global parameter named `color` to get the value of the request parameter with the same name. The document's background is set using this parameter with the `xsl:choose` statement. If no parameter is defined, the background is white.

Save the stylesheet to the Cocoon root directory and name it mygallery.xsl. Restart Cocoon and point your browser to `http://localhost:8080/cocoon/mygallery?color=` followed by the color's RGB value, such as 333333. The results are shown in Figure 4.7.

Figure 4.7. The color gallery.

This completes our first look at some examples. We will now show you some additional components that are available in Cocoon that can be used to enhance the pipelines you have built so far.

More Sitemap Components

So far you have learned about the most common sitemap components and how to use them to build pipelines. But let's extend your knowledge a bit by showing you some more sitemap component types so that you can build more-complex pipelines. We will look at the selector component, the action component, and the reader component. The first two let you control what happens in your pipeline, and the reader makes it easier to return information such as pictures.

Selectors

Like matchers, selectors are sitemap components that can be used to determine the flow through the sitemap. The selector is a special component that allows Cocoon to differentiate between certain aspects of the client application or system and respond to that data. After they are placed in a `map:pipeline` section, selectors allow Boolean evaluations to be performed and reactions to be configured accordingly.

The most common example of a selector is the browser selector, which allows the sitemap flow to be dependent on the used browser or device. In fact, this selector uses the device's user agent to determine the client. This user agent is sent by applications such as browsers with the request for the document.

Whereas a matcher can be seen as a simple `if` statement, selectors can be used in rather complex evaluations that you might be familiar with from `if-elseif-else` statements in common programming languages.

As with the `xsl:choose` statement of the stylesheet language, you can add an arbitrary number of test cases. Each case is added using a nested `map:when` element. The attribute `test` contains the value that should be tested by the selector. The cases are evaluated from top to bottom. If one case is equal, the elements inside the `map:when` are processed next. All other `map:when`s are ignored. You can also specify a default case using the `map:otherwise` element. This section is processed only if no case matches (see Listing 4.32).

Listing 4.32 A Browser Selector Example

```
<map:generate src="content.xml"/>
<map:select type="browser">
    <map:when test="explorer">
        <map:transform src="stylesheets/iehtml.xsl"/>
        <map:serialize/>
    </map:when>
    <map:when test="lynx">
        <map:transform src="stylesheets/text.xsl"/>
        <map:serialize type="xml"/>
    </map:when>
    <map:when test="netscape">
        <map:transform src="stylesheets/nshtml.xsl"/>
        <map:serialize/>
    </map:when>
    <map:otherwise>
        <map:transform src="stylesheets/html.xsl"/>
        <map:serialize/>
    </map:otherwise>
</map:select>
```

In this sample pipeline fragment, the XML file content.xml is read and then transformed and serialized according to the used device. If the device is a browser, either Internet Explorer or Netscape Navigator, the appropriate stylesheet is used. If the device is lynx, the content is transformed into XML. And in all other cases, simple HTML is created.

Notice that the reading of the file content.xml is not dependent on the browser and therefore is the same for all clients. This allows true separation of content and layout. A complete example using this selector is described later in this chapter.

As you can see with this example, using selectors and matchers allows you to define complex pipelines. This gives the sitemap the same flexibility you have in structured programming languages. Another important component that increases the flexibility of what you can do with a pipeline is the action component.

Actions

The sitemap components shown so far share one common feature: They influence the result of the request—the document. This is done either by controlling the flow or by taking part in the XML pipeline.

When building applications, however, you often need to perform some tasks that do not directly influence the document. For example, if you want to build a shop with Cocoon, you might need to add a new user to your database, add items to your shop-ping cart, and so on. That's where actions come into the picture.

The definition of a Cocoon action is very simple: An action performs a defined task. An action does not produce any display data and does not take part in the XML processing pipeline. Because the possible range of actions is unlimited, nearly everything can be done with actions. Cocoon offers several actions for accessing request information such as cookies, for communicating with databases, and for authorizing users in a portal. An action can also control the flow and provide information to other sitemap components. This makes actions even more powerful.

An action is declared with the `map:action` element inside the `map:actions` element in the sitemap. For example, `<map:action name="resource-exists" src="org.apache.cocoon.acting.ResourceExistsAction"/>` defines an action called `resource-exists` that is implemented by the given class. This action will be used later in an example to test whether a given file exists on the server, before it is read, transformed, and returned to the client application.

Another way of accessing a file is by using a component we introduced earlier in the gallery example but did not explain further—the reader.

Readers

Up to this point, we have only dealt with sitemap components for processing XML or controlling the processing flow. However, real-world applications often need documents that are not XML-based, such as images, movies, or JavaScript files.

You could use generators, transformers, and serializers to produce such formats, of course. For example, simple dynamic images can easily be built using SVG, as you will see later in the examples. But this is not always possible and often is not desirable, because images or movies are created with special software and are then in a binary format. The same applies to such formats as JavaScript and Word documents.

The `reader` component was designed for when the document is already in the desired presentation format. The reader simply reads a resource and then streams it, as is, back to the device or application. In addition, the reader can set the appropriate MIME type.

So a reader component can be seen as a combination of a special generator and serializer: The generator reads the binary format and converts it into XML, and the serializer

converts this XML format back into the binary format. Thus, the reader eliminates the need for a processing pipeline between these components.

A reader is declared with the `map:reader` element inside the `map:readers` element in the sitemap. For example, `<map:reader name="resource" src="org.apache.cocoon.reading.ResourceReader"/>` defines a reader called `resource` that is implemented by the given class.

With the introduction of these new components, you are ready to take another look at the pipelines and fill in more details on how they work.

Pipelines Revisited

When we first looked at the pipelines section in the sitemap, we talked about the concepts that allow a request to be handled by a specific pipeline. Now we will take a more detailed look at how a request is matched to a pipeline and how you can use the different parts of a request inside the pipeline for greater flexibility.

Pattern Matching

The examples so far have concentrated on "direct matches." For example, if you have three documents called news, jobs, and products, you would create three matcher directives—one for each document.

If you think of a real-world application, you might have hundreds or even thousands of documents. It would be a nightmare to create a matcher directive for each available document. In fact, the very concept of a sitemap would be questionable.

Using sophisticated matchers such as the wildcard matcher allows you to use pattern matching based on wildcards. For example, assume that the three documents just mentioned all create a similar pipeline: A file generator reads an XML document (called news.xml, jobs.xml, or products.xml), the same stylesheet is used, and the same serializer finishes the job. In this case, you can simply write one matcher directive, as shown here:

```
<map:match pattern="*">
  <map:generate src="{1}.xml"/>
  <map:transform src="stylesheet.xsl"/>
  <map:serialize/>
</map:match>
```

What happens here? The wildcard matcher uses common wildcards to test the pattern. So matching with an asterisk matches any flat document name, which means that the document name should not contain a path. Therefore, all three requests—jobs, news, and products—do match.

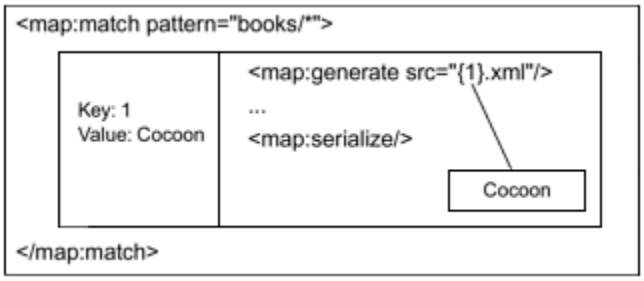
Inside the `match` element, you need to know what the document name is. The pattern matcher outputs this information under a distinct key to the included components. This key has the name `1`. Using the integrated value substitution provided by Cocoon, the document name is substituted for `{1}`, so either news.xml, jobs.xml, or products.xml is read.

The pipeline continues by applying the same stylesheet to the document and then serializing the result to HTML.

Value Substitution

Cocoon's value-substitution mechanism (see Figure 4.8) is very useful inside a pipeline. Actions and matchers can provide key-value pairs that can be used by all other sitemap components.

Figure 4.8. Value substitution.



The mechanism uses scopes for the validity of key-value pairs. If an action or a matcher provides a pair, this pair is available only for components that are inside the element of the matcher or action. No other components have access to the pair.

The nested components can retrieve the value only if they know the key. When you write the name of the key enclosed in curly brackets, this tag is substituted when the sitemap is executed.

To make this as simple as possible, matchers usually enumerate their values starting with `1`, but generally a matcher can define any key name.

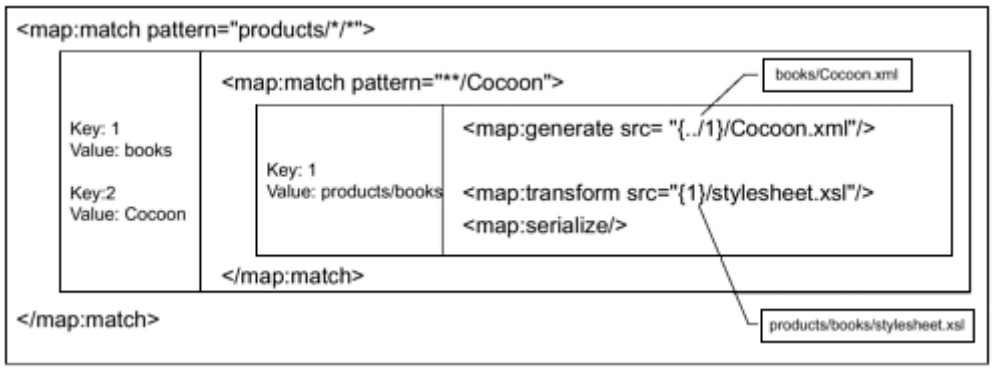
Because actions can be a lot more complex, they do not adhere to this rule. The key names used by actions are totally action-dependent.

So if you want to use a matcher or an action, you have to know which keys to use to access the needed values. Otherwise, you cannot use value substitution in your pipeline.

Furthermore, it is also possible to nest these components (see Figure 4.9). For example, you can nest two matchers. Inside the outer matcher, you can reference the value simply

by using brackets: {1}. If you use the same key inside the inner matcher, you get the value of the inner matcher.

Figure 4.9. Complex value substitution.



How can you get values from outer components? The substitution mechanism allows paths for the keys, just as they are used for directory structures. So, using {1} refers to the "key of current level." Using {../1} means "key of above level," which tries to get the value of the outer match.

Even if the nested components use different names for their keys, you have to specify the path for the keys. So substituting a value from a top-level component happens only if the correct level is selected.

More About the Processing Flow

So far you have seen a simplified view of how a request flows through a sitemap. Now is the time to extend that view by adding more details. The sitemap spawns a virtual URI space. This space is created by the different sitemap components—most notably, the wildcard matcher.

Because the pipelines section of the sitemap is processed top-down, the flow can be described as follows:

- 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 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. 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 executed next.
- If a selector directive is found, the selector performs the various test cases from top to bottom. When a test case is verified, the directives inside this case are executed next, and all others are ignored. If no test case matches, the default case (if 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.
- If a transformer directive is found, it transforms the XML document, and the next directive on the same level is executed.
- If a serializer directive is found, it serializes the XML document, and the processing is finished.
- If a reader directive is found, the reader delivers the document, and the processing is finished.
- If an error occurs, the error handler of the current `map:pipeline` is called.

As you can see by this description of an abstract flow through a pipeline, defining the sitemap is just like writing a program in a structured programming language—only on a much higher level. You have to choose the right components, and then you have to combine them in the correct way in order to get the result you want.

When you start building more-complex applications, you will find the concepts and components discussed in this section very helpful, because they allow you to build pipelines that cater to a variety of requests and provide you with greater flexibility inside them. Now is the time to present some additional components and add some examples that use the new concepts you have just learned about.

Advanced Components and Examples

You have now learned about all of Cocoon's important basic concepts in order to use it to build your own web site. Now let's use this knowledge for some more-advanced examples that use everything we've talked about so far. You will see how Cocoon provides components that let you generate PDF documents or select the correct output format based on information that the end device or application sends you. Additional examples will show you how to configure pipelines to generate PDF using the new component. Then you will enhance your gallery example.

Components

When building applications that can publish to a variety of formats, it is important that the system provide components that help you do this.

Cocoon provides components that can generate the PDF document format and JPEG images from standardized XML formats. The selector component type allows the flow through a pipeline to be influenced by data that has perhaps been sent with the request. Together these components aid in the building of multichannel applications.

PDF Serializer

One of Cocoon's most interesting capabilities is that it can generate PDF from XML data using a stylesheet and the correct serializer. The way this works is slightly more complicated than the examples you have seen so far, so let's look at it in a bit more detail.

The first thing to note is that Cocoon uses another Apache project to do this. The project is called FOP (Formatting Objects to PDF). It is a Java print formatter driven by XSL Formatting Objects (XSL:FO). As explained in Chapter 2, “Building the Machine Web with XML,” XSL:FO is defined as part of the XSL standard.

XSL:FO is a standardized way of describing document layout. So, as soon as you have used a stylesheet to transform your XML data into XSL:FO, you can then use a component such as the PDF serializer to create the PDF format. The PDF serializer is a Cocoon component that acts as a wrapper around other components from the FOP project. This is a common way of integrating external components into Cocoon, and it allows them to be reused. Other serializers in Cocoon can create other formats, such as PostScript.

XSL:FO is quite complex. Indeed, we could write a book just about that. Later in this chapter we will present a very simple example. For now, Listing 4.33 shows how the serializer is defined in the sitemap and how you would use it in a pipeline.

Listing 4.33 FOP Serializers

```
<map:serializers>
    <map:serializer mime-type="application/pdf" name="fo2pdf"
src="org.apache.cocoon.serialization.FOPSerializer"/>
</map:serializers>
...
<map:serialize type="fo2pdf"/>
```

Remember that the serializer does not magically generate PDF. You need to use a stylesheet and the xslt transformer to generate the XSL:FO format first.

SVG Serializer

Another sophisticated sitemap component is the svg serializer. SVG (Scalable Vector Graphics) is a W3C standard. It is an XML language that describes graphics. The SVG engine interprets XML documents containing SVG commands such as “Put a rectan-gle there, fill it with red, and lay a text with this font over it.”

As with the PDF serializer, the XML data must already be in the standardized SVG format before the serializer can act on it. Then the serializer generates a binary image format from the SVG format. More precisely, Cocoon offers a series of serializers that can convert SVG documents into images. Each serializer is a wrapper around components from the Apache SVG project, Batik. Different serializers exist for each different image format. Two are included in Cocoon by default—the svg2jpeg serializer and the svg2png serializer, which convert SVG into JPEG or PNG. In addition, the serializers set the correct MIME type for the images. Listing 4.34 shows how they are defined in the sitemap and how you would use them.

Listing 4.34 SVG Serializers

```
<map:serializers>
    <map:serializer name="svg2jpeg"
                    mime-type="image/jpg"
                   src="org.apache.cocoon.serialization.SVGSerializer"/>
<map:serializers/>
...
<map:serialize type="svg2jpeg"/>
```

This example shows the configuration of the serializer in the sitemap and how to use the component in a pipeline. A typical use-case for the svg serializers is to dynamically create images, such as navigation bars where you don't know beforehand how many entries are available or what the titles of the items are.

Browser Selector

The multichannel challenge is perhaps one of the most interesting ones that modern application architectures face today. However, delivering the same document in different formats is very time-consuming and error-prone if you don't use Cocoon. If you use Cocoon, the process becomes easier, because you can use available components to build your multichannel application.

You define the content you want to display, write a stylesheet for each format you want to support (such as HTML or WML), and use the browser selector to control the flow inside the pipeline. That's it. Adding new formats or supporting different devices that interpret a standard slightly differently is just as easy. Just create a new stylesheet and add it to the pipeline.

You need a way of telling Cocoon what do to for each channel. Usually the content and logic are the same—only the layouts differ—so most of the pipeline can be reused. Only the layout part depends on the target format. This is where the browser selector, shown in Listing 4.35, comes into play. It allows Cocoon to choose the correct sitemap components depending on the user agent of the browser or device.

Listing 4.35 The Browser Selector

```
<map:selectors default="browser">
    <map:selector name="browser"

     src="org.apache.cocoon.selection.BrowserSelectorFactory"/>
    <!-- # NOTE: The appearance indicates the search order. This is very
        # important since some words may be found in more than one browser
        # description. (MSIE is presented as "Mozilla/4.0 (Compatible; MSIE
        # 4.01; ...")
    -->
    <browser name="explorer" useragent="MSIE"/>
    <browser name="pocketexplorer" useragent="MSPIE"/>
    <browser name="handweb" useragent="HandHTTP"/>
    <browser name="avantgo" useragent="AvantGo"/>
    <browser name="imode" useragent="DoCoMo"/>
    <browser name="opera" useragent="Opera"/>
    <browser name="lynx" useragent="Lynx"/>
```

```
    <browser name="java" useragent="Java"/>
    <browser name="wap" useragent="Nokia"/>
    <browser name="wap" useragent="UP"/>
    <browser name="wap" useragent="Wapalizer"/>
    <browser name="mozilla5" useragent="Mozilla/5"/>
    <browser name="mozilla5" useragent="Netscape6/"/>
    <browser name="netscape" useragent="Mozilla"/>
</map:selectors>
...
<map:generate src="helloworld.xml"/>
<map:select>
    <map:when test="explorer">
        <map:transform src="stylesheets/iehtml.xsl"/>
        <map:serialize/>
    ➥</map:when>
    <map:when test="lynx">
        <map:transform src="stylesheets/text.xsl"/>
        <map:serialize type="text"/>
    </map:when>
    <map:when test="netscape">
        <map:transform src="stylesheets/nshtml.xsl"/>
        <map:serialize/>
    </map:when>
    <map:otherwise>
        <map:transform src="stylesheets/html.xsl"/>
        <map:serialize/>
    </map:otherwise>
</map:select>
```

As you can see in the snippet from the sitemap, the browser selector is a configurable component. It is configured in the `map:selectors` section of the sitemap. It gets a set of definitions for the different devices and browsers available. You can also define sym-bolic names (such as `wap`) for a group of browsers that have the same user agent.

When the browser selector is used in a pipline, it tests its configuration from top to bottom. If the user agent contains the given text for a configuration entry, the browser is detected. Otherwise, the search continues. In our example, this means that different stylesheets are used, depending on the user agent. Note that the browser selector is the default selector, so you do not need to add the `type` attribute when you use the component in a pipeline.

Parameter Selector

Whereas the browser selector can be used to test the user agent sent by the client, the parameter selector, shown in Listing 4.36, can test any value available in the current sitemap pipeline.

Listing 4.36 The Parameter Selector

```
<map:selectors>
    <map:selector name="parameter"
src="org.apache.cocoon.selection.ParameterSelectorFactory"/>
```

```
</map:selectors>
...
<map:act type="getAValueForUserName">
    <map:select type="parameter">
        <map:parameter name="parameter-selector-test" value="{userName}"/>
        <map:when test="administrator">
           <!-- This is the administrator -->
        </map:when>
        <map:otherwise>
        </map:otherwise>
     ...
    </map:select>
</map:act>
```

For example, using this selector you can test values set by an action or a matcher. You set a parameter named `parameter-selector-test` on the selector and give it the value to test.

For each value you want to test, create a separate `map:when` statement. Add a default rule by specifying a `map:otherwise` section. In Listing 4.36, the fictional action `getAValueForUserName` gets the name of the current user and makes it available for nested components using the key `userName`. The parameter selector is nested inside the action. It is configured with the sitemap parameter `parameter-selector-test`, getting the current value for the key `userName`.

The following `map:when` statement tests this value against `administrator`. If it is true, the statements inside this branch are executed. If the name is not `administrator`, the `map:otherwise` section is processed next. So this selector helps you control the flow through the sitemap by any value available.

Resource-Exists Action

With the `resource-exists` action, you can test to see whether a resource is available. This action is very useful in combination with pattern matching. If you use the asterisk as a pattern, remember that every document will match. (Refer to the example introduced in the "Pattern Matching" section, in which we discussed the document news, jobs, and products.)

The example enhanced by this action is shown in Listing 4.37. Assume that you have only the three XML documents—news.xml, jobs.xml, and products.xml. If you request news, jobs, or products, everything works as expected, and you get the resulting HTML document.

If the document contact is requested, the pattern matches even though the source for this document does not exist. The matcher does not know anything about the processing pipeline. It can only match, and nothing more. Thus, the file generator tries to read an

XML file called contact.xml that is unavailable, and an error occurs. This error continues the processing in the error pipeline.

You can avoid this error caused by a too-generic pattern by using the `resource-exists` action. This action can be used to test whether the XML file exists. Only then is the pipeline built. If the XML file does not exist, a different XML pipeline can be processed, generating an error page.

Listing 4.37 The *resource-exists* Action

```
<map:actions>
   <map:action name="resource-exists"
               src="org.apache.cocoon.acting.ResourceExistsAction"/>
</map:actions>
...
<map:match pattern="*">
   <map:act type="resource-exists">
      <map:parameter name="url" value="{1}.xml"/>
      <!-- It's available -->
      <map:generate src="{1}.xml"/>
      <map:transform src="stylesheet.xsl"/>
      <map:serialize/>
   </map:act>
   <!-- Not found -->
   <map:generate src="DocumentNotAvailable.xml"/>
   <map:transform src="stylesheet.xsl"/>
   <map:serialize/>
</map:match>
```

As you can see from Listing 4.37, the `resource-exists` action gets one parameter—`url`. The value of this parameter is used to test whether a resource at the given location exists. In this case, you test to see if the XML document exists. If it is available, the usual XML processing pipeline is assembled, reading the XML document and transforming it to HTML.

If the resource does not exist, the components nested in the pipeline are skipped, and the processing continues right after the action. So Listing 4.37 uses a different XML processing pipeline. This pipeline always reads the same XML document, DocumentNotAvailable.xml, and transforms it into HTML.

So using the `resource-exists` action, you can interact with the document generation and control the flow to always generate a document.

Request Parameter Action

In one of the basic examples, we showed you how to access request parameters within a stylesheet. But you can also use request parameters in the sitemap.

With the `request` action, shown in Listing 4.38, you have access to several pieces of information contained in the current request. These values can then be used in the pipeline processing. For example, a `request` parameter can determine which XML file should be read, and so on.

Listing 4.38 The *request* Action

```
<map:actions>
   <map:action name="request
               src="org.apache.cocoon.acting.RequestParamAction">
</map:actions>
...
<map:act type="request">
   <map:parameter name="parameters" value="true"/>
   <map:generate src="product_{name}.xml"/>
   <map:transform src="stylesheet.xsl"/>
   <map:serialize/>
</map:act>
```

This action defines several keys that can be used in value substitution:

- `context`: The servlet's context path. This is the part of the URI a client uses that determines that the request should be handled by Cocoon. Usually this is `/cocoon`.
- `requestURI`: The name of the request document without all the parameters.
- `requestQuery`: The request's query string. This means all parameters that have their values prepended with a question mark, such as `?myparameter=avalue&name=book`.
- All `request` parameters: If the parameter named `parameters` is specified with the value `true`, all parameters are available for value substitution. Each parameter can be accessed by its name.

In addition, it is possible to set default values for a `request` parameter. When this parameter is not specified, the default value is used instead. This can be done by adding parameters with the name `default` and then appending the parameter name. For example, if you want to add a default value for a parameter called `name`, the line could look like this: `<map:parameter name="default.name" value="Cocoon Book"/>`.

Resource Reader

Cocoon's most-used and therefore default reader is the resource reader (see Listing 4.39). It can be used to deliver any data ranging from binary to text files to the client application.

The resource reader reads a file—on either a local or distant system—using the appropriate protocol and then returns it unchanged to the client. In addition, it sets the response's MIME type according to the type of file.

You can specify the optional `expires` parameter to set special HTTP headers, setting the document's validity. Using these headers, the browser can cache the document itself until the expires date is reached. The value is in milliseconds from the time the document was requested.

Listing 4.39 The Resource Reader

```
<map:readers>
    <map:reader name="resource
                src="org.apache.cocoon.reading.ResourceReader">
</map:readers>
...
<map:read src="images/background.gif"/>
<map:read src="images/background.gif"/>
  <map:parameter name="expires" value="3600000"/>
</map:read>
```

Listing 4.39 shows you how to configure the component in the sitemap and various ways of using it in the pipelines.

This completes our look at some additional components. We will now put them to work and use them in some further examples.

Examples

The following examples in this section use the new components to generate output formats such as PDF and SVG. We will also extend the gallery example to provide generated images and show you how you can use Cocoon to build an application that allows files to be downloaded from a server.

Your First PDF

In this example, you will generate your first simple PDF document. As with every example, the first step is to enter the pipeline into the sitemap, as shown in Listing 4.40.

Listing 4.40 A Pipeline Fragment

```
<map:pipeline>
    <map:match pattern="myfirstpdf">
        <map:generate src="myfirstpdf.xml"/>
        <map:transform src="myfirstpdf.xsl"/>
        <map:serialize type="fo2pdf"/>
    </map:match>
</map:pipeline>
```

This sample pipeline first reads an XML file and then uses a stylesheet to format the data as XSL:FO. The last step in the pipeline is the fo2pdf serializer, which generates the PDF. Next, you need the data you want to see in your document. Enter this into a new XML

file using your editor of choice, and save it to the Cocoon context directory (see Listing 4.41). Make sure you name it myfirstpdf.xml.

Listing 4.41 Data for the PDF Document

```
<?xml version="1.0"?>
<data>
    <name>Insert your name here</name>
</data>
```

As you can see from the data, you are just going to generate a document that contains a name.

Finally, Listing 4.42 shows the stylesheet that transforms the data into XSL:FO format. The data is then serialized into PDF by the fo2pdf serializer. For this example, we have kept the stylesheet as simple as possible. Even so, it is still quite large, and you must enter a lot of XML in order to set up the FOP. You might want to refer to the CD that comes with this book and copy the stylesheet into your Cocoon environment instead of typing it in. If you do enter the code into an editor, save the complete file as myfirstpdf.xsl.

Listing 4.42 The FOP Stylesheet

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <xsl:template match="/">
   <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <fo:layout-master-set>
     <fo:simple-page-master master-name="page"
                page-height="29.7cm"
                page-width="21cm"
                margin-top="1cm"
                margin-bottom="2cm"
                margin-left="2.5cm"
                margin-right="2.5cm">
       <fo:region-before extent="3cm"/>
       <fo:region-body margin-top="3cm"/>
       <fo:region-after extent="1.5cm"/>
     </fo:simple-page-master>

     <fo:page-sequence-master master-name="all">
       <fo:repeatable-page-master-alternatives>
       <fo:conditional-page-master-reference master-name="page"
     page-position="first"/>
       </fo:repeatable-page-master-alternatives>
     </fo:page-sequence-master>
    </fo:layout-master-set>

    <fo:page-sequence master-name="all">
      <fo:flow flow-name="xsl-region-body">
        <xsl:apply-templates/>
      </fo:flow>
```

```
    </fo:page-sequence>
   </fo:root>
  </xsl:template>

  <xsl:template match="data">
    <fo:block font-size="36pt" space-before.optimum="24pt"
text-align="center"><xsl:value-of select="name"/></fo:block>
  </xsl:template>

</xsl:stylesheet>
```

Now point your browser to `http://localhost:8080/cocoon/myfirstpdf`. If you typed in everything correctly, you should see your first PDF document. It should look something like Figure 4.10.

Figure 4.10. The first PDF document.

The stylesheet for this example formats the XML data into XSL:FO format. The serializer then converts it into the PDF you see in Figure 4.10. Because the serializer appends the correct MIME type to the PDF document, the browser knows to start the PDF viewer application to display the document.

Being able to generate PDF on-the-fly is a very important feature when it comes to building larger Cocoon-based applications, especially in commercial environments. Another feature your application might need to offer is to allow users to download certain files from the server.

Downloading

This example implements a simple download server using the components available in Cocoon. You want to allow the user to download the file only if he knows the server's address and the exact filename. After you have completed this example and installed it into your Cocoon environment, you can download a specific file via the address

`http://localhost:8080/cocoon/download?file=name`, where the name of the file is appended to the URI using `?file=`*`name`*. So to download the file called mypicture.jpg, you would enter the following link into your browser: `http://localhost:8080/cocoon/download?file=mypicture.jpg`.

In order to build your little download server, you need to use the action component type introduced earlier. Specifically, you will use the `request` action and the `resource-exists` action. The first action extracts the filename from the request by looking at the request parameter named `file`. The second action tests whether this file exists.

If the file is available, a reader reads it. If it is unavailable, a simple pipeline delivering HTML is processed which states that the file was not found on the server (see Listing 4.43).

Listing 4.43 A Pipeline Fragment

```
<map:pipeline>
   <map:match pattern="download">
      <map:act type="request">
         <map:parameter name="parameters" value="true"/>

         <map:act type="resource-exists">
            <map:parameter name="url" value="download/{file}"/>
            <map:read src="download/{file}"/>
         </map:act>
         <map:generate src="filenotfound.xml"/>
         <map:transform src="filenotfound2html.xsl"/>
         <map:serialize/>
      </map:act>
   </map:match>
</map:pipeline>
```

This pipeline example shows how the pipeline should look. You will use a simple combination of an XML file and an XSL stylesheet to generate an error message for the user if the file cannot be found on the server. Edit and save the XML document shown in Listing 4.44 to the Cocoon context directory, and name it filenotfound.xml.

Listing 4.44 The XML Document filenotfound.xml

```
<?xml version=”1.0 ”?>
<document>
    <text>The file is not available on the server.</text>
</document>
```

You should also save the stylesheet shown in Listing 4.45 to the Cocoon context directory and name ir filenotfound2html.xsl.

Listing 4.45 The XSL Stylesheet filenotfound2html.xsl

```
<?xml version=”1.0 ”?>

<xsl:stylesheet version="1.0">
              xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="document">
  <html>
    <body>
      <h1><xsl:value-of select="text"/></h1>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

If the file is not found, the stylesheet formats the error message as XHTML which is then returned to the calling application, such as the browser.

It might be a good idea to add a warning graphic to the error page when you return it to the user. The following example shows you how to use the SVG serializers to dynamically generate images.

SVG

The SVG serializers can be used to generate images from a textual description. You will now use this feature to extend the image gallery example started earlier in this chapter.

You will add a link to each thumbnail that is presented as an image. In addition, only four images will be displayed within a single document. A navigation feature with “previous” and “next” will also be added. Figure 4.11 shows what the end result will look like.

Figure 4.11. The enhanced gallery.

To start the example, let's generate some images by creating a pipeline for all requests that start with `graphics/` (see Listing 4.46). The name following this part is used as the text for the image. So requesting `graphics/previous` generates an image that has the text "previous" on it.

Listing 4.46 A Pipeline Fragment for SVG Graphics

```
<map:pipeline>
   <map:match pattern="graphics/*">
      <map:generate src="svg.xml"/>
      <map:transform src="addlabel.xsl">
         <map:parameter name="use-request-parameters" value="true"/>
         <map:parameter name="label" value="{1}"/>
      </map:transform>
      <map:serialize type="svg2jpeg"/>
   </map:match>
</map:pipeline>
```

The pipeline for an image reads an XML document containing directives for SVG (see Listing 4.47). These create an image consisting of the text and an ellipse around it. Because our focus is on Cocoon and not on SVG, we will not discuss the SVG commands in detail. To understand the example, it's sufficient to know that the `text` element prints text with the given attributes and that the `ellipse` command draws an ellipse.

Listing 4.47 The SVG Document

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

```
<svg xmlns:xlink="http://www.w3.org/1999/xlink" width="120" height="30">
  <ellipse cx="60" cy="15" rx="59" ry="15"/>
  <text style="font-family:arial; font-size:14px; font-weight:bold"
       fill="white"
       text-anchor="middle"
       x="60" y="18"><label/></text>
</svg>
```

Save this data as svg.xml in your Cocoon context directory. The main problem is to get the text's dynamic information into the read XML document. You do this using a trick. The read XML document contains a placeholder—the `label` element. This placeholder is detected by the stylesheet shown in Listing 4.48, addlabel.xsl. This stylesheet gets a sitemap parameter called `label` that in turn gets the value for the text from the document name—the part following `graphics/`.

Listing 4.48 The Stylesheet That Replaces the *label* Element

```
<?xml version="1.0"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
<xsl:param name="label"/>
 <xsl:template match="label">
    <xsl:value-of select="$label"/>
 </xsl:template>
  <xsl:template match="@*|node()">
   <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
   </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Save the stylesheet as addlabel.xsl to your Cocoon context directory. The stylesheet accesses the parameter by defining a global parameter called `label`. The template rule for the `label` element inserts the value of this parameter instead of the element.

You can test the image generation by invoking `http://localhost:8080/cocoon/graphics/Previous` or any other text after the last slash.

Now, let's add these images to your gallery (see Listing 4.49). You use the normal picture gallery as the base. As a further example (did anyone say "homework"?), you could incorporate these additions into the personalized picture gallery.

Because you want to display only four images in a document, the pipeline and the stylesheets need more logic. Which images should be displayed is driven by a request parameter named `page`. For example, if `page` has the value 1, the first four images are displayed, and so on.

Listing 4.49 A Pipeline Fragment for the Gallery

```
<map:pipeline>
    <map:match pattern="gallery">
       <map:generate src="gallery" type="imagedirectory"/>
       <map:transform src="gallery.xsl">
          <map:parameter name="use-request-parameters" value="true"/>
          <map:parameter name="page" value="1"/>
       </map:transform>
       <map:serialize type="html"/>
    </map:match>

    <!-- Gallery Images -->
    <map:match pattern="gallery/**">
       <map:read src="gallery/{1}"/>
    </map:match>
</map:pipeline>
```

The stylesheet gallery.xsl uses this parameter to filter the images. The xslt transformer has a default value for the page parameter defined in the pipeline (see Listing 4.50).

Listing 4.50 The Stylesheet for the Gallery

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

<xsl:param name="page"/>
<xsl:template match="/">
<html>
<body>
<center>
    <h1>The Gallery - Page <xsl:value-of select="$page"/></h1>
</center>
<xsl:variable name="count" select="count(dir:directory/dir:file)"/>
<table>
<tr>
    <xsl:for-each select="dir:directory/dir:file">
       <xsl:variable name="first" select="(($page - 1)*4)+1"/>
       <xsl:variable name="last" select="($page * 4)"/>
       <xsl:if test="position() >= $first and position() <= $last">
           <td width="130" align="middle">
              <a href="{/dir:directory/@name}/{@name}">
                 <img src="{/dir:directory/@name}/{@name}"
                     width="100"
                     height="100"/>
                 <br/><br/>
                 <img border="0" src="graphics/{substring-before(@name,
     '.')}"/>
              </a>
           </td>
       </xsl:if>
    </xsl:for-each>
</tr>
<!-- now the navigation -->
<tr>
```

```
    <td width="130"> </td>
    <td width="130">
       <xsl:choose>
        <xsl:when test="$page > 1">
            <a href="gallery?page={$page - 1}">
               <img border="0" src="graphics/Previous" width="130"
    height="30"/>
            </a>
        </xsl:when>
        <xsl:otherwise>
             
            </xsl:otherwise>
        </xsl:choose>
    </td>
    <td width="130">
       <xsl:choose>
          <xsl:when test="($page * 4) < $count">
             <a href="gallery?page={$page + 1}">
                <img border="0" src="graphics/Next" width="130"
    height="30"/>
             </a>
          </xsl:when>
          <xsl:otherwise>
              
          </xsl:otherwise>
       </xsl:choose>
   </td>
   <td width="130"> </td>
</tr>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

The stylesheet is now a little more complex, because it tests the page parameter and filters the images. All images are now displayed in a table. The first row contains the thumbnails, and the second row contains the navigation with the “previous” and “next” links.

As with the PDF example, we realize it might be too much to edit this stylesheet’s code yourself. Refer to the CD for this example, and copy the files to your local Cocoon installation. Because this book can only touch on the underlying formats such as XSL:FO, we refer you to the available literature on Internet sites for this informa-tion. You will find a list of web sites in Appendix C, “Links on the Web.”

Summary

This chapter started by presenting an overview of Cocoon and its concepts. We showed you how Cocoon can be run as a servlet and how you can configure it for maximum performance. We introduced the sitemap as the central configuration file and used examples to show how to build pipelines that can integrate various data sources and

publish to a variety of formats. Additional components allow the processing flow inside a pipeline to be controlled by parameters sent to the server or obtained elsewhere.

We also suggested that you perhaps try to change the given examples by adding additional components or by formatting the data as some other output. Now is a good time to reiterate that suggestion before we move on.

All in all, you have now completed the first part of the Cocoon overview from a user perspective. The next step is to show you how you can build a real-world application with this knowledge. We will start off slowly and then, as you learn more details about Cocoon, enhance the application until it becomes something that perhaps you could sell. Or give away.

Chapter 5. Cocoon News Portal: Entry Version

In this chapter, you will build the first version of a Cocoon-based news portal. The goal of this application is to show how the different concepts in Cocoon can be put to work to build an interesting solution. The application will be built over three chapters, with each new version of the portal adding functionality described in the preceding chapters. When complete, the portal will allow you to log on to view the news from themes you previously selected and view the information in various for-mats. You will also integrate a database for storing the user profiles.

In the first version of the portal, you will build the functionality that integrates different online news feeds into an installed version of Cocoon and show how the news data can be formatted into HTML, WML, and PDF for viewing in a browser or on a mobile phone.

Before we start describing the various steps, we want to mention a few points before we get started:

- You need a running version of Cocoon, as described in Chapter 3, “Getting Started with Cocoon.”
- In order to be able to access the news sources online, you need an Internet connection from the system on which Cocoon is running.
- If you do not have an online connection, you can use the sample data we have provided on the CD.

This chapter starts with a description of the type of data source you will integrate into your news portal. Then you will define and implement the layout of the published information. The last section describes how to combine the previous points in your application architecture.

Which Data Sources?

After defining the functionality of the first version, you will take a look at the data you want to integrate. To make the portal interesting and up to date, you will integrate “live” information. When a user logs on to the portal, the portal will access the news site and fetch the news the user has requested.

Many news sites already offer their information in an XML format. You will structure the application so that you can add additional news feeds yourself as well. To do this, we have opted for news feeds that return their data in a standard format, RSS. RSS stands for Resource Description Framework (RDF) Site Summary. It is designed as an easy-to-use format for syndication. The current version is 1.0, but at the time of writing the version used the most in available feeds is 0.91. If you are interested in the exact definition of this format, see Appendix C, "Links on the Web," where you will find the relevant URIs.

Before you look at some actual feeds, Listing 5.1 shows the format the data will be returned in, using the popular site `linuxtoday.com` as an example. Typing the URI http://linuxtoday.com/backend/biglt.rss into your browser causes an XML file to be returned that looks similar to Listing 5.1. Note that this is not the complete file or valid XML—just the first part of what you would see.

Listing 5.1 Sample RSS Data

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
                  "http://my.netscape.com/publish/formats/rss-0.91.dtd">
<rss version="0.91">
  <channel>
    <title>Linux Today</title>
    <link>http://linuxtoday.com</link>
    <language>en-us</language>
    <description>Linux Today News Service</description>
    <image>
      <title>Linux Today</title>
      <url>http://linuxtoday.com/pics/ltnet.png</url>
      <link>http://linuxtoday.com</link>
    </image>
    <item>
      <title>AbiWord Weekly News #75 by Jesper Skov</title>

     <link>http://linuxtoday.com/news_story.php3?ltsn=2002-01-14-004-20-
     NW-SW</link>
     <
     description>"The Bug hunt has started; lots of fixed Bugs in this week,
     and also
quite a few QAd/Closed. There's some feature improvements too (Word importer
     mainly), and
David started converting the documentation to HTML."</description>
  </item>
  <item>
    <title>GNOME Summary for 2002-01-05 - 2002-01-12</title>

     <link>http://linuxtoday.com/news_story.php3?ltsn=2002-01-14-003-20-
     NW-GN</link>
    <description>This week: Nautilus gets newsgroup binary viewing;
     Evolution 1.0.1
released, GUADEC 3 draws closer, GNOME interviews; GNOME 2.0 status report;
     more.</
description>
```

```
</item>
```

As you can see from this example, the RSS format is not too complex. Therefore, it is an easy format to write a stylesheet for, as you will see later in this chapter.

When you think about using the received information in your own portal, the relevant tags can be defined as follows:

- `<title>`: The name of this news channel

For each item of news, you have the following tags:

- `<item>`: A news item
- `<title>`: The title of a news item
- `<link>`: The link to the complete news item
- `<description>`: A short description of the news item

Now that you have looked at the format you will be receiving from the news sources, you need to design the stylesheets that will format this data into the output formats you need.

Designing the Layout

As soon as you have the data side of things sorted out, you must think about the layout you want to present. Using the sample data from the preceding section, you can design stylesheets that will format the news into three of the most popular formats: HTML, WML, and XSL:FO for serialization into PDF.

HTML

A news portal must be able to provide the news in an HTML format, so you need to design a stylesheet for this purpose. Listing 5.2 shows a sample stylesheet you can use in your application.

Listing 5.2 The Stylesheet for Formatting RSS into HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="*|/"><xsl:apply-templates/></xsl:template>
<xsl:template match="text()|@*"><xsl:value-of select="."/></xsl:template>

<xsl:template match="rss">
    <table><font face="Arial, Helvetica, sans-serif"><tr><th><xsl:value-of
select="channel/title"/></th></tr>
        <xsl:apply-templates select ="channel/item"/>
        </font>
    </table>
</xsl:template>
```

```
<xsl:template match="item">
   <xsl:if test="position() < 6">
      <tr bgcolor="#ffffff">
          <td><a target="_blank" href="{link}">
              <font size="-1" color="#333333"><b><xsl:value-of
    select="title"/></b></
font></a><br/>
              <font size="-2" color="#46627A"><xsl:value-of
    select="description"/></
font>
             </td>
      </tr>
      <tr bgcolor="#ffffff"><td bgcolor="#ffffff" height="5"></td></tr>
   </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

We have kept the stylesheet very simple. You are free to extend it as you wish. Remember, you can find all the files on the companion CD, so there is no need to type them in if you don't want to.

Looking at the stylesheet, you can see that the first template that does anything of interest triggers on the `rss` tag. Refer back to the example data and you will see this tag near the beginning of the file. As soon as the template is triggered on this tag, you select the title of the channel and display it above your table. Then you add a new row to the table for every item of news contained in the file.

If you look at the template that controls how the row is formatted, note how you check the current position in the data file. For this example, we show how to select just the first five items of news. Of course, you can change this number to a different value. You can also change the font colors and sizes, so you can experiment with the look and feel of the presentation after the first version of the portal is complete.

WML

Now that we have defined the HTML stylesheet, we will do the same for the WML format. This format lets you view the news on a mobile phone or using any other program that can present data formatted in WML. Again, the stylesheet is very simple, as shown in Listing 5.3.

Listing 5.3 The Stylesheet for Formatting RSS into WML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="*|/"><xsl:apply-templates/></xsl:template>
<xsl:template match="text()|@*"><xsl:value-of select="."/></xsl:template>

<xsl:template match="rss">
   <wml>
<card id="news" title="{channel/title}">
```

```
            <xsl:apply-templates select="channel/item"/>
        </card>
    </wml>
</xsl:template>

<xsl:template match="item">
    <xsl:if test="position() < 6">
        <p/>
        <small><b><xsl:value-of select="title"/></b>: <xsl:value-of
select="description"/></small><br/>
    </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

This simple stylesheet takes the XML data and formats it into the typical card layout of a WML page. In Listing 5.3, all the news items are formatted into one card. Of course, you could use a separate card for each item.

So far the stylesheets have been very simple and easy to follow. The next format we will present in the portal is PDF. To do this, we first need a stylesheet that formats the data into the XSL:FO format.

XSL:FO (PDF)

As we have mentioned, the XSL:FO format consists of many different tags and options. The main advantage of this format is that it allows the data to be serialized into printable formats such as PDF and PostScript.

Again, we have kept the stylesheet as simple as possible, as shown in Listing 5.4.

Listing 5.4 The Stylesheet for Formatting RSS into XSL:FO

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">
<xsl:template match="*|/"><xsl:apply-templates/></xsl:template>
<xsl:template match="text()|@*"><xsl:value-of select="."/></xsl:template>

<xsl:template match="rss">
   <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <fo:layout-master-set>
     <fo:simple-page-master master-name="page"
                page-height="29.7cm"
                page-width="21cm"
                margin-top="1cm"
                margin-bottom="2cm"
                margin-left="2.5cm"
                margin-right="2.5cm">
       <fo:region-before extent="3cm"/>
       <fo:region-body margin-top="3cm"/>
       <fo:region-after extent="1.5cm"/>
     </fo:simple-page-master>
```

```
    <fo:page-sequence-master master-name="all">
      <fo:repeatable-page-master-alternatives>
       <fo:conditional-page-master-reference master-name="page"
      page-position="first"/>
      </fo:repeatable-page-master-alternatives>
      </fo:page-sequence-master>
      </fo:layout-master-set>

     <fo:page-sequence master-name="all">
    <fo:static-content flow-name="xsl-region-after">
   <fo:block text-align="center"
           font-size="10pt"
        font-family="serif"
        line-height="14pt">page <fo:page-number/></fo:block>
   </fo:static-content>

   <fo:flow flow-name="xsl-region-body">
     <fo:block font-size="24pt" line-height="36pt"
    space-before.optimum="24pt"
text-align="center"><xsl:value-of select="channel/title"/></fo:block>
      <xsl:for-each select="channel/item">
          <fo:block font-size="10pt" line-height="14pt">
              <fo:inline font-variant="small-caps"
    font-weight="bold"><xsl:value-of
select="title"/></fo:inline>:<xsl:value-of select="description"/>
       </fo:block>
       <fo:block><fo:leader leader-length="100%" leader-pattern="rule"
rule-style="solid"
          rule-thickness="1px" color="black"/>
       </fo:block>
     </xsl:for-each>
   </fo:flow>
  </fo:page-sequence>
 </fo:root>
</xsl:template>
</xsl:stylesheet>
```

This stylesheet formats the XML data into a page layout that consists of the channel title and a block of information for each item. The title of each item is formatted as bold small capitals, and a horizontal rule is used to separate each piece of information.

Now that the layout and data are defined, you can build the pipelines in Cocoon that will fetch the news and format the data into the appropriate styles.

The Application Architecture

You might have noticed that so far we have not used any Cocoon specifics when looking at the data format your news provider sends and when dealing with the layout of your data. This shows how these individual parts of designing an application can be separated from each other.

We will now define the pipeline you need to access the sample news you use in this chapter. Remember that you are receiving RSS-formatted data from your provider. This means that the pipeline you need to access `linuxtoday.com` will work in the same way if you choose to access a different news site that provides an RSS feed.

The pipeline you need is really quite simple. You require a component that can access the news source over the Internet using HTTP and a component that formats the XML data into the output formats you defined with the stylesheets.

Listing 5.5 shows the pipeline.

Listing 5.5 The Pipeline for Accessing *linuxtoday.com*

```
<map:pipeline>
<map:match pattern="linuxtoday_*">
            <map:generate
    src="http://linuxtoday.com/backend/biglt.rss"/>
            <map:transform src="portal/styles/rss091_{1}.xsl"/>
            <map:select type="parameter">
                <map:parameter name="parameter-selector-test"
    value="{1}"/>
                <map:when test="html">
                <map:serialize type="html"/>
            </map:when>
                <map:when test="wml">
                <map:serialize type="wap"/>
            </map:when>
            <map:when test="pdf">
                <map:serialize type="fo2pdf"/>
            </map:when>
            <map:otherwise>
                <map:serialize type="xml"/>
            </map:otherwise>
        </map:select>
</map:match>
</map:pipeline>
```

Most of the pipeline should be familiar, but we will go through it step by step so that it is clear what you are doing. The first thing you should note is that we have chosen not to use the browser selector we talked about in the preceding chapter. The browser selector is a component that allows the automatic selection of, say, a stylesheet, depending on the user agent the browser sends to the server.

The problem with using the browser selector is that in order to see any data formatted in WML, you need a device or program that sends the correct user agent. In this first version of the portal, you will use a different means of selecting the different formats so that you are not limited to using a mobile phone to see WML. You will change back to the browser selector when you extend the portal.

In this version of the portal, you will use a key, encoded in the URI, to select the style you want to use. When you use a browser to request the information, you must append

the style (`html`, `wml`, or `pdf`). After the data has been fetched from the `linuxtoday.com` web site (remember that this is the same regardless of the format), the passed style is used to select the stylesheet.

Also, because the serialization step is different depending on the style you want to present, you use the parameter selector to select the correct serializer.

Putting It All Together

Now that you have all the different parts, all that is left to do is put them together.

The first step is to create a directory named portal directly below the cocoon directory. Then create a directory named styles below portal.This directory is where you will store the stylesheets. Either save them into this directory or copy them from the CD. Make sure they are named rss091_html.xsl, rss091_wml.xsl, and rss091_pdf.xsl, respectively.

Then open the sitemap and add the pipeline, as you have done in the other examples. The next step is to start Cocoon if it is not already running. You do this by starting the servlet engine.

Use a browser to access the news, adding the style you would like to receive to the URL when you type it in. For example, to see the news in HTML, use the URI `linuxtoday_html`. For WML, use `linuxtoday_wml`. If you want to view the WML format, make sure you use a browser that can display this format (such as Opera). To view the PDF version, you need to have the Adobe PDF viewer installed.

After you have made sure that the basics work correctly for one news feed, you can extend the portal to offer news from other sources in the same way.

Adding News Sources

As we have mentioned, adding news sources is easy, especially when the data returned is also in RSS format. It becomes even easier if you choose a news provider that offers a large selection of different news feeds in the same format. The news provider Moreover (www.moreover.com) offers more than 3,000 different news feeds and provides them in several formats, including RSS.

Selecting a news topic from Moreover is quite easy. A typical URI looks like this:

http://p.moreover.com/cgi-local/page?index_health+rss

Let’s now look at the pipeline that you can use to access the Moreover news feeds (see Listing 5.6).

Listing 5.6 The Pipeline for Accessing *moreover.com*

```
<map:match pattern="moreover_*_*">
<map:generate src="http://p.moreover.com/cgi-local/page?index_{1}+rss"/>
    <map:transform src="portal/styles/rss091_{2}.xsl"/>
      <map:select type="parameter">
             <map:parameter name="parameter-selector-test" value="{2}"/>
          <map:when test="html">
               <map:serialize type="html"/>
            </map:when>
          <map:when test="wml">
               <map:serialize type="wap"/>
            </map:when>
      <map:when test="pdf">
           <map:serialize type="fo2pdf"/>
      </map:when>
      <map:otherwise>
             <map:serialize type="xml"/>
      </map:otherwise>
    </map:select>
</map:match>
```

Looking at the pipeline, you can see that this is a very flexible setup. The first wildcard allows you to access a specific news feed, and the second wildcard lets you define the format you want to receive.

Entering the URI `http://myserver/cocoon/moreover_health_html` into the browser returns the health news in HTML format. This notation allows you to enter other names of news feeds directly into the browser without having to reconfigure Cocoon or write additional pipelines.

Here are the names of some additional feeds you can try:

- entertainmentgossip
- moviereviews
- musicbiz
- banking
- insurance
- onlinebanking

Visit the Moreover web site to find out which news feeds are available.

Add the pipeline to your sitemap below the previous entry for `linuxtoday.com`. Both pipelines should be enclosed by the same `map:pipeline` tag. The complete entry should look like Listing 5.7.

Listing 5.7 The Complete Sitemap Entry

```
<map:pipeline>
<map:match pattern="linuxtoday_*">
            <map:generate
     src="http://linuxtoday.com/backend/biglt.rss"/>
            <map:transform src="portal/styles/rss091_{1}.xsl"/>
```

```
            <map:select type="parameter">
            <map:parameter name="parameter-selector-test" value="{1}"/>
                <map:when test="html">
                    <map:serialize type="html"/>
            </map:when>
                <map:when test="wml">
                    <map:serialize type="wap"/>
            </map:when>
            <map:when test="pdf">
               <map:serialize type="fo2pdf"/>
            </map:when>
            <map:otherwise>
                  <map:serialize type="xml"/>
            </map:otherwise>
        </map:select>
      </map:match>
      <map:match pattern="moreover_*_*">
            <map:generate
    src="http://p.moreover.com/cgi-local/page?index_{1}+rss"/>
            <map:transform src="portal/styles/rss091_{2}.xsl"/>
            <map:select type="parameter">
            <map:parameter name="parameter-selector-test" value="{2}"/>
                <map:when test="html">
                    <map:serialize type="html"/>
            </map:when>
                <map:when test="wml">
                    <map:serialize type="wap"/>
            </map:when>
            <map:when test="pdf">
                <map:serialize type="fo2pdf"/>
            </map:when>
            <map:otherwise>
                    <map:serialize type="xml"/>
            </map:otherwise>
        </map:select>
      </map:match>
</map:pipeline>
```

Now that you have additional news feeds, your portal is nearly complete. At the moment, however, the user needs to know which link to enter into his browser in order to see a particular news feed. Let’s make this easier by adding an index page.

An Index Page

The last step in building this version is to add a simple index page that allows a user to select a particular news feed and format from a complete list you provide. You first need to define a format for your index page and its entries, as shown in Listing 5.8.

Listing 5.8 The XML Format for the Index Page

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<index>
    <title>Cocoon News Portal</title>
    <subtitle>Entry Version</subtitle>
```

```
    <entries>
        <entry>
            <name>LinuxToday</name>
            <htmllink>linuxtoday_html</htmllink>
            <wmllink>linuxtoday_wml</wmllink>
            <pdflink>linuxtoday_pdf</pdflink>
        </entry>
        <entry>
            <name>Moreover - Health</name>
            <htmllink>moreover_health_html</htmllink>
            <wmllink>moreover_health_wml</wmllink>
            <pdflink>moreover_health_pdf</pdflink>
        </entry>
    </entries>
</index>
```

This simple format allows additional feeds to be added easily by appending an `<entry>` for each news topic presented on the portal. In addition, the portal's title and subtitle are also defined in the XML format.

XML gurus might complain that having a separate tag for each format is not really necessary, but doing it this way makes it clearer—especially for those just starting out with XML and Cocoon.

To integrate the index into Cocoon, you need to create a directory named resources directly below the portal directory you created earlier. Save Listing 5.8 into a file named index.xml.

Listing 5.9 shows a stylesheet that formats the index file into an HTML table.

Listing 5.9 The Stylesheet to Format the Index File

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="*|/"><xsl:apply-templates/></xsl:template>
<xsl:template match="text()|@*"><xsl:value-of select="."/></xsl:template>

<xsl:template match="index">
    <html>
    <body>
    <center>
    <h1><xsl:value-of select="title"/></h1>
    <h2><xsl:value-of select="subtitle"/></h2>
    <hr/>
    <table>
    <xsl:apply-templates select="entries/entry"/>
    </table>
    </center>
    </body>
    </html>
</xsl:template>
```

```
<xsl:template match="entry">
    <tr>
    <td><b><xsl:value-of select="name"/></b></td>
    <td><a href="{htmllink}">html</a></td>
    <td><a href="{wmllink}">wml</a></td>
    <td><a href="{pdflink}">pdf</a></td>
</tr>
</xsl:template>
</xsl:stylesheet>
```

The stylesheet adds a new row to the table for each entry contained in the XML file. Therefore, there is no need to change the stylesheet if you add news feeds.

The only remaining point is the pipeline you need to add to the sitemap for the index page (see Listing 5.10).

Listing 5.10 The Pipeline for the Index Page

```
<map:match pattern="newsportal">
    <map:generate src="portal/resources/index.xml"/>
    <map:transform src="portal/styles/index.xsl"/>
    <map:serialize type="html"/>
</map:match>
```

Add this pipeline to the sitemap as before. Make sure it is enclosed in the `map:pipeline` tag. Start Cocoon (if it is not already running) and access the news portal URI. This should bring up the index page containing links to the two news feeds.

The Complete Entry Version

That completes the entry version of our news portal. Things might seem pretty simple so far. Even so, it pays to take a step back from what we have been doing and see how we arrived at the first portal.

Using only a few lines of sitemap configuration and a small number of stylesheets, you have built a fully functional news site. This site can access an extensible list of news feeds over the Internet and then put them in formats suitable for viewing in a browser or on a mobile phone or for printing as PDF. A user can select the news feed and the format from a complete list of all the feeds.

Now that we have reached this point, feel free to adapt the stylesheets to your own tastes or to add news feeds. When you extend the portal in Chapter 7, “Cocoon News Portal: Extended Version,” you will add database support and user management. We will also show how it is possible to combine several different feeds into one document. First, though, we will look at what Cocoon provides in the way of components and concepts that can help you do this.

Chapter 6. A User's Look at the Cocoon Architecture

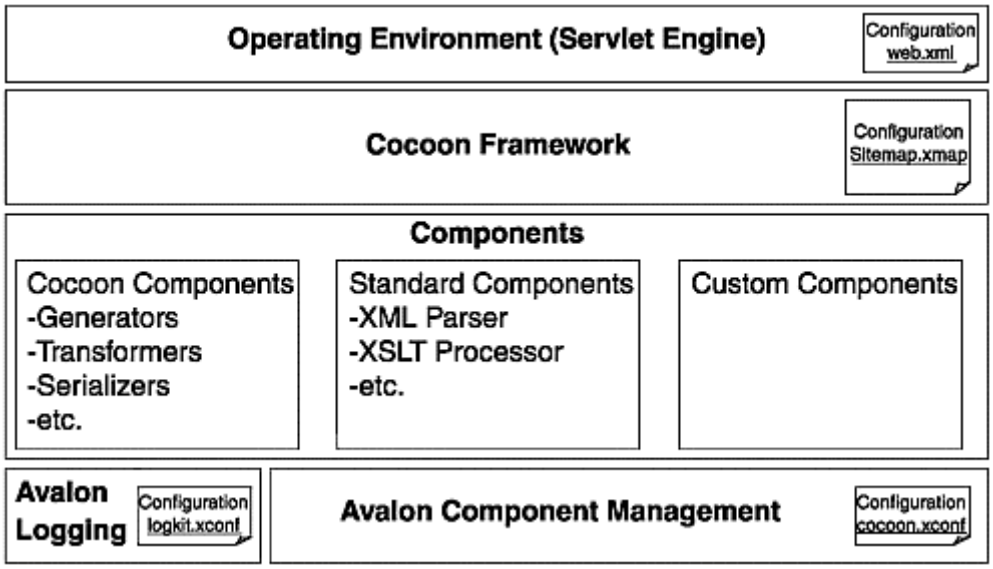
In Chapter 4, "Putting Cocoon to Work," you saw a simplified view of the Cocoon architecture. You built a first version of a news portal in Chapter 5, "Cocoon News Portal: Entry Version." Now that we have gone over the basics, it is time to fill in the missing pieces from a user perspective. This chapter 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.

Figure 6.1. The big picture of Cocoon.



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.

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"
.-->
<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.PriorityFilterTargetF
     actory"/>
    <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

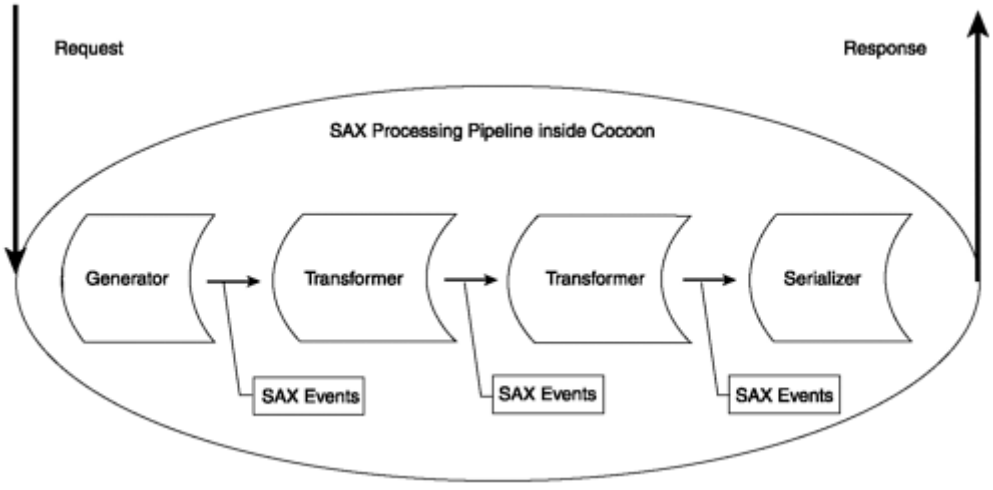
XML pipelines also work internally with the SAX model. Therefore, a generator sends SAX events to the following component in the pipeline. This component sends SAX events to the next one, and so on until the final serializer gets the final SAX events, serializes them, and creates the output document.

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.

Figure 6.2. SAX event handling.



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.

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.

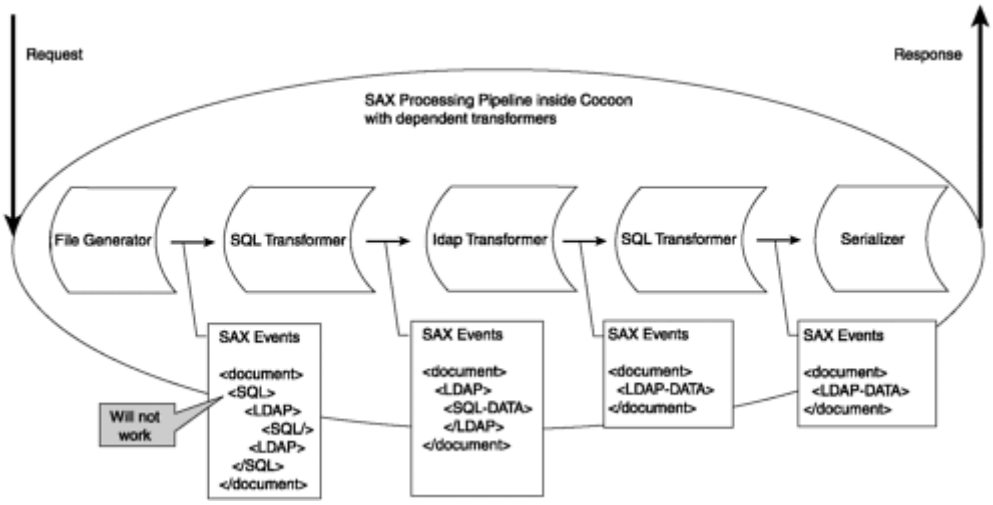
Of course, you need not stick to this simple rule. In some cases, the information delivered from one transformer cannot be used directly by another transformer. Then you should use intermediate stylesheet transformation, which converts the data of the first transformer to usable input for the second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

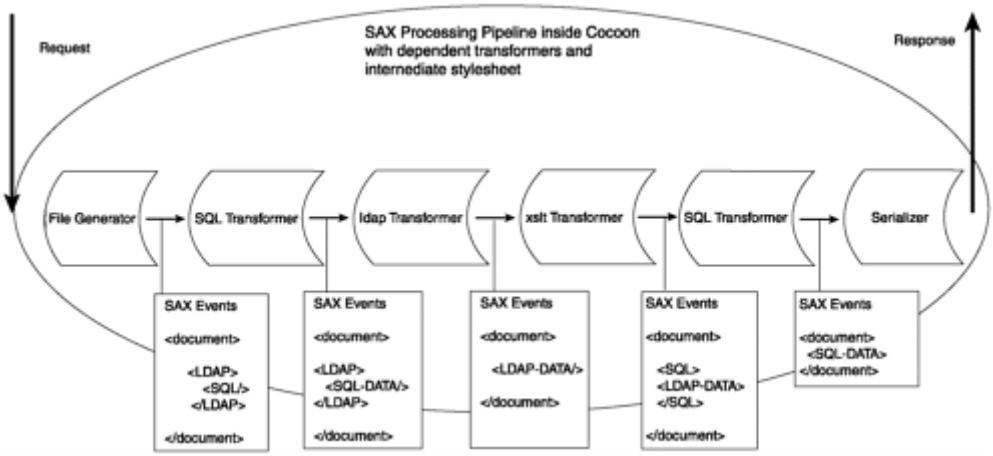
The simple approach shown in Figure 6.3 will not work. If you follow the rule we set up, you would build the structure of the commands as set out in the first block at the beginning of the chain in the figure—first the outer tags for the last sql transformer, and then the tags for the ldap transformer, 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.

Figure 6.3. Incorrect chaining of dependent transformers.



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

Figure 6.4. Using an intermediate stylesheet.



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.

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 for-merly 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`.

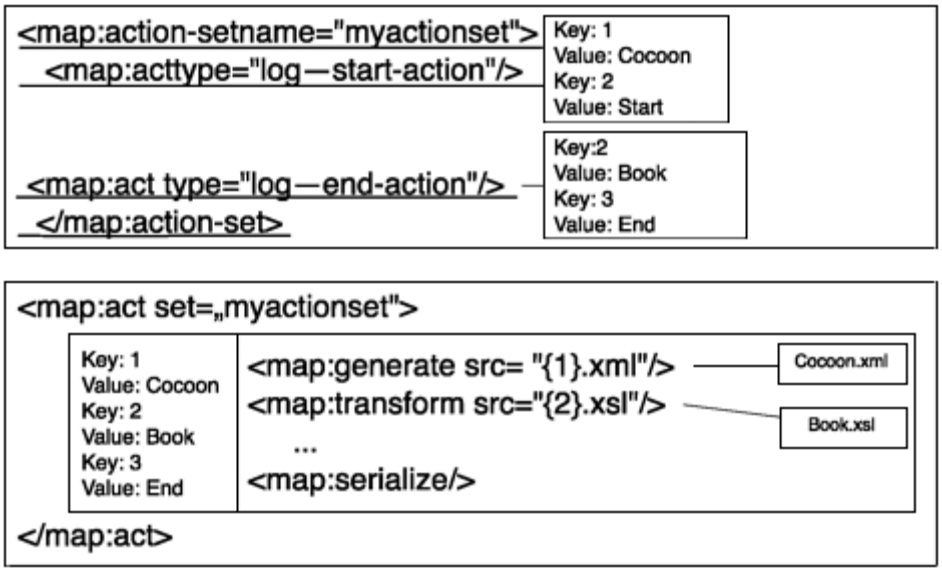
If you use an action-set, all actions of this set are called in the order they are defined. In addition, it is possible to selectively call an action inside an action-set. To do this, you can define each action in the action-set to have an attribute `action`. If the current request being processed by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

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.

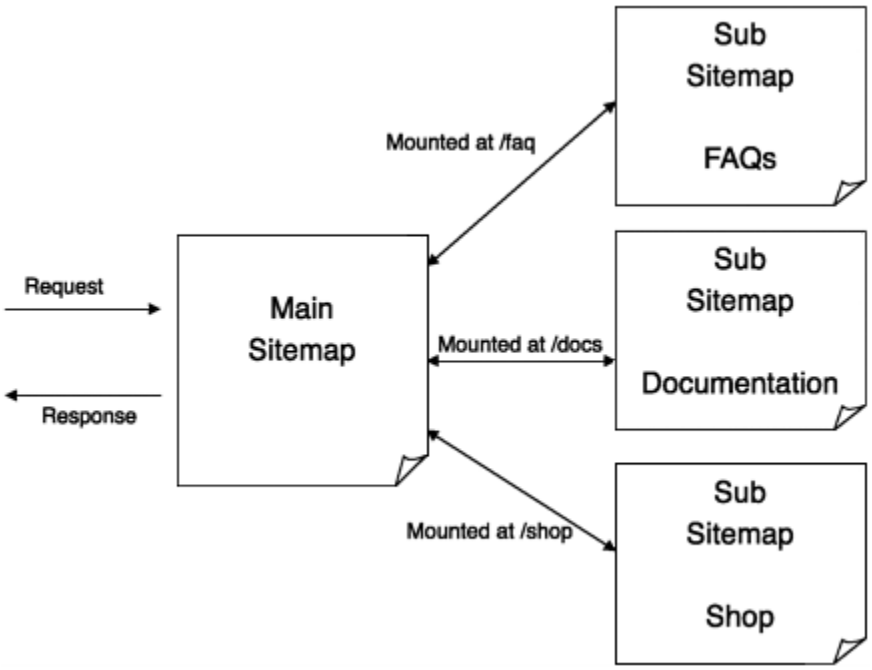
Later you will see practical examples and tips and a real-world example of content aggregation. This example—the Cocoon online documentation—also uses some other features not explained yet. One of them is the concept of subsitemaps.

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 readabil-ity 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.

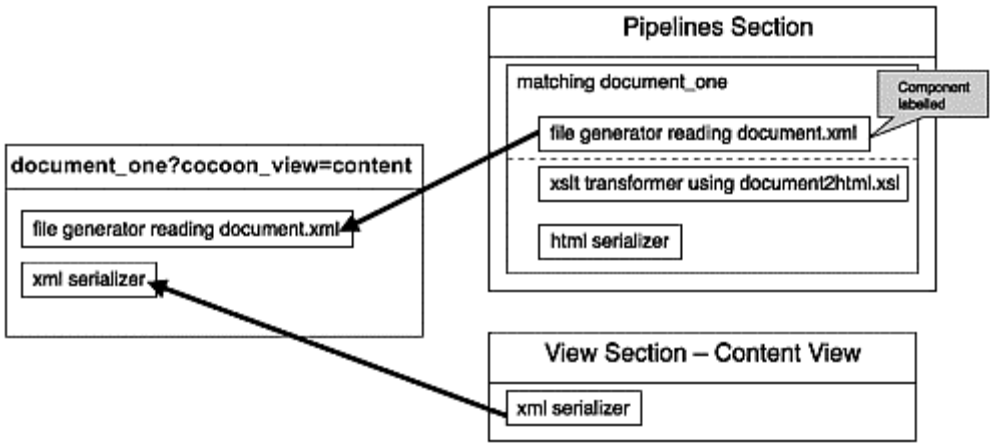
Listing 6.16 An Example of Labeled Views

```
<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.

Figure 6.7. A simple example of using views.

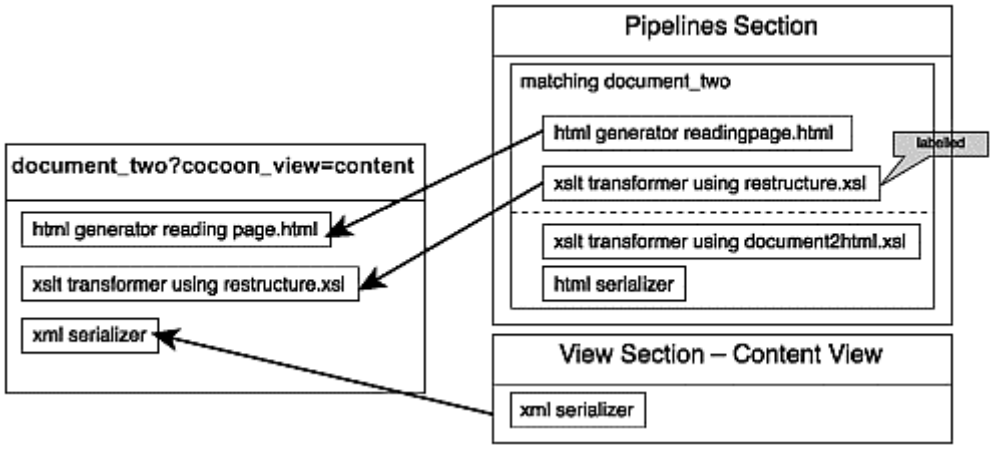


The file generator is labeled, so it is used. If a component is labeled, it is added to the pipeline for the view, and the usual pipeline processing is passed to the views section. All

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.

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.

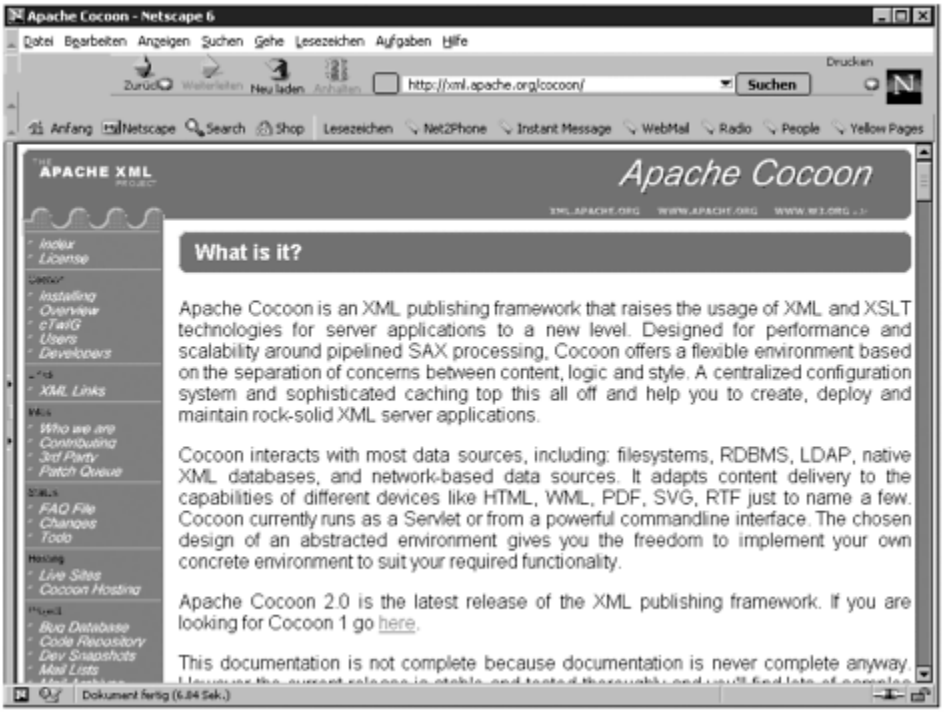
The Cocoon Documentation System

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.)

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.NonCachingEventPipelin
     e"/>
<stream-pipeline
     class="org.apache.cocoon.components.pipeline.NonCachingStreamPipeli
     ne"/>
```

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.

Chapter 7. Cocoon News Portal: Extended Version

Now that you have learned more about what Cocoon offers in the way of components and concepts for building applications, you will now put some of what you have learned to use by extending your news portal. You developed the first version of the portal in Chapter 5, "Cocoon News Portal: Entry Version." Now you will extend the solution by adding a database, where you will store portal users and their individual profiles. Most Internet applications require some form of database, so integrating one into Cocoon is one of the more common tasks that face any Cocoon application builder. The advantage of the Cocoon distribution is that a database is already included, so there is no need to set up any additional software.

As soon as a user has logged on to the portal, you need to let her edit the news she receives. Storing the user's profile in the database enables you to provide an HTML form in which the data can be edited. Because this is the extended version of the portal, you will also let Cocoon aggregate different news feeds into a single document and present the complete news page to the user.

Before you begin, you need to make sure you have Cocoon installed, as described in Chapter 3, "Getting Started with Cocoon." Although it is not necessary to have the entry version of the portal running, we recommend that you do this first to familiarize yourself with some of the basic concepts we will expand on in this chapter. You will find the first version of the news portal in Chapter 5.

As with all the examples in this book, we have provided all the files on the companion CD. Instead of editing the various information files yourself, you can copy them from the CD into your setup.

You will start by looking at the application's architecture and defining the various functions you want to present. Because the extended version of the portal is more complex than the first version you built, it makes sense to start with a concept of what you want to do. Then we will examine the components you need in order to implement the various functions.

While going through this example, please remember that we're deliberately keeping it as simple as possible. Don't expect to win any awards for the presentation. At all times you are encouraged to extend the example and add your own look and feel or cater to situations we have omitted. There are some further ideas for changing this version at the end of this chapter.

Designing the Portal

You will define your portal's architecture by looking at the functionality you need to provide and examining the flow through the individual functions. This will help you define the structure inside Cocoon.

Obviously, the basic functionality you need is the ability to authenticate the user. So you need a form where the user can enter the ID and password. That data will be sent to Cocoon and checked against a database. If the user is contained in the database and the password is entered correctly, the user receives a personalized portal page containing the news feeds she has subscribed to. You will also personalize the page by changing the background color. For the sake of simplicity, you will define that if the authentication is unsuccessful, the user will just get a blank welcome page with no news feeds.

From the portal page, the user has the option of editing the news feeds she has subscribed to. Your application will generate a document containing the list of news feeds, and the user can then select a feed to delete or add a new feed. All the changes are then sent back to Cocoon and are stored in the database so that they are available when the user logs in the next time.

Now that we have outlined the basic functionality, it is time to take a look at each function in more detail. By doing this, you should arrive at the Cocoon technologies and concepts you need in order to realize the complete solution.

Logging In to the Portal

Logging in to the portal consists of several different steps that each mean using different components. Here are the steps:

1. A request is made to the server to access the login form.

2. The user enters the data (ID and password), which is submitted to the server.

3. The database is accessed. The user is selected from the user table based on the entered information.

4. The selected information also contains the user's preference for the background color.

5. If the process is successful, the news feed table is accessed, and the configured news feeds for the user are selected.

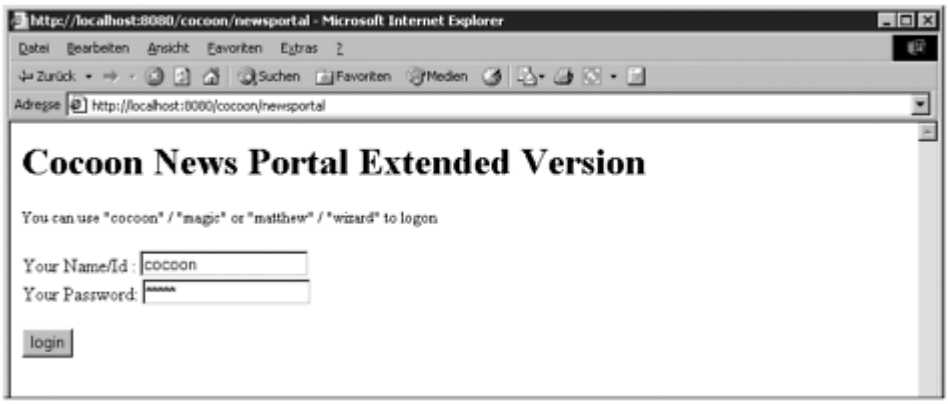
6. The news feeds are transformed into appropriate URIs to access the news server over the Internet.

7. The various news data is aggregated into a single XML file.

8. The XML file is transformed into the output format and is presented to the user.

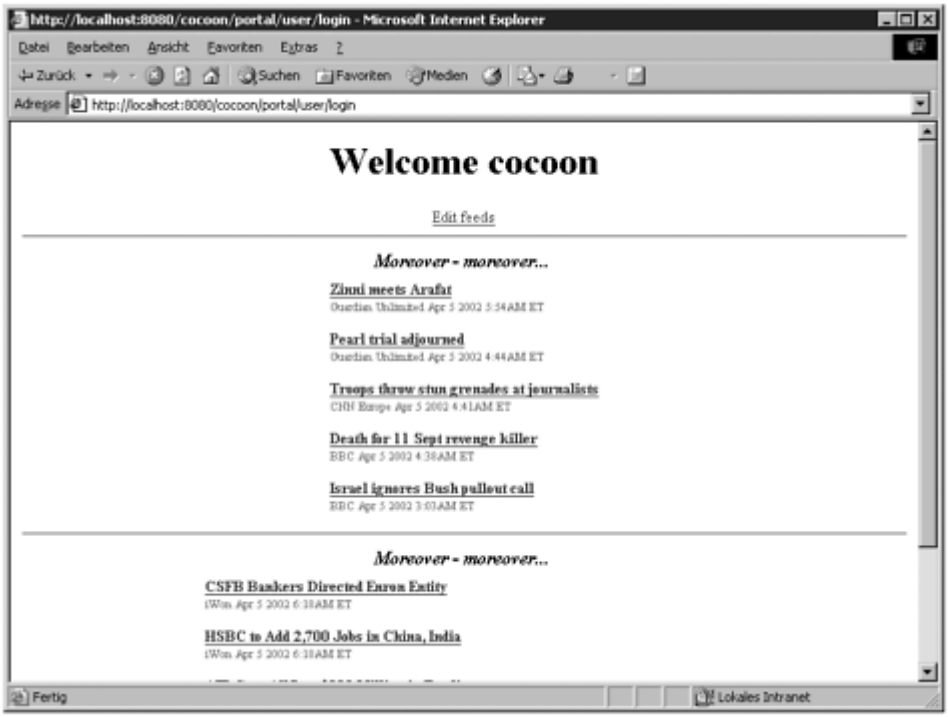
Figure 7.1 shows the login page that the user receives as a result of the first step.

Figure 7.1. The Login page.



After the last step has been completed, the user should see her selected news feeds, as shown in Figure 7.2.

Figure 7.2. The Portal page containing the selected news feeds.



Because you've specified that the user may edit the selected news feeds, we next need to take a look at this function.

Editing the News Feeds

After the user logs on, the news feeds are fetched and presented. On the same page, you will also provide a link to the edit function. When the user clicks that link, several steps need to be performed on the server:

1. A parameter containing the user ID is passed to the server with the request.

2. The called function takes the passed information and accesses the news feed table in the database.

3. The names of the selected news feeds are formatted into a form that the user can manipulate.

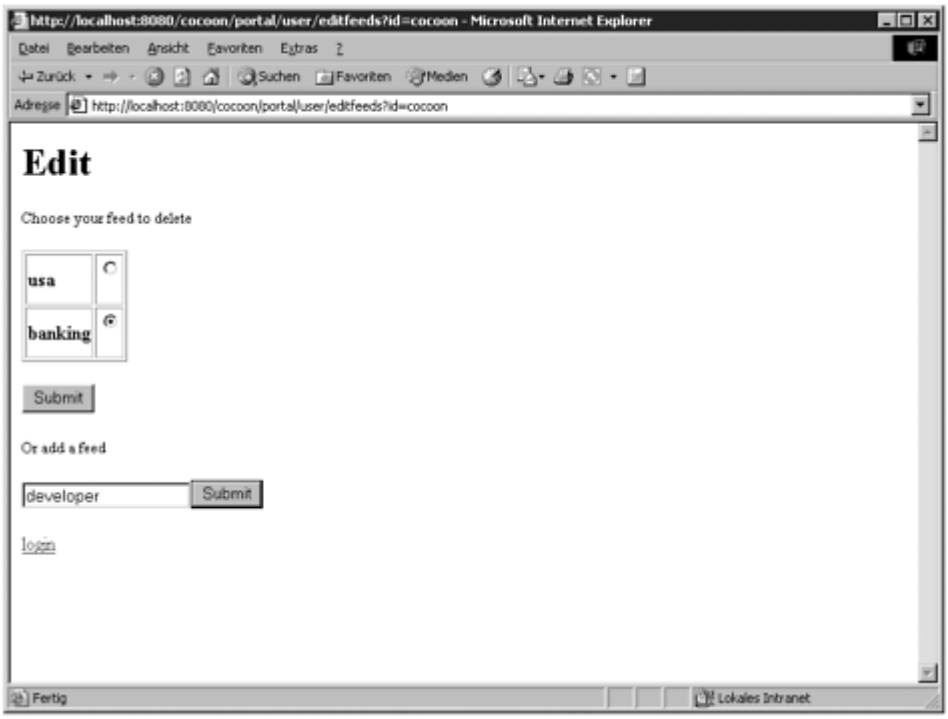
4. The user can delete certain feeds or add new ones by way of a simple edit box on the form.

5. The form data is submitted to a function on the server.

6. The server receives a parameter containing the requested action (add or delete) and the user ID.

The document that the user receives in order to edit her news feeds is shown in Figure 7.3.

Figure 7.3. The Edit page allows news feeds to be deleted or added.



After the data has been passed back to the server, the database tables must be edited to reflect the new information. Here are the steps in the editing process:

1. Depending on the requested action, the news feed table is updated to reflect the changes.

2. As a result of the changes, a new edit form is generated and is returned to the user.

3. From the edit form, the user can access the portal's login page.

As you will see when we explore the possibilities of building the Cocoon pipelines for these functions, you can combine these editing steps into a single pipeline. This will greatly simplify the changes you need to make to the sitemap.

The next step is to take a look at the different data sources your portal needs and to get started on actually configuring your application.

Integrating Data Sources into the Portal

The extended version of your portal has two data sources. You will use a database to store various information and an online news feed to provide the news items the user selects.

Storing Information in the Database

As installed, Cocoon comes with an integrated HSQL database that you can use to store your users and their profiles. Cocoon also comes with the necessary components needed to integrate other databases via Java Database Connectivity (JDBC).

The first step is to configure the database tables you need. You do this by using the script that HSQL provides and that is read and executed when the server is started. Because the HSQL database is integrated into Cocoon, it is started automatically when the servlet engine is started. Your portal needs two new tables in the database. In addition to creating those tables, you will add some initial data so that your database is not empty to begin with.

Adding Tables to HSQL

HSQL provides a file called cocoondb.script that can be found in the \WEB-INF\db directory of a Cocoon installation. You can edit this file with any text editor. You need to append the lines shown in Listing 7.1 to the end of the file.

Listing 7.1 Additional Entries for the cocoondb.script

```
CREATE TABLE PORTALUSER_TABLE(ID VARCHAR,PASSWORD VARCHAR,COLOR VARCHAR,
    UNIQUE(ID))
CREATE TABLE MOREOVER_TABLE(ID INTEGER,NAME VARCHAR,NEWSFEED
    VARCHAR,UNIQUE(ID))

INSERT INTO PORTALUSER_TABLE VALUES('cocoon','magic','white')
INSERT INTO PORTALUSER_TABLE VALUES('matthew','wizard','yellow')
INSERT INTO MOREOVER_TABLE VALUES(1,'matthew','banking')
INSERT INTO MOREOVER_TABLE VALUES(2,'cocoon','usa')
INSERT INTO MOREOVER_TABLE VALUES(3,'cocoon','banking')
```

As you probably can understand from these lines, you are creating two additional tables and inserting some sample data into them. Feel free to change the values if you would rather log in to the portal using your own name.

The user table contains an entry for each portal user. The entry consists of the ID, password, and the user's favorite color. Please note that this version of the portal only allows the color to be changed in this script, not via a form. But that is something we leave to you to extend as an exercise.

The Moreover table contains the configured feeds for each user. Each entry consists of a unique ID, username, and news feed name.

Be sure to save the edited file to its original location. That completes the HSQL configuration. The next time the servlet engine is started, this file will be executed and the new tables created. If you choose to edit this file later, you will notice that the order of the commands has changed. This is because HSQL dynamically updates the file when it is running. However, this has no effect on the portal.

Now that you have set up HSQL, you need to make Cocoon aware of your new tables and create a database connection you can use inside the portal.

Configuring a Connection in cocoon.xconf

Next you need to create a new database connection. You do this by adding a few lines to a file you already know: cocoon.xconf. You can open this file with your favorite XML editor (or a simple text editor if you prefer). Now, you need to find the <datasources> section of the file. This part of cocoon.xconf contains the configured database connections, and this is where you will add the new one. Add the lines shown in Listing 7.2 inside the <datasources> tags.

Listing 7.2 Adding a New Database Connection

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

Basically, all you have done is to use the default settings of the existing database connection and to give your new connection the name “portal.” This means that you can now use the new connection to access the database from, say, the sql transformer, as you will see later.

That is all there is to setting up the database for this version of the portal. It’s pretty simple so far. Of course, the data source you are really interested in is the news site. In this chapter you will use an available news feed you integrated previously: Moreover.com.

Your Portal News Provider: Moreover.com

As in the previous version of the portal, you will use news feeds from Moreover.com in this version of the portal. Moreover offers a large number of different feeds, so you can adapt the news feeds we have chosen to fit your particular interests.

Accessing the news feeds at Moreover is pretty easy, as you saw previously. Basically, you need to request a URI that contains an identifier for the news you want to see. A sample URI looks like this:
http://p.moreover.com/cgi-local/page?index_usa+rss. This link returns current

U.S. news headlines. The actual news is then returned in the XML format RSS. You can transform this into an output format such as HTML using a stylesheet and transformer.

Because the user might have several feeds configured, you will access Moreover more than once, combining the XML into a single stream and then formatting the end result into the completed portal page. You can reuse the stylesheet you used in Chapter 5 and just make a few changes to it to reflect the fact that you might have multiple feeds and not just one.

The portal lets the user add and delete configured feeds. Because the URI is the same for each feed (except for the name of the feed) you only need to store the names in the database. Therefore, the user only needs to enter a new name in the edit field you present. To give you some news feeds to play with, here are the names of some additional topics:

- entertainmentgeneral
- mutualfunds
- outdoorrecreation
- personalfinance
- entertainmenttvshows
- foodanddrink
- fitness
- naturalhealth
- parenting

A complete list of feeds can be found on the Moreover.com web site (http://www.moreover.com). If you choose to use news feeds from Moreover in your own portal, be sure to comply with the Moreover licensing terms.

In addition, remember that you are using RSS as the data format. This means that it should be no problem to change the site to a different one offering news feeds or to add additional sites. You did this in Chapter 3, so perhaps now is a good time to check out the additional information on RSS feeds in that chapter.

Building the Portal's Functionality

Up to now we have described the portal in very general terms. But in the end, you will need to use various Cocoon components and concepts to build the functionality. We will start with the login process and then look at the editing function.

To set up the portal's directory structure, you need to create a few new directories below the Cocoon root:

1. Create a portal directory below the cocoon directory.

2. Create two new directories, resources and styles, below the portal directory.

As you go through each function, you will edit sitemap.xmap and add the pipelines you need. So now is a good time to open the sitemap in an editor and find the best place to edit the new pipelines. We suggest that you find the opening `<map:pipeline>` tag and start the editing process by entering a new pair of `<map:pipeline>` tags to enclose the portal pipelines you will add as you continue.

Logging In to the Portal

As you saw when we discussed the login function earlier in this chapter, you first need a document that allows an ID and password to be entered. The entered data is then sent to the server to be checked. As a result of a successful authentication, the news feeds are aggregated, and the complete portal is presented to the user.

Therefore, the first step in the portal login process is the presentation of a form that allows the user to enter his or her ID and password. The pipeline you need is pretty simple, as shown in Listing 7.3.

Listing 7.3 The Pipeline to Generate the Login Form

```
<map:match pattern="newsportal">
    <map:generate src="portal/resources/start.xml"/>
    <map:transform src="portal/styles/start.xsl"/>
    <map:serialize type="html"/>
</map:match>
```

Add this pipeline to the new `<map:pipeline>` section you created in the preceding section. As you can see in this pipeline, the file generator reads an XML file, and a stylesheet formats the information into HTML, which is then returned to the user.

Listing 7.4 shows the XML data that must be edited into the start.xml file.

Listing 7.4 XML Data for the Entry Form

```
<?xml version="1.0"?>
<start>
    <pipeline>portal/user/login</pipeline>
    <idfield>id</idfield>
    <passfield>password</passfield>
</start>
```

The format is quite simple. The tags contained in the file represent parts of the form that you return to the user. The `<pipeline>` tag identifies the resource that is to be called when the user presses the Submit button. The names of the input fields are defined by `<idfield>` and `<passfield>`, respectively.

What you need next is the stylesheet that formats this data in the required output format—in this case, HTML (see Listing 7.5). Again, we have kept the stylesheet as simple as possible. You are welcome to add any bells and whistles you want.

Listing 7.5 The Stylesheet That Presents the Login Form

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="start">
<html>
<body>
<h1>Cocoon News Portal Extended Version</h1><p/>
<small>You can use "cocoon" / "magic" or "matthew" / "wizard" to logon</small>
     <form action="{pipeline}" method="post">
     Your Name/Id : <input type="text" name="{idfield}"/><br/>
     Your Password: <input type="password" name="{passfield}"/> <p/>
     <input type="submit" value="login"/>
     </form>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

The stylesheet formats the XML tags into an HTML form. If you changed the initial user data when configuring the database, be sure to change the comment in the stylesheet. Also make sure the stylesheet is saved to a file called start.xsl in the styles directory.

That completes the initial pipeline that returns the login form to the user. Now is perhaps a good time to start the servlet engine and call this pipeline to see if everything works as expected. If it is successful, the result of calling `http://localhost:8080/cocoon/newsportal` (or however your particular Cocoon installation is set up) should be an HTML form allowing the input of ID and password.

The login page was generated by reading a very simple XML file and applying a stylesheet that lays out the HTML. As always, and in particular with Cocoon, there are several other ways of doing this. As soon as the portal is complete, changing the login pipeline to read in an XHTML file using a Reader component would be a good exercise.

Authenticating the User

After the user has entered his ID and password and presses the Submit button, the data is sent to Cocoon to be processed by the pipeline portal/user/login. Looking back to where you defined the XML format, you see that the pipeline is configured there. This means that you can configure a different authentication pipeline to be used by changing the content of the `<pipeline>` tag.

For this sample portal, you'll wrap two separate application steps into one pipeline. Your authentication pipeline will authenticate the user against the database you previously set up and will obtain the names of the feeds and access the news online. This means that the result of a successful login will be the complete portal page containing the feeds.

Because the pipeline for this function contains quite a few entries, you will take it step by step and build the pipeline as you go along. Because of the way information is received in the form of request parameters (from the form input) and because this information is needed to generate the correct `select` statements for the database, you will use stylesheets to build the necessary statements. We mentioned at the beginning of this chapter that there are other ways of doing this. We refer you to the section "Closing the Portal" for additional ideas.

The start of the pipeline is very simple. All you need is an XML file that contains a tag to start things off. Remember that you need a generator as the start point of your pipeline, so you also need to provide at least the simplest of XML files for it to read.

Listing 7.6 shows the XML file login.xml.

Listing 7.6 login.xml

```
<?xml version="1.0"?>
<login/>
```

Save the file login.xml to the portal/resources directory. The next step is to write a stylesheet that reads the passed parameters from the form and creates the necessary database `select` statements. Listing 7.7 shows a stylesheet that does exactly that.

Listing 7.7 A Stylesheet to Generate the Authentication *select* Statements

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

<!-- Make the request parameters available -->
<xsl:param name="id"/>
<xsl:param name="password"/>

<xsl:template match="login">

  <user>
  <execute-query xmlns="http://apache.org/cocoon/SQL/2.0">
   <query>
       SELECT id,password,color from PORTALUSER_TABLE where id =
     '<xsl:value-of
select="$id"/>' and password = '<xsl:value-of select="$password"/>'
   </query>
  </execute-query>
  </user>

</xsl:template>

</xsl:stylesheet>
```

Save this file as buildlogin.xsl in the portal/styles directory.

There are a few things to note here. Notice that two parameters in the stylesheet are defined using the `<xsl:param>` tag. These are the same names you gave the input fields in your form. You will see in a moment how to configure the pipeline so that the parameters are passed to the stylesheet processing. As soon as the processing starts, the variables contain the values the user entered on the form.

The next thing to point out is the definition of the SQL namespace in the `<execute-query>` tag. This is necessary so that the sql transformer will be able to recognize the commands as they are sent via SAX events.

Inside the `select` statement are the parameters you defined earlier. This lets you build a complete statement, including the user ID and password. The `select` statement selects the ID, password, and color columns from the table `PORTALUSER_TABLE`.

Now is a good time to take a look at the pipeline you need in order to process the XML and XSL files you have developed up to now (see Listing 7.8).

Listing 7.8 A Pipeline Fragment

```
<map:match pattern="portal/user/login">
    <map:generate src="portal/resources/login.xml"/>
    <map:transform src="portal/styles/buildlogin.xsl">
        <map:parameter name="use-request-parameters" value="true"/>
    </map:transform>
    <map:transform type="sql">
        <map:parameter name="use-connection" value="portal"/>
    </map:transform>
```

Do not copy this fragment into the sitemap, because it is incomplete. We provide the complete pipeline at the end of this section. Looking at this fragment from top to bottom, you can see that the file generator first reads in the simple login.xml file. Next, the stylesheet that builds the database select is processed. Here you use the ability to pass request parameters to the processing step. Next in line is the sql transformer that actually performs the select against the database and returns the user information if successful. The configured database connection “portal” is provided as a parameter to the transformer.

That completes the authentication step. Notice that we have not taken into account that the authentication might fail—other than the portal page’s remaining empty when presented to the user. This is something you can add when the example is complete.

The next step in this pipeline is to create the `select` statements for the database table containing the configured feeds for the authenticated user. As in the previous step, you will use another stylesheet that does this for you (see Listing 7.9).

Listing 7.9 A Stylesheet to Generate the News Feeds *select* Statement

```
<?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="user">

  <user>
    <name>
        <xsl:value-of select="sql:rowset/sql:row/sql:id"/>
    </name>
      <background>
        <xsl:value-of select="sql:rowset/sql:row/sql:color"/>
      </background>
        <feeds>
          <execute-query xmlns="http://apache.org/cocoon/SQL/2.0">
              <query>SELECT newsfeed from MOREOVER_TABLE where name
    ='<xsl:value-of
select="sql:rowset/sql:row/sql:id"/>'</query>
          </execute-query>
        </feeds>
  </user>

</xsl:template>

</xsl:stylesheet>
```

Save this file as buildfeeds.xsl in the portal/styles directory.

This stylesheet is similar to the preceding one, so there is not much to explain here. You need to use the retrieved customer data to build the `select` statements. You also need to pass on the `<name>` and `<color>` information you received in the preceding step. One important thing to point out is the fact that you need to append the `sql:` namespace prefix when accessing data you received in a previous SQL select.

As soon as you have the `select` statement set up, you need to use the sql transformer to retrieve the data. Listing 7.10 shows the pipeline fragment that you need for these two steps.

Listing 7.10 A Pipeline Fragment

```
<map:transform src="portal/styles/buildfeeds.xsl"/>
<map:transform type="sql">
    <map:parameter name="use-connection" value="portal"/>
</map:transform>
```

The result of this step should be the configured news feeds for the user. Now you can use these feeds to build the information you need to access Moreover.com.

In previous chapters, we talked about how Cocoon allows different pipelines to be aggregated. In this chapter, you will do this differently—by using the `cinclude` transformer. You can read more about this transformer in Appendix A, “Cocoon

Components," and in the documentation provided on the CD. For now, let's look at the stylesheet that builds the statements for the transformer. It's shown in Listing 7.11.

Listing 7.11 The Stylesheet That Builds Statements for the *cinclude* Transformer

```
<?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"
     xmlns:cinclude="http://apache.org/cocoon/
include/1.0">

<xsl:template match="user">
      <user>
      <name>
         <xsl:value-of select="name"/>
      </name>
        <background>
          <xsl:value-of select="background"/>
        </background>

          <feeds>
            <xsl:apply-templates
     select="feeds/sql:rowset/sql:row/sql:newsfeed"/>
           </feeds>
  </user>

</xsl:template>

<xsl:template match="sql:newsfeed">
     <cinclude:include
     src="http://p.moreover.com/cgi-local/page?index_{.}+rss"
element="{.}"/>
</xsl:template>

</xsl:stylesheet>
```

Save this file as buildincludes.xsl in the portal/styles directory.

Again, this stylesheet is pretty simple, so we will concentrate on the new parts. Generating statements for the `cinclude` transformer is easy. All you need to do is create `<cinclude:include>` tags that have the appropriate URI as the `src` parameter. The transformer then accesses the URIs and returns the received XML to the pipeline. Notice how we have used the `element` attribute, with a value of the current feed, as a way of marking where the different feeds are to be inserted into the XML data.

Obviously you also need the transformer to do the work, so Listing 7.12 shows the pipeline snippet that covers these two steps.

Listing 7.12 A Pipeline Fragment

```
<map:transform src="portal/styles/buildincludes.xsl"/>
```

```
<map:transform type="cinclude"/>
```

You should now have a complete XML definition for the portal page. Included in the XML data are the user's ID, the chosen color, and the different feeds fetched from Moreover.com. This means that you are nearly finished! All you need now is a stylesheet that formats the XML into HTML.

But wait. Didn't we say that this was the extended version of the portal? Just adding a stylesheet to the pipeline would be far too simple. Let's use the browser selector to choose a different stylesheet, depending on whether the user is surfing the Web with Microsoft Internet Explorer or Netscape. First you need a stylesheet that will format the XML for you (see Listing 7.13).

Listing 7.13 The Stylesheet That Formats the Portal XML into HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="*|/"><xsl:apply-templates/></xsl:template>
<xsl:template match="text()|@*"><xsl:value-of select="."/></xsl:template>

<xsl:template match="user">
    <html>
        <body bgcolor="{background}">
            <center>
            <h1>Welcome <xsl:value-of select="name"/></h1><p/>
            <small><a href="editfeeds?id={name}">Edit feeds</a></small>
            <xsl:apply-templates select="feeds"/>
            </center>
        </body>
    </html>
</xsl:template>

<xsl:template match="rss">
     <hr/>
     <table><font face="Arial, Helvetica,
     sans-serif"><tr><th><em><xsl:value-of
      select="channel/title"/></em></th></tr>
         <xsl:apply-templates select ="channel/item"/>
         </font>
     </table>
</xsl:template>

<xsl:template match="item">
     <xsl:if test="position() < 6">
       <tr bgcolor="#ffffff">
           <td><a target="_blank" href="{link}">
               <font size="-1" color="#333333"><b><xsl:value-of
     select="title"/></b></
font></a><br/>
               <font size="-2" color="#46627A"><xsl:value-of
     select="description"/></
font>
```

```
            </td>
        </tr>
        <tr bgcolor="#ffffff"><td bgcolor="#ffffff" height="5"></td></tr>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Save this file as portal_html.xsl in the portal/styles directory.

Notice how you are reusing the stylesheet you wrote for the first portal version. The difference is that you are calling the template several times, depending on the number of feeds. You have also added a link that takes the user to a form where the configured feeds can be edited. You'll read more about this in a moment. The stylesheet also personalizes the color of the HTML, depending on the user's configured parameter.

As we mentioned, you will use the browser selector to select a different stylesheet for Netscape. Just save this stylesheet as portal_ns_html.xsl in the same styles directory. Now you can alter the stylesheet to perhaps use different colors for the fonts or something similar. We leave that exercise up to you.

Now that you have completed all the necessary stylesheets, it is time to present the complete pipeline (see Listing 7.14).

Listing 7.14 The Complete Pipeline

```
<map:pipeline>
  <map:match pattern="portal/user/login">
     <map:generate src="portal/resources/login.xml"/>
     <map:transform src="portal/styles/buildlogin.xsl">
     <map:parameter name="use-request-parameters" value="true"/>
     </map:transform>
        <map:transform type="sql">
              <map:parameter name="use-connection" value="portal"/>
     </map:transform>
     <map:transform src="portal/styles/buildfeeds.xsl"/>
        <map:transform type="sql">
              <map:parameter name="use-connection" value="portal"/>
     </map:transform>
     <map:transform src="portal/styles/buildincludes.xsl"/>
     <map:transform type="cinclude"/>
     <map:select type="browser">
         <map:when test="explorer">
             <map:transform src="portal/styles/portal_html.xsl"/>
         </map:when>
         <map:when test="mozilla5">
             <map:transform src="portal/styles/portal_ns_html.xsl"/>
         </map:when>
         <map:otherwise>
             <map:transform src="portal/styles/portal_html.xsl"/>
         </map:otherwise>
     </map:select>
     <map:serialize type="html"/>
```

```
  </map:match>
</map:pipeline>
```

You have added the browser selector and configured the stylesheets that format the HTML output. This completes the pipeline for portal authentication, with the end result being the personalized portal containing the news feeds. Perhaps now is a good time to try out what you have done so far. Just start Cocoon and then enter the URI `http://localhost:8080/cocoon/newsportal` into the browser. After logging in, you should see your personalized portal.

Now we will move on to the pipeline that lets you edit the configured feeds.

Editing the News Feeds

Your portal would quickly become boring if you allowed the user to see only the feeds you configured in the database setup script. Therefore, you need a way of allowing the user to edit the news topics. Now that you have had enough practice entering pipelines and stylesheets, let's dive right into things.

You will be changing information in the database. You can use the sql transformer to do this. However, Cocoon also provides some additional database components that help you. You first need to add a few lines to the sitemap so that you can use them.

The first thing to do is to add the following two lines to the `<map:actions>` section of the sitemap:

```
<map:action name="add-feed"
     src="org.apache.cocoon.acting.DatabaseAddAction"/>
<map:action name="del-feed"
     src="org.apache.cocoon.acting.DatabaseDeleteAction"/>
```

These entries configure `DatabaseAddAction` and `DatabaseDeleteAction` so that you can use them under the names `add-feed` and `del-feed`, respectively. Next you will create an action-set. You need to add the following lines to the `<map:action-sets>` section of the sitemap:

```
<map:action-set name="portal">
 <map:act type="add-feed" action="Add"/>
 <map:act type="del-feed" action="Delete"/>
</map:action-set>
```

This creates an action-set named "portal" that you can use in a pipeline. The different actions in the set are triggered by a parameter sent from the form the user has edited.

Because the action-set is triggered only if this parameter is sent from the browser, you can use just one pipeline both to present the information for the user to edit and to actually change the information in the database.

Let's look at the first part of the pipeline, shown in Listing 7.15.

Listing 7.15 The Start of the Edit Pipeline

```
<map:match pattern="portal/user/editfeeds">
        <map:act set="portal">
              <map:parameter name="descriptor"
    value="context://portal/resources/
dbfeeds.xml"/>
    </map:act>
    <map:generate src="portal/resources/editfeeds.xml"/>
```

First you define the match that is triggered when Cocoon receives the edit request. Looking back at the stylesheet that generates the portal, notice that there is a link to the edit resource:

```
<small><a href="editfeeds?id={name}">Edit feeds</a></small>
```

In addition to calling the correct pipeline, you also pass the user's ID. You will need it later.

The first component you use in the pipeline is the action-set you defined earlier. An action from this set is triggered when a parameter named `cocoon-action` with a value of either `Add` or `Delete` is received.

Obviously, the first time you call this pipeline (from the portal page), you do not have a parameter named `cocoon-action`, so you can ignore this entry for now. You will see what it does when you check out what happens on the server when the user wants to delete something.

The next entry is pretty simple. You use a file generator to read the file editfeeds.xml. As in the preceding section, this file just contains an initial tag to get everything going, as shown in Listing 7.16.

Listing 7.16 editfeeds.xml

```
<?xml version="1.0"?>
<editfeeds/>
```

As with all the XML files in this chapter, save this file to the portal/resources directory.

The next part of the pipeline strongly resembles part of the pipeline from the preceding section. Basically, you want to select the configured feeds from the database. To do this,

you first build the `select` statements using a stylesheet. Then you pass them to the sql transformer for processing. Listing 7.17 shows the pipeline snippet.

Listing 7.17 A Pipeline Fragment

```
<map:transform src="portal/styles/editfeeds.xsl">
    <map:parameter name="use-request-parameters" value="true"/>
</map:transform>
<map:transform type="sql">
    <map:parameter name="use-connection" value="portal"/>
</map:transform>
```

The stylesheet you use to build the `select` statements looks like Listing 7.18.

Listing 7.18 editfeeds.xsl

```
<?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:param name="id"/>

<xsl:template match="editfeeds">
  <feeds>
    <execute-query xmlns="http://apache.org/cocoon/SQL/2.0">
        <query>SELECT id,newsfeed from MOREOVER_TABLE where name
    ='<xsl:value-of
select="$id"/>'</query>
    </execute-query>
  </feeds>
</xsl:template>

</xsl:stylesheet>
```

If you have been following along, we don't really need to explain what's going on here, because you did the same thing in the second pipeline you edited. Just save this file as editfeeds.xsl to the portal/styles directory.

The result of the select are the feeds the user is currently registered for. Therefore, the final step in the chain is to build an HTML form that lets the user delete individual items or add new feeds. The pipeline fragment that does this is shown in Listing 7.19.

Listing 7.19 A Pipeline Fragment

```
<map:transform src="portal/styles/displayfeeds.xsl">
    <map:parameter name="use-request-parameters" value="true"/>
</map:transform>
<map:serialize type="html"/>
```

Something to notice here is that you pass the request parameters to the stylesheet processing. So let's now look at the stylesheet, shown in Listing 7.20.

Listing 7.20 displayfeeds.xsl

```
<?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:param name="id"/>

<xsl:template match="feeds">
<html>
<body>
<h1>Edit</h1><p/>
<small>Choose your feed to delete</small>
    <form action="editfeeds" method="post">
    <table border="1">
        <xsl:apply-templates select="sql:rowset/sql:row"/>
        <input type="hidden" name="id" value="{$id}"/>
        <input type="hidden" name="cocoon-action" value="Delete"/>
        </table>
        <p/>
        <input type="submit" value="Submit"/>
    </form>
    <p/>
    <small>Or add a feed</small>
    <form action="editfeeds" method="post">
        <input type="hidden" name="cocoon-action" value="Add"/>
        <input type="hidden" name="id" value="{$id}"/>
        <input type="hidden" name="name" value="{$id}"/>
        <input type="text" name="newsfeed" length="20"/>
        <input type="submit" value="Submit"/>
    </form>
    <p/>
    <a href="../../newsportal">login</a>
</body>
</html>

</xsl:template>

<xsl:template match="sql:row">
    <tr><td><b><xsl:value-of select="sql:newsfeed"/></b></td><td><input
    type="radio"
name="feedid" value="{sql:id}"/><br/></td></tr>
</xsl:template>

</xsl:stylesheet>
```

The stylesheet contains two different HTML forms. The first part lists the feeds and allows the user to delete individual ones. The second part lets the user add a new feed. Notice how the form action is set to `editfeeds`, which is the pipeline you are currently editing. Each form has some hidden fields—the `cocoon-action` we mentioned a moment ago and a field for the user `id`.

Next you need to look at the individual forms, because there are some differences worth pointing out. The first form, which allows the user to delete items, generates a radio button and entry for each news feed. Each entry in the database has a unique ID. This ID forms the value that is ultimately sent back to Cocoon if the user chooses to delete a feed. Then Cocoon knows exactly which news feed entry to delete from the database.

The second form contains an additional field called `name` that also holds the user's ID. The database table requires this parameter on an insert.

That completes our look at the stylesheet. So now you know that any data entered in the form is returned to the same pipeline. Next you need to return to your action-set and see how the actions can delete items or add new feeds. This is quite easy, really, because they use a special XML file called dbfeeds.xml. This is a parameter passed to the action-set if it is triggered by the `cocoon-action` parameter. Listing 7.21 is the file that needs to be saved to the resources directory.

Listing 7.21 dbfeeds.xml

```
<?xml version="1.0"?>
<portal>
  <connection>portal</connection>
  <table name="moreover_table">
    <keys>
      <key param="feedid" dbcol="id" type="int" mode="manual"/>
    </keys>
    <values>
      <value param="name" dbcol="name" type="string"/>
      <value param="newsfeed" dbcol="newsfeed" type="string"/>
    </values>
  </table>
</portal>
```

This file contains a definition of the database tables that is then used by the different actions to add items to and delete items from the news feed table. This hides the `select` statements from you, so this way is slightly easier than using the sql transformer. More information on the database actions can be found on the CD.

That completes the edit function. Listing 7.22 shows the complete pipeline that allows the user to change the news feeds.

Listing 7.22 The Complete Pipeline That Allows Feeds to Be Altered

```
<map:match pattern="portal/user/editfeeds">
        <map:act set="portal">
              <map:parameter name="descriptor"
     value="context://portal/resources/
dbfeeds.xml"/>
    </map:act>
    <map:generate src="portal/resources/editfeeds.xml"/>
    <map:transform src="portal/styles/editfeeds.xsl">
        <map:parameter name="use-request-parameters" value="true"/>
```

```
    </map:transform>
    <map:transform type="sql">
        <map:parameter name="use-connection" value="portal"/>
    </map:transform>
    <map:transform src="portal/styles/displayfeeds.xsl">
        <map:parameter name="use-request-parameters" value="true"/>
    </map:transform>
    <map:serialize type="html"/>
</map:match>
```

All the files we have described in this chapter are available on the CD, so there is no need to edit them. However, entering data into an XML editor helps you get used to the available tools and find out which one fits your requirements best.

Closing the Portal

This completes the extended version of your news portal. In only three pipelines, you have built a system that allows a user to be authenticated against a database and be presented with a news portal containing his or her configured information. In addition, the portal page is configured with the chosen color, and the list of news feeds can be altered by deleting or adding news feeds.

Instead of moving on to the next chapter, you could look at ways of extending this version and perhaps experiment with different components. As we have often mentioned, Cocoon provides various ways of writing applications. Which concepts you choose depends on the exact scenario the application has to cater to. Here are a few ideas to get you started:

- Let the user edit the background color.
- Add a way to add and delete users.
- Use the browser selector to generate WML for mobile phones instead of the HTML used here.
- In the authentication pipeline, use an XML file containing SQL statements instead of generating them using the stylesheet.
- Present a list of predefined feeds for the user to choose from when adding a new feed.
- Split the authentication and data-gathering steps into two separate pipelines.

Something else that we need to point out is that the completed portal is insecure, meaning that this version has no concept of session management. This means that someone can change a user's profile without having to log on first.

Having said that, however, these deficits will be corrected when you build the next portal version in Chapter 10, "Cocoon News Portal: Advanced Version." Before that, however, we will look at how you can develop additional components for Cocoon. We will also look at some of the more advanced features the platform has to offer.

Chapter 8. A Developer's Look at the Cocoon Architecture

So far this book has looked at the Cocoon architecture from a user perspective. Many different types of XML applications can be built using Cocoon as it is installed and with the components provided in the standard distribution. However, Cocoon obviously does not provide components for every data source available or for all the different types of applications you might want to build.

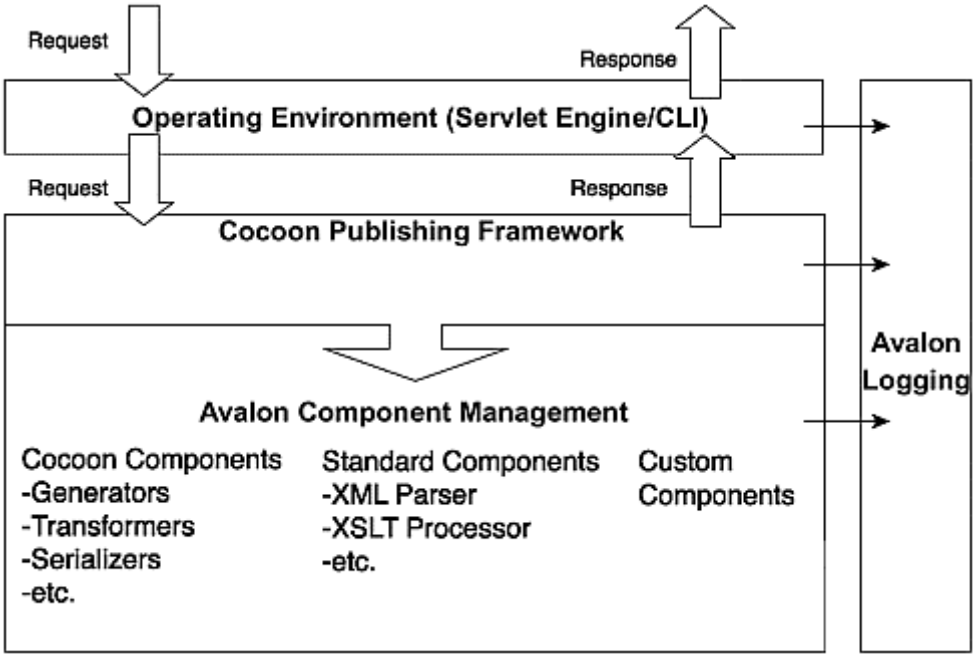
One of the great advantages of Cocoon is the fact that there is no need to wait for a new version to come out that provides a component for your particular data source. Because Cocoon is freely available and because it is a component-based architecture, new components can be written and integrated into the architecture easily. Cocoon can therefore be extended to meet new challenges as they arise.

This chapter looks at the inner workings of Cocoon from a developer's point of view. We will lay out the foundation for developers who want to develop their own components and who want to understand how Cocoon and the underlying architectures work.

If you are more interested in actually *writing* components, you might want to skip to the next chapter and then come back to this one when you need more information on why Cocoon components need to be written the way they do. That being said, we recommend that you read this chapter *first* and then move on to developing new components after you understand the basics. Don't worry if not all the points are clear when you reach the end of this chapter. The following chapter will help you understand how all the different pieces fit together. Appendix A, "Cocoon Components," and Appendix B, "Cocoon API Specifications," contain the information in a reference form. These are also good places to find additional information when developing components for Cocoon.

We will be repeating some key points from other chapters of this book, such as how Cocoon handles requests when running as a servlet. These are the points that you need to remember when examining the Cocoon architecture from a development perspective. To begin our tour and to see why there are quite a few things we need to detail in this chapter, look at the Cocoon architecture, shown in Figure 8.1.

Figure 8.1. The Cocoon big picture.



Cocoon uses the request-response cycle for document generation. Because Cocoon can be embedded in different environments, such as a servlet, or used from the command line, this surrounding environment is layered on top of the core processing framework. The requests passed to the servlet or from the command line are translated into requests that the publishing framework can understand. Cocoon processes the request and generates a response that is passed back to the surrounding environment. The environment transforms this response into the appropriate format. For example, the servlet environment writes to an HTTP stream back to the client browser, and the CLI saves the response to a file.

We will be examining how the environment works and how Cocoon and its components actually receive the various requests. We will be covering all the Cocoon core components and classes that can be used and that can be derived from to extend Cocoon with new components. Because Cocoon processing is based on the SAX model, it is important to also understand how SAX works and how it is implemented in Cocoon.

This chapter discusses various Java interfaces and classes, so a basic knowledge of Java will help you here. In order to keep the interface descriptions as compact as possible, we have often simplified the interfaces or classes by showing only the most important methods. Appendix B contains a more detailed API documentation.

As you can see from Figure 8.1, and perhaps remember from what you have seen already, many of the different components involved in request processing are embedded inside the Avalon component management architecture and use the Avalon logging facilities.

The Avalon Component Model

Cocoon consists of many different types of components, such as generators, transformers, and serializers. These components are managed by Avalon, a Java framework specialized

for this purpose. The Avalon project is divided into several subprojects. Only the subprojects LogKit, Excalibur, and the Avalon Framework are used in Cocoon.

The Avalon LogKit is a Java-based logging API. This logging functionality is used throughout all Avalon-based projects and inside Cocoon. The logging configuration is very flexible, as you will see later.

The Avalon Framework is the base of Avalon. It defines several concepts and interfaces for component development in Java. It covers the basics of defining, configuring, and managing software components and how to use them.

The Avalon Excalibur project is layered on top of the Avalon Framework. It implements common reusable components and offers some component management facilities so that you can fine-tune your installation.

Throughout Avalon and the projects that use it, you will encounter two design patterns: Inversion of Control (IoC) and Separation of Concerns (SoC). Both patterns are very common in object-oriented component development. Because the Avalon Framework was defined for exactly this reason, it supports these two patterns.

From a component's point of view, IoC means that the configuration and setup information is provided to the component instead of the component's having to ask for the information. So the software that creates and uses the component is responsible for giving all the necessary information to the component. The component is controlled and managed from the outside.

When developing an application, you have to deal with several areas or problem domains. Following the SoC paradigm, you need to identify these different domains and clearly separate them into different components. Each component is then responsible for one area or concern.

These two patterns are the basics of the Cocoon architecture. Cocoon's software design is highly object-oriented. Cocoon is made up of many components.

The following sections use the terms *application* and *component* to describe the various aspects of the Avalon framework. An application is software that *uses* a particular component, and a component is the piece of software that is *used*. To make things slightly more complicated, a component can also *use* other components. Up to this chapter, the application has been whatever solution you care to build with Cocoon, such as a portal or publishing application. In this chapter, the application is (in most cases) Cocoon itself.

If you write new components that are to be integrated into Cocoon, the following information on which interfaces your component needs to implement is important. Also, if the component you write requires access to other components, the information on how components are managed is important.

Before looking at the exact components contained in Cocoon, we need to first define what we mean by *component* when it comes to Avalon.

Defining Components

The ideal way to write object-oriented software is to assemble the new application by taking reusable pieces of software and combining them.

In Java, such a component is described using an interface and is implemented using a class. An instance of this class is the actual usable component. Why are components important? Well, one advantage is that they let you exchange implementations—perhaps replacing a given component with one you wrote yourself that might be better.

Consider the XML parser used in Cocoon. Cocoon comes with the Xerces parser as the default component for XML processing. But you could write your own XML parser and use it in Cocoon if you wanted to.

The two implementations of the parser, your own and Xerces, will differ, of course. Perhaps your parser is faster when processing small XML documents, but Xerces might use less memory. So, sooner or later you will face the question of which parser implementation you want to use. Because your own parser has some advantages, there will be use-cases for your application in which you will want to use your version. You therefore need a way to replace a given component with a different one, without having to rewrite the whole application.

This is a common problem that the Avalon framework helps you with. Rather than dealing with real components, such as the Xerces parser or your own, you define *roles* for the components. You first describe the XML parser role by writing a Java interface. This interface specifies all the functionality required from a parser. So when we talk about the role of a component, we are actually talking about a Java interface that defines the functionality of any component that wants to play that role.

The second thing you have to do is provide specific implementations for this role (or interface). You could adjust your own parser to implement this interface, and you could write a bridging implementation wrapping the Xerces parser.

Throughout your application, you only need to deal with the role of a compo-nent—not with the implementation. Because Avalon also provides mechanisms to manage roles and components, you can request components from it. Actually, you don’t ask Avalon directly. You use a framework provided by Avalon to fulfill this task.

What then remains is to configure which parser you want to use for the current application. To be more exact, you enumerate all the available roles your application needs and configure an implementation for each role. Although we are now introducing the configuration details for the first time, you have already implicitly seen how this works. Most of the configuration is contained in cocoon.xconf.

Avalon separates the definition of the available roles for an application and the configuration of the implementations for each role. The set of available roles is a fixed configuration for an application that is defined by the application developer. You will get a closer look at this in Chapter 9, "Developing Components for Cocoon," when you start writing your own components. Choosing an implementation for a particular role is the concern of the administrator who maintains the application. The role configuration in cocoon.xconf is

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

This is the main purpose of cocoon.xconf:You can decide which implementations for a given role you want to use. In the preceding excerpt, you specify that the `JaxpParser` class should be used as the implementation for the role with the name `parser`.

In addition, you can configure these implementations, as you will see later. However, the actual set of available roles is not defined in cocoon.xconf. We will provide more information on defining roles in Chapter 9, where you write your own components.

After looking at how components are managed by Avalon, you will see how the component life cycle is controlled and what mechanisms Avalon provides that allow a component to be reused.

The Component Manager

An Avalon component is defined by an interface description and an implementation. For example, a parser is described by a Java interface that specifies the services of this parsing component. In order to use the parser inside the application, you need an implementation that conforms to the interface and that can actually do something.

Throughout the application, whenever a component such as the parser is required, instead of the component's being directly instantiated, it is requested (or looked up) from a manager. More exactly, the application requests an implementation for a particular role from the manager. You know that you can configure which implementation is used for a role in cocoon.xconf. But how do you access components from your application?

When Cocoon is started—as a servlet or from the command line—some Avalon mechanisms are executed. The most important tasks they fulfill are to read cocoon.xconf and to make the components described in the configuration available to the application via a special object—the *component manager*.

The component manager is the central point for managing components. It conforms to the interface `org.apache.avalon.framework.component.ComponentManager`, as shown in Listing 8.1.

Listing 8.1 The Component Manager Interface

```
package org.apache.avalon.framework.component;

public interface ComponentManager
{
    Component lookup( String role ) throws ComponentException;

    boolean hasComponent( String role );

    void release( Component component );
}
```

So, if you need a component that plays a specific role, you simply look it up from the component manager using `lookup`. The information required is the name of the role, which is usually the name of the Java interface. One guideline for writing Avalon components is to define a constant named `ROLE` within the interface containing the name of the role. Because this is a generic interface where all kinds of components can be looked up, there has to be a common object type for all components. This is not `Object`, the base class of all objects in Java, but the Avalon `org.apache.Avalon.framework.components.Component` class. This is a marker interface that defines a class as an Avalon component.

Because Avalon is based on the IoC pattern, the component manager is responsible for setting up and configuring the component. After the component has been looked up, it can be used just as though it were directly instantiated. When the component is no longer needed, the component manager needs to be told that you have finished using the component. Calling the release method with the component as an argument does this. Let's have a look at a sample role—a Parser.

Listing 8.2 contains a simple interface for such a component. The role name is `org.apache.cocoon.components.parser.Parser`. This role defines a `parse` method that parses an `InputSource` object and sends SAX events to a `ContentHandler` and `LexicalHandler`. If you are unfamiliar with the SAX model, don't worry; we will introduce it later in this chapter. If you are more familiar with Cocoon and its components, you probably have noticed that this is not the real parser interface used in Cocoon. We have simplified it for this example. In order to understand component management, it is sufficient to know that special objects called `ContentHandler` and `LexicalHandler` can accept SAX events. An `InputSource` represents an XML document, so the parser knows what to parse.

Listing 8.2 The Parser Role

```
package org.apache.cocoon.components.parser.Parser;

public interface Parser
extends org.apache.Avalon.framework.component.Component
{
   String ROLE = "org.apache.cocoon.components.parser.Parser";

   void setContentHandler(ContentHandler contentHandler);
```

```
   void setLexicalHandler(LexicalHandler lexicalHandler);

   void parse(InputSource in) throws SAXException, IOException;
}
```

Listing 8.3 is a simple example of looking up a component in the parser role. The component is used to parse an XML document and is released afterwards. This is a common pattern for using Avalon components. It is very important to release a component after using it in order to allow other components to use the parser as well. To do this, you need to put the invocation of the release method into the final clause of your own code. We will provide more information on the shared usage of components later.

Listing 8.3 Using an Avalon Component

```
import org.apache.cocoon.components.parser.Parser;

public void parseDocument(InputSource document)
{
    // we have an instance variable called manager
    // containing the component manager
    Parser parser = (Parser) this.manager.lookup( Parser.ROLE );
    try {
        parser.setContentHandler( this );
        parser.parse( document );
    } catch ( Exception ignore ) {
    } finally {
        this.manager.release( parser );
    }
}
```

Until now we have had a one-to-one relationship between the role and the implementation. In an application such as Cocoon, only one implementation is needed for a role such as that of a parser. But there are other cases when several implementations for one role might be required at the same time. Without explicitly mentioning this, we showed you many examples in the preceding chapters. All sitemap components, such as actions, generators, and transformers, follow this pattern exactly.

For each component type, there exists exactly one role. So there is one role called Action, one role called Generator, and so on. Several components implement a role, such as `file generator` and `html generator`. If you were to use the component manager to look up a component that plays the Generator role, you would get only one component. So how are multiple components implementing a single role handled?

Avalon has the answer: a component selector. This is a component that conforms to the interface `org.apache.avalon.framework.component.ComponentSelector`. When you look up the component for the Generator role, you don't get a generator directly. The component manager returns a component selector that holds all the different components that implement the Generator role. In a way, this is similar to receiving a list of components instead of a single component.

The component selector has methods similar to those of the component manager: a `lookup` method, a `release` method, and a `hasComponent` method. So after the selector is returned from the component manager, you can look up a specific generator using the selector. After the generator has been used, it can be released using the selector. At some point in the application, the selector must also be released using the component manager. This is shown in Listing 8.4.

Listing 8.4 Using an Avalon Component Selector

```
import org.apache.cocoon.generation.Generator;

public void getGenerator(String type)
{
    // we have an instance variable called manager
    // containing the component manager
    ComponentSelector selector =
        (ComponentSelector) this.manager.lookup(Generator.ROLE +
     "Selector");
    try {
        Generator generator = (Generator) selector.lookup( type );
        try {
            // use the generator here
        } catch ( Exception ignore ) {
        } finally {
            this.manager.release( generator );
        }
    } finally {
        this.manager.release( selector );
    }
}
```

In the listing, you first get a selector for all generators. This is usually done by looking up a component from the component manager using the role name ending in `Selector`. Using this selector, you can look up a generator using its name, such as `file` or `html`.

Now you know everything you need to about requesting components from the component manager. According to the IoC pattern, a component is managed from the outside, so the component manager sets everything required by the component. Now let's look at how this works, starting with a component's life cycle.

A Component's Life Cycle

An Avalon component has a life cycle controlled by the component manager. At a certain time, a new instance is created and configured by the manager. After it has been used by the application, the manager destroys it.

So the first question that needs to be answered is when is a component created? For those familiar with Avalon, we won't discuss pooling and reusing instances just yet. This information is provided later in this chapter. For the moment, we will show the simple

process in which a new instance is created every time a component for a role is looked up from the component manager.

By using the configuration, the component manager determines the role's implementation and instantiates a new object of this class by calling the `newInstance()` method on the class object.

What follows is the component's configuration phase. For this purpose, Avalon offers some interfaces that a component can inherit from in order to receive the required information. We will describe these interfaces in the order in which the component manager tests them.

The *Contextualizable* Interface

The first interface tested by the component manager is the `Contextualizable` interface. It offers one method called `contextualize`. This method is called on the component with a `Context` object as the argument. Listing 8.5 provides a brief overview of the methods of these two classes.

Listing 8.5 The *Context* and *Contextualizable* Interfaces

```
package org.apache.avalon.framework.context;

public interface Contextualizable
{
    void contextualize( Context context ) throws ContextException;
}

public interface Context
{
    Object get( Object key ) throws ContextException;
}
```

The context is a store for information that the application provides to the component. In the case of Cocoon, the context contains some system information such as the name of the current temporary directory. Because the information is stored using key-value pairs, there must be a well-defined contract between the application providing the context and the components that need the information contained in the context.

The *Composable* Interface

The most important interface is the `Composable` interface. If a component implements this interface, it receives the current component manager via the `compose` method. Remember that the component manager is required to look up any Avalon component. So if you write a component that needs access to other components, you *must* implement the `Composable` interface. See Listing 8.6.

Listing 8.6 The *Composable* Interface

```
package org.apache.avalon.framework.component;

public interface Composable
{
    void compose( ComponentManager componentManager )
        throws ComponentException;
}
```

The implementation of the `compose` method is rather simple. The component usually stores the manager in an instance variable until it is needed.

The *Configurable* Interface

Besides the `Composable` interface, there are two other important interfaces: `Configurable` and `Parameterizable`. A component is allowed to implement only one of them, because they have the same purpose: to give the component its configuration. Whereas the `Parameterizable` interface allows the configuration via key-value pairs, the `Configurable` interface allows nested XML fragments. Let's look at a sample component configuration from cocoon.xconf:

```
<xslt-processor
     class="org.apache.cocoon.components.xslt.XSLTProcessorImpl">
   <parameter name="use-store" value="true"/>
   <parameter name="incremental-processing" value="false"/>
</xslt-processor>

<parser class="mypackage.parser.Parser">
  <settings>
    <use-store>true</use-store>
  </settings>
</parser>
```

The first component, the xslt-processor, has a configuration that consists of two key-value pairs. The parser has a nested configuration. Whereas the first example is a use-case for the `Parameterizable` interface, the second one requires the `Configurable` interface shown in Listing 8.7. Using the `configure` method, the component gets a `Configuration` object containing the whole configuration of the parser component.

Listing 8.7 The *Configurable* and *Configuration* Interfaces

```
package org.apache.avalon.framework.configuration;

public interface Configurable
{

    void configure( Configuration configuration )
        throws ConfigurationException;
}

public interface Configuration
```

```
{
    Configuration getChild( String child );

    Configuration[] getChildren();
    Configuration[] getChildren( String name );
    String[] getAttributeNames();
    String getAttribute( String paramName ) throws ConfigurationException;
    String getValue() throws ConfigurationException;
}
```

The `Configuration` object wraps the configuration of one XML element. So the object that the component gets via the `configure` method points to the parser element. It is now possible to get this element's attributes or its child elements/`Configuration` objects. For example, a call to `getChild("settings")` returns a new `Configuration` object. You can then ask this object via `getChild("use-store")` for a `Configuration` object pointing to the `use-store` element. You can ask this object for its value using the `getValue()` method.

There are more methods than we have listed in Listing 8.7. You can request the value of a `Configuration` object or its attributes in different type representations, such as a Boolean or float. Appendix B contains the whole Java API. For simple key-value configurations, it is easier to use the `Parameterizable` interface.

The *Parameterizable* Interface

The `Parameterizable` interface should be used whenever the configuration is flat, which means that the configuration is built from key-value pairs. The `parameterize` method is invoked with the `Parameters` object. Both classes are shown in Listing 8.8.

Listing 8.8 The *Parameterizable* Interface and the *Parameters* Class

```
package org.apache.avalon.framework.parameters;

public interface Parameterizable
{
    void parameterize( Parameters parameters )
       throws ParameterException;
}

public class Parameters
{
    public String[] getNames();

    public String getParameter( final String name, final String
    defaultValue );

}
```

The Parameters object holds the configuration as key-value pairs. If you look at the example of the Configurable description shown a moment ago, the xslt-processor gets two parameters as a configuration.

If the xslt-processor is Parameterizable, the two values can be requested via the getParameter() method. The first argument is the name of the parameter (use-store or incremental-processing), and the second parameter is a default value that is used if the parameter is not set in the configuration. Like the Configuration object, the Parameters object has more methods than we have listed here. They too are explained in Appendix B.

The *Initializable* Interface

Some components might need an initialization phase after they are configured properly but before they can be used. This can be specified using the Initializable interface, as shown in Listing 8.9.

Listing 8.9 The *Initializable* Interface

```
package org.apache.avalon.framework.activity;

public interface Initializable
{
    void initialize()
        throws Exception;
}
```

The component manager calls the initialize method. The component can then perform all the necessary steps, such as allocating other resources and looking up other components.

The Initializable interface is the last one the component manager tests on a component before returning it to the calling application. Now the component can be used. When it is not needed anymore, it should be released using the component manager.

During this release phase, again the component manager tests the component for one interface, Disposable, to help the component perform cleanup operations.

The *Disposable* Interface

The Disposable interface marks the component as wanting to deallocate or release resources before it can be destroyed. For example, if the original component looked up another component in the initialization phase, it needs to be released at some point. The dispose method of the Disposable interface, shown in Listing 8.10, is exactly the right place to do this.

Listing 8.10 The *Disposable* Interface

```
package org.apache.avalon.framework.activity;

public interface Disposable
{
    void dispose();
}
```

This interface finishes our tour through a component's life cycle. The IoC pattern in combination with the different interfaces a component can conform to results in a very powerful mechanism for managing and configuring components.

This completes our discussion of the basic interfaces that a component can implement. These interfaces allow the Avalon component manager to control and configure the component from the outside (IoC). Next, we will look at what means Avalon provides to make component instantiation and garbage collection as quick as possible.

Pooling Components

We have been talking about the basics of the Avalon framework, especially the management of components. Each time you look up a component, a new instance is created. This instance is destroyed when you release the component using the manager.

When you write components in Java, at least two significant areas can decrease your Web application's performance—object creation and garbage collection. A compo-nent-based architecture such as Cocoon has many lookups and releases. A Web application's performance would be drastically affected if each lookup of a component resulted in the instantiation of a new object and each disposal resulted in the object's being garbage-collected.

Avalon offers ways of writing more-sophisticated components with respect to object creation. The mechanisms we have described so far are suitable for components that can be used only once, and where it is necessary to create new instances each time the component is requested. This is standard component manager behavior. You can explicitly tell the component manager to manage the components in this way by making your components conform to the `SingleThreaded` interface, shown in Listing 8.11. As you can see, `SingleThreaded` is a *marker interface*, which means that it does not have any methods associated with it.

Listing 8.11 The *SingleThreaded* and *ThreadSafe* Interfaces

```
package org.apache.avalon.framework.thread;

public interface ThreadSafe
{
}


package org.apache.avalon.framework.thread;
```

```
public interface SingleThreaded
{
}
```

The opposite of this behavior is when a component is instantiated only once during the first lookup. It is never destroyed when it is released. Each time the corresponding role is looked up again, this exact instance is returned again. This minimizes object creation and garbage collection. This behavior can be achieved for a component by implementing the `ThreadSafe` marker interface.

Although you actually implement the singleton design pattern by conforming to the `ThreadSafe` interface, the intention behind the definition of `ThreadSafe` and `SingleThreaded` is slightly different: As the names imply, it has to do with multithread-ing. The singleton pattern describes a component type that has exactly one instance in the whole application. So a lookup of such a component always delivers exactly the same object.

As a servlet, Cocoon runs in a multithreaded environment. This means that several tasks are running concurrently. It is possible that they execute the same commands. For example, if two requests are processed simultaneously, both might need the file generator. So both look up the same Avalon component and act with the generator at the same time. If both tasks look up the same instance, this instance needs to be thread-safe. So this is the reason for the name of the interface `ThreadSafe`.

In contrast, a single-threaded component is not thread-safe. This means that if two threads need a component for the same role, they must receive different instances.

However, there is a compromise between the fastest solution's being thread-safe and the worst one's being single-threaded. You can create single-threaded components that are recycled. This means that when two threads need a component for the same role at the same time, they get a different instance. But after these instances are released, they are not destroyed, but pooled. The next time a component for this role is looked up, these instances are retrieved from the pool and are served by the component manager.

This pooling of components avoids not only the heavy costs of extra object creation and garbage collection but also the extra costs of component initialization and configuration. To allow the component manager to pool components, the component must conform to the `Poolable` marker interface, shown in Listing 8.12.

Listing 8.12 The *Poolable* and *Recyclable* Interfaces

```
package org.apache.avalon.excalibur.pool;

public interface Poolable
{
}

public interface Recyclable
```

```
    extends Poolable
{
    void recycle();
}
```

If a component conforms to this marker protocol, the life cycle interfaces for initialization (`Loggable`, `Contextualizable`, `Composable`, `Configurable`, and `Parameterizable`) are evaluated only when the component is instantiated. For each subsequent lookup, this component is already properly initialized and configured, so there is no need to do this again.

If the poolable component allocates resources during its usage, it must deallocate them after it is used. Because the component is never destroyed, the `Disposable` interface cannot be used for this purpose. But the component can conform to the `Recyclable` interface, also shown in Listing 8.12. When the component is released, the component manager calls the `recycle()` method, and the component can clean up and deallocate all resources.

Because it is very difficult to write thread-safe components, the poolable and recyclable components are the ones used most often. It is always the best solution if you can make your component thread-safe. If that is not possible, you should at least make it poolable or recyclable. Only if it is not possible to reuse your component should you declare it single-threaded. However, if none of the life cycle interfaces are used with a component, Avalon automatically treats it as a single-threaded component. This can lead to many unnecessary object creations and garbage collections. However, because we will show you how to write components that *do* implement the correct interfaces, this will not be a common problem.

Now you have the basics of Avalon components. One common need when writing components or applications is debugging. A helpful tool for debugging is using log messages throughout the components. So we will turn to another area of the Avalon framework used in Cocoon: logging.

Logging with the LogKit

The Avalon project consists of several subprojects. One of them is the Avalon LogKit. It provides an easy-to-use and powerful logging framework that can be configured and extended to meet nearly every need. We showed you some of the configuration possibilities in Chapter 6, “A User’s Look at the Cocoon Architecture.”You might want to have a quick look at that section, “LogKit Configuration,” before we continue. There’s no hurry. We’ll still be here when you get back.

Because the LogKit is quite complex, we will stick to explaining how to use the framework in an application. As with Cocoon, the Avalon logging framework can be extended with additional components. For more information on extending the LogKit,

refer to Appendix B, which discusses the Java API and the LogKit web site. You can find a list of relevant Internet links in Appendix C, "Links on the Web."

One of the advantages of the LogKit, from an application's point of view, is that it doesn't have to worry about the question of where the log messages will be logged. The framework takes care of this. The messages only need to be logged with a specific log level, and that's it. In order to log something, you need a logger. The most common methods of the `Logger` class are shown in Listing 8.13.

Listing 8.13 The *Logger* Class

```
package org.apache.log;

public class Logger
{
    public final boolean isDebugEnabled();
    public final void debug( final String message, final Throwable
     throwable );
    public final void debug( final String message );

    public final boolean isInfoEnabled();
    public final void info( final String message, final Throwable throwable );
    public final void info( final String message );

    public final boolean isWarnEnabled();
    public final void warn( final String message, final Throwable throwable );
    public final void warn( final String message );

    public final boolean isErrorEnabled();
    public final void error( final String message, final Throwable
     throwable );
    public final void error( final String message );

    public final boolean isFatalErrorEnabled();
    public final void fatalError( final String message, final Throwable
     throwable );
    public final void fatalError( final String message );
}
```

For each different log level, there is a method that allows the application to test whether that particular level is currently enabled. In addition, there are two methods for actually logging the message with the chosen level. The first one takes a text message as input, and the second logs a text message together with an exception.

For example, if you want to log a message with the level "warn," you can simply call `warn("This is a warning")` on the `Logger` object. This method then logs the message if the log level is set to "warn," "info," or "debug." The logger then checks implicitly to see if the log level is enabled. It does not log the message if it is disabled.

So there is no need to call the `isWarnEnabled` method beforehand, except for performance reasons. Imagine a call such as `warn("Parameters: " + p1 + ", " + p2 +`

", " + p3). Even if the warn level is not enabled, the string concatenation takes place. If p1, p2, and p3 are complex objects that perform heavy tasks to provide their string representation, this could take a long time. So it's best to always test the log level before logging—for performance reasons.

So, using the LogKit, logging is really very simple. For a component developer, the most difficult part is deciding where to log what information and at which log level. Unfortunately, it is up to the developer to make these decisions. A good rule of thumb is to log as much as possible at the debug level, such as when a method is entered and when it returns. This helps you find bugs or problems with your own component.

Another rule of development is that errors always occur where there is no logging. The reason for this is that if you decide to log a particular action, you probably will be sure to code that part correctly. However, if you forget to add logging, you might neglect to code that part correctly as well.

As we detail in Chapter 11, "Designing Cocoon Applications," it is especially important to log when you pass control to other systems (such as when your component accesses information from another system). Logging when control leaves your system and when it returns helps you find bottlenecks. Also log the data received, because it might not be in the format you expected.

There is only one problem left to solve: How do you get a Logger object? If your component implements the Loggable interface, shown in Listing 8.14, the component manager automatically gives you the Logger object just after the constructor is invoked and before Avalon's life cycle interfaces are tested.

Listing 8.14 The *Loggable* Interface

```
package org.apache.avalon.framework.logger;

public interface Loggable
{
    void setLogger( org.apache.log.Logger logger );
}
```

For convenience, Avalon provides the org.apache.avalon.framework.logger.AbstractLoggable class from which your component can inherit. This abstract class implements the Loggable interface and provides a getLogger() method returning the logger.

We have now finished our tour through the basics of the Avalon framework. You know about the component life cycle, and you know how to use the logging facilities. Because this is important information, we will now recap the points learned so far.

The Whole Story about Component Handling

We have explored many details of the Avalon framework that are important when you write your own Avalon components. Let's now summarize this before we take the big step and explore SAX and then the core of Cocoon.

Avalon follows the IoC pattern. This means that components are configured and initialized from the "outside." The component manager is the heart of Avalon. It can be used to look up and release components for a given role.

If a component is looked up, the component manager tests the component for several interfaces in order to initialize and configure the component. The configuration of a component takes place in cocoon.xconf.

When a component is released, the component manager tests the component for several interfaces so that the component can perform housekeeping duties.

For improved performance, components can be declared thread-safe or poolable. Different threads can use thread-safe components at the same time. Poolable components are recycled for reuse.

All components use the Avalon LogKit for logging. If a component implements a special interface, it automatically gets a logger component, which can be used to log messages with different log levels.

Now you know how to deal with Avalon components, how they are managed, and how to log messages. Components used in a Cocoon pipeline, such as transformers, do all this, so this is important stuff when you build your own components. However, something else you need to understand in order to build a transformer is how SAX events flow from one component to the next.

SAX Event Handling

The most striking difference between the previous versions of Cocoon and the current release (included on the companion CD) is the XML processing. It has shifted from the memory-consuming DOM model to the event-based SAX model. Whereas the DOM model creates Java objects in main memory that represent the XML document, the SAX model is event-based.

The SAX model consists of a set of interfaces and classes. We will not present every detail of the SAX model, because that would make this book quite heavy. Instead, we will focus on the essential parts.

The most important part of SAX is the set of events sent by the XML parser to a component that is "listening" for them. This component can then decide what to do by

acting on the incoming events. So you have two objects interacting with one another: one that sends the SAX events, and one that receives them.

The usual situation is when a parser parses an XML document and sends the data to a receiving component. In the discussion of looking up components from the component manager at the beginning of this chapter, we introduced the Parser role. This parser uses some SAX interfaces. Let's have a look at this role again (this is not the complete interface used in Cocoon; it's an abbreviated version):

```
package org.apache.cocoon.components.parser.Parser;

import java.io.IOException;
import org.xml.sax.ContentHandler;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.ext.LexicalHandler;

public interface Parser
extends org.apache.Avalon.framework.component.Component
{

    String ROLE = "org.apache.cocoon.components.parser.Parser";

    void setContentHandler(ContentHandler contentHandler);

    void setLexicalHandler(LexicalHandler lexicalHandler);

    void parse(InputSource in) throws SAXException, IOException;
}
```

The parser parses an XML document described by the class `org.xml.sax. InputSource`. This class wraps a stream of bytes or characters that can be read using the usual Java IO classes. Using this stream, the parser reads the XML document and generates SAX events. These events are sent to an object previously set with the `setContentHandler()` method. The `ContentHandler` interface describes the set of events sent by the parser.

Although the common terminology is "sending SAX events," this is not precisely what happens when it comes to Java. This sending of events is actually implemented by invoking methods. The parser invokes a method on the content handler in order to signal the event. The content handler consumes this event by implementing the method. When the method returns, the parser can send the next event (it can invoke the next method).

The methods of the `ContentHandler` interface that are used the most are shown in Listing 8.15. The most important events are the start and end of the document, the start and end of elements, and character data. To find out about the start and end of the document, the content handler must implement the `startDocument` and `endDocument` methods. These are the first and the last methods invoked by the parser. The start

document event can be used to initialize the content handler, and the end document event indicates that the whole document has been parsed.

Listing 8.15 The *ContentHandler* Interface

```
package org.xml.sax;

    public interface ContentHandler
{
    public void startDocument () throws SAXException;
    public void endDocument()   throws SAXException;

    public void startElement (String namespaceURI, String localName,
                          String qName, Attributes atts)
    throws SAXException;
    public void endElement (String namespaceURI, String localName,
                        String qName)
    throws SAXException;

    public void characters (char ch[], int start, int length)
    throws SAXException;
}
```

In general, an XML document is assembled from three different constructs: elements, attributes, and characters. These types are honored by different SAX events. An element always consists of an opening tag and a closing tag. Even if you choose the abbreviated syntax, such as `<tag/>`, or if the element has no children, this is interpreted as `<tag></tag>`. The XML parser signals the opening and closing of an element with an individual event.

The start element event signals the opening of an element. It is indicated by calling the `startElement` method on the content handler. This method has four parameters. The first one is the element's namespace URI, such as `http://mynamespace.com` or `null` if the element does not have a namespace. The second argument is the element's local name. The third argument is the raw name, which means that it is exactly like the element is written in the XML document. For example, if you use namespaces with prefixes, such as `<myprefix:thetag>`, the local name of this element is `thetag` and the raw name is `myprefix:thetag`. So it is always best to check the namespace URI and the local name to test for a specific element. The fourth argument of the start element event is an `Attributes` object that contains an element's attributes and provides methods to get all the attribute names and their individual values.

The end element event is indicated by invoking the method `endElement`, which has three arguments. These three arguments are exactly the same as the first three arguments of the start element event. Because the attributes are defined with an element's opening tag, they are not sent again with the end element event.

Finally, regular character data is handled by calling the `characters` method. It has three parameters. The first one is a character array, and the remaining two indicate from which position and up to which position in this array the character data can be found. Note that the parser is free to chunk the character data any way it wants, so you cannot count on all of the character data content of an element's arriving in a single `characters` event. It is therefore necessary to concatenate the characters received by continuous `characters` events.

The content handler interface provides some more methods that are invoked due to the corresponding SAX events, but they are very rarely needed. In addition, the SAX model offers a second interface called `org.xml.sax.ext.LexicalHandler`, which a component can conform to. If the XML parser gets such a component, it creates even more events, such as reporting DTDs. For more information, we suggest that you visit the SAX home page. Its link is http://www.saxproject.org/.

Let's summarize what we have just covered by going through a simple example. We will look at an XML document and some of the SAX events generated by the parser for it. Here is the document:

```
<?xml version="1.0"?>
<page title="mypage">
    <paragraph>
       <text>Hello</text>
    </paragraph>
    <footer/>
</page>
```

The following listing shows the most important SAX events. Everything starts with the start document event, followed by the events for each opening tag of an element, the events for closing the tags, and the character events.

```
Start Document
Start Element: page with attribute title
Start Element: paragraph
Start Element: text
Characters   : Hello
End Element  : text
End Element  : paragraph
Start Element: footer
End Element  : footer
End Element  : page
End Document
```

We explained the SAX model by using a parser as the component that generates the events. But more generally speaking, not only parsers can generate SAX events, but also other components. If a component has a content handler, it can simply invoke the `startDocument` method, the `startElement` method, and so on by hand. This creates an

XML document (or the SAX events for this document) by invoking methods in a distinct order.

Furthermore, Cocoon uses real pipelines for XML processing. This means that more than two components are interacting. One component, called `XMLProducer`, generates SAX events for a whole XML document. This can be either a parser or a component that directly invokes the methods. The producer sends the SAX events to a component called `XMLConsumer`. Both interfaces are shown in Listing 8.16.

Listing 8.16 The *XMLProducer* and *XMLConsumer* Interfaces

```
package org.apache.cocoon.xml;

import org.xml.sax.ContentHandler;
import org.xml.sax.ext.LexicalHandler;


public interface XMLProducer {

    void setConsumer(XMLConsumer consumer);

}

public interface XMLConsumer
extends ContentHandler, LexicalHandler {
}

public interface XMLPipe
extends XMLConsumer, XMLProducer {
}
```

`XMLConsumer` is a combination of the `ContentHandler` and `LexicalHandler` interfaces discussed earlier. This makes an XML consumer a component that can understand all possible SAX events. `XMLProducer` has only one method—`setConsumer`. Invoking this method sets the XML consumer for the producer. When the XML producer starts sending the SAX events, all these events are sent to the consumer.

If the XML consumer is also an XML producer, we speak of an `XMLPipe`. The interface is also shown in Listing 8.16. An XML pipe is a component that can receive SAX events and also send them on to another XML consumer.

In Cocoon we define the XML pipelines in the sitemap by specifying a generator, one or more transformers, and a serializer. Implemented in Java with the interfaces we have just explored, this means that the generator is an XML producer, a transformer is an XML pipe, and the serializer is an XML consumer. As you will see later in this chapter, there are some more methods that make a component a generator, a transformer, or a serializer.

With the introduction of the basic XML pipeline components and the description of SAX handling, we have finished the explanation of the required models. In itself, this

information might be enough for you to start writing a new transformer, but you also need to understand the environment these components exist in. You need to look at the internals of Cocoon and, for example, find out which part is responsible for processing the request and how pipelines are built.

Cocoon Internals

Referring back to Figure 8.1, we will now explore the Cocoon architecture from the top down. We will start with the core processing engine and move to the various components that are needed during processing.

The most important component is the Cocoon Processor. This component is the gateway to Cocoon. The processor encapsulates the whole Cocoon processing framework. It generates a document triggered by a request. So this is the processing of the usual request-response cycle.

As you can see from the interface shown in Listing 8.17, the method `process` has only one argument: an `Environment` object. This environment describes a single request and abstracts from the context Cocoon is currently running in. As you will see later, the `Environment` object is the interface between the outside world and Cocoon—or, more precisely, between the servlet or the command-line interface and Cocoon.

Listing 8.17 The *Processor* Interface

```
package org.apache.cocoon;

import org.apache.cocoon.environment.Environment;

public interface Processor {

    String ROLE = "org.apache.cocoon.Processor";

    boolean process(Environment environment)
    throws Exception;
}
```

The environment in which Cocoon is embedded is responsible for bootstrapping the system. This includes creating a component manager, reading cocoon.xconf, and providing a processor component. If you are using Cocoon in a supported environment, such as a servlet engine or from the command line, you really don't have to worry about these tasks. And you don't have to worry about creating an `Environment` object and calling the `process` method yourself. These tasks are already implemented by the different environments, but taking a look at these objects will help you understand what happens when a request is sent to Cocoon.

When a Cocoon process is triggered by the surrounding environment, such as an incoming request's being routed to the Cocoon servlet running in the servlet engine, an environment object is created and is passed on to the processor. The `Environment` object

gives the Cocoon processor all the information that is needed to process the request. The most important methods of the environment are shown in Listing 8.18.

Listing 8.18 The *Environment* Interface

```
public interface Environment
{

    void setContentType(String mimeType);
    void setContentLength(int length);
    void setStatus(int statusCode);
    OutputStream getOutputStream() throws IOException;
    Map getObjectModel();

}
```

The most important method of the `Environment` object is the `getObjectModel` method. It returns a map containing key-value pairs. This object model is passed to nearly every sitemap component. It contains at least two important objects: `Request` and `Response`. The `Request` object describes the current request by giving information about what URI is requested, which parameters are used, the client's user-agent, and so on. The `Response` object is the counterpart of the `Request` object: It describes the response. Listing 8.19 contains some important methods of these interfaces.

Listing 8.19 The *Request* and *Response* Interfaces

```
package org.apache.cocoon.environment;

public interface Request
{
    // get information about the requested URI
    String getSitemapURI();
    String getProtocol();
    String getScheme();
    String getServerName();
    int getServerPort();

    // get parameters
    String getParameter(String name);
    Enumeration getParameterNames();
    String[] getParameterValues(String name);

    // get headers
    String getHeader(String name);
    Enumeration getHeaderNames();

    // attributes for exchanging information
    Object getAttribute(String name);
    Enumeration getAttributeNames();
    void setAttribute(String name, Object o);

}
public interface Response
```

```
{
    // add headers
    void setHeader(String name, String value);
    void addHeader(String name, String value);
}
```

As the object model is passed to most components inside Cocoon, all these components have access to the `Request` and `Response` objects. The class `org.apache.cocoon.environment.ObjectModelHelper` defines some constants for the keys to use with the object model: `REQUEST_OBJECT` and `RESPONSE_OBJECT`. So if you have the object model, you can retrieve the request object by a call of

`get(ObjectModelHelper.REQUEST_OBJECT)`. The call to get the `Response` object is similar. The `ObjectModelHelper` class also provides two static methods, `getRequest()` and `getResponse()`, which take the object model as a parameter. They return the corresponding object from the object model. We advise using these two methods instead of using the key directly to get the request or response.

If you are familiar with the servlet API, you will have noticed that these two objects are very similar to the `HttpServletRequest` and `HttpServletResponse` objects. In fact, the `Request` and `Response` objects are modeled using the servlet variants as a base. But there is one important difference for security reasons: Except for setting headers, it is not possible to change anything in the response directly. It is not possible to get the output stream the response is written to. Nor is it possible to set the mime type or the length of the response.

Now that we have examined the classes and interfaces that make up the gateway into Cocoon, it is time to take a detailed look at how the sitemap processing takes place when a request is received. Up to now you have seen a pipeline represented as XML tags in the sitemap. Now you will see which objects actually do the work behind the scenes.

Processing the Sitemap

When the processor is invoked to process an environment object, nearly all the components needed from this point on are looked up from a component manager instead of being directly instantiated. This makes the whole architecture very flexible and improves the performance, because most components are either thread-safe or poolable.

The processor now needs the sitemap in order to be able to execute the request. Each sitemap—the main sitemap and the possible subsitemaps—is handled by a sitemap manager. Each sitemap has its own manager component. The processor looks up the sitemap manager component for the main sitemap and passes the `Environment` object to the manager, so the processing is passed from the processor to the sitemap manager.

The main task of the sitemap manager is to manage a sitemap component. This component is the real object representation of sitemap.xmap or any subsitemap. After the manager has access to this sitemap component, it passes the environment object to the

sitemap object. So again, the task of processing the request has moved on to another object. The sitemap object then extracts the necessary information and processes the pipelines section from top to bottom (this process is discussed in Chapters 4 and 6).

The processing of the sitemap results in the lookup of even more components. The first thing the sitemap object does is create pipeline objects that describe the current request. The `StreamPipeline` is a pipeline object that describes the whole document-creation process. It is initialized by the sitemap object with the required information or components to create the document.

The Stream Pipeline

The stream pipeline outputs its result into a byte stream, which is provided by the `Environment` object. The servlet environment passes the byte stream back to the client, and the CLI writes the file on the hard drive using this stream.

A stream pipeline itself does not create any output. It manages other components that actually do the work. Depending on the pipeline definition in the sitemap, a stream pipeline can consist of either a sole `Reader` or an `EventPipeline` in combination with a `Serializer`. We will examine these two different types in a moment.

The more general stream pipeline, which can be thought of as a wrapper around one of the two types just mentioned, has nothing to do with XML. It simply outputs a byte stream. This can be either a binary file, an image, or the text of an HTML page. The event pipeline, in contrast, is a SAX-only XML processing pipeline. This means that the output of this pipeline consists of SAX events. This is the reason that a serializer is then needed to serialize the events back into a byte stream.

When the processing of the sitemap starts as described in Chapter 6, the pipelines section of the sitemap is executed top-down. Whenever an element is reached, an action is executed in the sitemap object.

For example, if a `<map:match pattern="helloworld"/>` is reached, the default matcher is looked up. It is initialized with the current object model and the pattern that is to be tested. The result of calling this matcher influences the flow through the sitemap.

During the processing of the sitemap, several other components are also created and used: `Matchers`, `Selectors`, `Actions`, `Generators`, `Transformers`, `Serializers`, and `Readers`. When we explained the flow through the sitemap in Chapters 4 and 6, we showed that matchers, selectors, and actions are processed immediately. All other component types are assembled to build the event pipeline. This fact is reflected in the different handling of these components. Whereas matchers, selectors, and actions are looked up, executed, and released each time a corresponding statement is found in the sitemap; generators, transformers, serializers, and readers aren’t looked up at this stage. Instead, only the name of the component to be used is passed on to either the stream

pipeline or event pipeline. This information passed on to the pipeline objects is used later by the pipeline objects to build the real processing pipeline.

After the stream pipeline and event pipeline (if required) have all the information needed to process the request, the stream pipeline is executed. Calling the process method of the stream pipeline starts this. This method receives only the environment as an argument, so the processing of the request is moved from the sitemap manager object to the stream pipeline object.

Now that you have seen a general overview of how the processing takes place, let's look at the different pipeline types in more detail.

The Reader Component

A stream pipeline can consist solely of a reader. The stream pipeline knows the name of the reader to use. It looks it up using the component manager and a role selector for the role of Reader. Listing 8.20 shows the `Reader` interface.

Listing 8.20 The *Reader* Interface

```
package org.apache.cocoon.reading;

import org.apache.avalon.framework.component.Component;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.environment.SourceResolver;
import org.xml.sax.SAXException;

import java.io.IOException;
import java.util.Map;
import org.apache.cocoon.ProcessingException;
import org.xml.sax.SAXException;

import java.io.IOException;

public interface Reader
extends Component
{
    String ROLE = "org.apache.cocoon.reading.Reader";

    void setup(SourceResolver resolver, Map objectModel,
               String source, Parameters par)
    throws ProcessingException, SAXException, IOException;

    void setOutputStream(OutputStream out) throws IOException;

    void generate()
    throws IOException, SAXException, ProcessingException;

}
```

The reader is initialized by the invocation of two methods. The first one is the `setup` method. It passes all relevant information from the sitemap to the reader. The first component, the source resolver, can be used to get data from all possible sources. We will explore this component later in this chapter.

The object model contains the necessary information about the current request. You have already learned that it at least contains the `Request` and `Response` objects. The third argument, the source string, is the value of the attribute `src` used in the pipelines section of the sitemap. For example, if the sitemap contains `<map:read src="helloworld.jpg"/>`, the source argument contains the string `"helloworld.jpg"`. The fourth parameter is a `Parameters` object; it is the same class we saw during our tour through Avalon. This `Parameters` object contains all parameters written in the sitemap for this component, such as `<map:parameter name="reader_parameter" value="the_value"/>`.

The second method invoked on the reader is the `setOutputStream` method. Through this method, the reader gets access to the output stream that is later returned to the client. For example, the resource reader can read an image from the local hard drive and write the contents of the image to the output stream.

The function that is called on the reader interface and that then sets off the actual processing inside a reader component is the `generate` method.

As you can see from this description, implementing a reader is very simple. A new reader must implement the interface just described and, in particular, the three methods just mentioned. However, as we mentioned at the beginning, we have not yet shown you all the methods of the interface. The reader has some additional methods that must be implemented. But they are really very easy to implement, as you will see in the next chapter. Trust us.

Before you put this book aside and start implementing your own reader, please stay with us for the remainder of this chapter. The reader is only one possibility for the processing of the stream pipeline. It gets more complicated—and, of course, more interesting—if a real XML processing pipeline is established.

The Event Pipeline and Serializer

The stream pipeline can also be assembled from an event pipeline and a serializer. In this case, a SAX-based XML processing pipeline is established. Remember, this XML processing pipeline starts with an XML producer followed by XML pipe objects and ends with an XML consumer.

The pipeline's end point is the serializer. The `Serializer` interface therefore inherits from the XML consumer interface, as shown in Listing 8.21.

Listing 8.21 The *Serializer* Interface

```
package org.apache.cocoon.serialization;

import java.io.IOException;
import java.io.OutputStream;
import org.apache.cocoon.xml.XMLConsumer;

public interface Serializer extends XMLConsumer
{

    String ROLE = "org.apache.cocoon.serialization.Serializer";

    void setOutputStream(OutputStream out) throws IOException;

}
```

Because the serializer is the component that creates the output returned to the client, it gets the output stream via the `setOutputStream` method. The stream pipeline sets this output stream.

The event pipeline is regarded as a single XML producer that sends SAX events to the serializer. So the stream pipeline looks up the serializer using the component manager and sets it as an XML consumer for the event pipeline. Then the processing of the request is passed to the event pipeline.

The event pipeline already has the information on what generator and which transformers to use from the sitemap object. It looks up all these components using the component manager. First it gets the generator to use. The `Generator` interface is shown in Listing 8.22.

Listing 8.22 The *Generator* Interface

```
package org.apache.cocoon.generation;

import java.io.IOException;
import java.util.Map;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.environment,SourceResolver;
import org.apache.cocoon.xml.XMLProducer;
import org.xml.sax.SAXException;


public interface Generator extends XMLProducer
{
    String ROLE = "org.apache.cocoon.generation.Generator";

    void setup(SourceResolver resolver, Map objectModel,
              String source, Parameters par)
    throws ProcessingException, SAXException, IOException;

    void generate()
    throws IOException, SAXException, ProcessingException;
}
```

The generator inherits from the XML producer interface. Therefore, this is the component that really creates the SAX events as the start of the XML processing pipeline. For example, the file generator sends the events based on the input of an XML document.

But before the SAX events are sent, the generator is initialized for the current request. The event pipeline therefore calls the generator's `setup` method. By invoking this, the generator gets access to the current source resolver, the object model, the value of the `src` attribute used in the sitemap, and the possible parameters set in the sitemap for this generation step. This is the same as initializing a reader.

When the whole pipeline has been established, the XML processing is started by a call to the `generate` method. But before this can happen, the generator needs an XML consumer it can send the SAX events to. Therefore, the event pipeline looks up all the transformers used for this pipeline by questioning the component manager. The `Transformer` interface is shown in Listing 8.23.

Listing 8.23 The *Transformer* Interface

```
package org.apache.cocoon.transformation;

import java.io.IOException;
import java.util.Map;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.environment,SourceResolver;
import org.apache.cocoon.xml.XMLPipe;
import org.xml.sax.SAXException;

public interface Transformer extends XMLPipe
{
    String ROLE = "org.apache.cocoon.transformation.Transformer";

    void setup(SourceResolver resolver, Map objectModel,
               String source, Parameters par)
    throws ProcessingException, SAXException, IOException;
}
```

As a transformer receives SAX events—either from a generator or from a preceding transformer—it conforms to the XML consumer interface. But a transformer also sends events either to the serializer or to the next transformer. So it implements the XML producer interface, too. Therefore, the `Transformer` interface inherits from the `XMLPipe` interface, combining XML producer and XML consumer.

Like a generator, a transformer is first initialized by a call to the `setup` method. This method has the same arguments as a generator: the source resolver, the object model, the value of the `src` attribute used for this transformer in the sitemap, and the parameters passed to this transformer from the sitemap.

When all components have been looked up, the XML processing pipeline is assembled. The generator gets the first transformer as the XML consumer. The first transformer gets the second transformer as the XML consumer, and so on. The last transformer sends its SAX events to the serializer.

Finally, the XML processing starts by invoking the `generate` method on the generator. The generator follows the steps necessary to get the initial XML, such as by reading an XML document. This document's events are then sent to the first transformer.

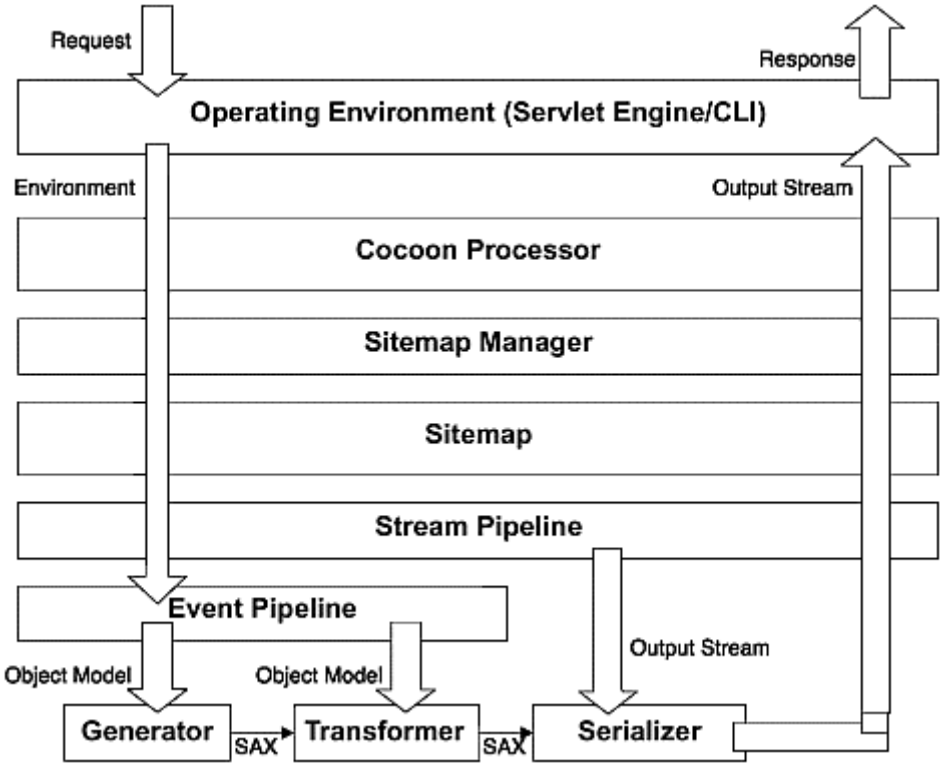
The transformer—if it doesn't change the data—passes all events unchanged to the next component in the pipeline, such as a second transformer. This second transformer passes the events to the next transformer or to the last component in the pipeline—the serializer. The serializer creates the output document from the received events. You will get a closer look at the event pipeline when you build your own components in Chapter 9.

Document generation takes place with the simple-looking call to the `generate` method on the generator. As you have seen, before this generation is started, a lot of actions are performed, and many components are looked up. Of course, when the generation process is finished, all components are released properly.

After going through all the information on the past few pages, now is perhaps a good time to summarize what you have learned. We will use a figure to do so.

Figure 8.2 summarizes the different steps of request processing.

Figure 8.2. Processing steps.



When a request enters the surrounding environment, such as the servlet or the CLI, it is wrapped by an `Environment` object. This object is passed to the core Cocoon processor. The processor gets a sitemap manager for the root sitemap and passes the `Environment` object to this component.

The manager looks up the sitemap object and passes the `Environment` object to this object. The sitemap processes the sitemap and generates the stream pipeline and event pipeline components. During the processing of the sitemap, the matchers, selectors, and actions are looked up, executed, and released. The event pipeline gets the information on what generator and which transformers to use later. The stream pipeline is initialized with the name of the serializer to use.

After the processing of the sitemap is finished, the sitemap component passes the `Environment` object to the stream pipeline. The stream pipeline looks up the serializer and initializes it with the output stream provided by the `Environment` object. The `Environment` object and the serializer are passed to the event pipeline.

The event pipeline looks up the generator and the transformers and initializes all of them. It does not pass the `Environment` object or the output stream to those components. But they all get access to the object model containing the most important information.

When the stream pipeline starts the processing by invoking the generator, the generator sends SAX events to the transformer. Through the pipeline, the events arrive at the

serializer. The serializer creates the output document by writing to the provided output stream. After all the SAX events are executed, the processing is finished.

In the case of a reader, the picture is simpler. You can forget the event pipeline and replace the serializer with the reader component. When the stream pipeline is invoked, it looks up the reader and gives it the output stream. After that, the reader is executed. It writes the data to the output stream, and the processing is also finished.

The next piece of the jigsaw we will look at is how source resolving takes place in Cocoon.

Source Resolving

One common problem, not only present in Cocoon, is the resolving of sources. By this we mean getting content from different sources, such as the file system, HTTP, or FTP. Cocoon is very flexible in this way, because it can be extended with any custom protocol not already provided by the Java Virtual Machine.

In Chapter 6, we talked about the various protocols available in Cocoon and how Cocoon handles the source definitions used as input for the sitemap components. For example, if you use `<map:generate src="helloworld.xml"/>`, this source is resolved according to the location of the current sitemap.

As you can probably guess, source resolving is done by another component: the `SourceResolver`. Unfortunately, this is not an Avalon component, so it cannot be looked up using a component manager. But before you get too disappointed, the good news is that the source resolver is passed to each sitemap component during the assembling phase, as described earlier in this chapter. Listing 8.24 presents the `SourceResolver` interface.

Listing 8.24 The *SourceResolver* Interface

```
package org.apache.cocoon.environment;

import org.apache.cocoon.ProcessingException;
import org.xml.sax.SAXException;

import java.io.IOException;

public interface SourceResolver
{
    Source resolve(String systemID)
    throws ProcessingException, SAXException, IOException;
}
```

The source resolver has only one method, `resolve`, which has only one argument, the system identifier. This string is the unique definition of a source—in other words, a URI. As you are used to, this URI can be absolute or relative. The source resolver

automatically resolves this for you and returns an object conforming to the `Source` interface, as shown in Listing 8.25. Be sure to resolve every URI using the source resolver. Do not use the java.net package for URIs or the java.io classes for files. The source resolver is a more powerful approach that makes using these packages superfluous.

Listing 8.25 The *Source* Interface

```
package org.apache.cocoon.environment;

import java.io.IOException;
import org.apache.avalon.excalibur.pool.Recyclable ;
import org.apache.cocoon.ProcessingException;
import org.xml.sax.ContentHandler;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;

public interface Source extends Recyclable
{

    InputStream getInputStream()
    throws ProcessingException, IOException;

    InputSource getInputSource()
    throws ProcessingException, IOException;

    void toSAX(ContentHandler handler)
    throws SAXException, ProcessingException;
}
```

The `Source` object can be used to get an input stream from which to read the data. To simplify XML handling, the `Source` object offers two methods especially for SAX event handling. The `getInputSource` method returns an `InputSource` object that can then be used by an XML parser to generate SAX events. The `toSAX` method goes one step further: It sends the SAX events directly to a content handler.

Although the source object is not an Avalon component, it implements the `Recyclable` interface. During the allocation of a `Source` object and during the fetching or parsing of the data, the `Source` object might collect other objects that must be carefully released. This can be done in the `recycle` method. So a component getting a `Source` object from the source resolver must always recycle this `Source` object! The usual handling of a `Source` object is shown in Listing 8.26.

Listing 8.26 Using a *Source* Object

```
public void sourceDemo(SourceResolver resolver,
                       String URI)
{
    Source source = resolver.resolve(URI);
    try {
        source.toSAX(contentHandler);
```

```
    } finally {
        if (source != null) source.recycle();
    }
}
```

As should be obvious from this example, source resolving with Cocoon is very simple. The only thing you have to do is call the `Resolve` method on the resolver. The source object you get is already optimized for XML processing with the SAX model. And don't forget to recycle it afterwards.

Writing your own component for Cocoon is not so complicated. As soon as you understand the basic concepts, it most often comes down to simply implementing an interface, such as the reader or transformer interface. Many common tasks are already contained in Cocoon by special Avalon components, such as source resolving.

Enough Theory

Still with us? Good. We did warn you that this chapter might be slightly difficult to digest. But it is important to learn how to drive before you get into a car. You can learn by just driving off, but how many other cars will you run into on the way? It is the same with Cocoon. In our experience, any programmer who tries to write a Cocoon component without grasping the basic concepts first will learn the hard way. In addition, it is important to remember that Cocoon does a lot of the work for you.

This chapter laid out the basics of developing your own Java components to use with Cocoon. We explored the Cocoon architecture top-down, introducing the most common objects. Nearly all objects inside Cocoon adhere to the Avalon component model, so we started with Avalon component management and then moved on to the logging facilities and the SAX model for XML processing.

In Chapter 9, you will use this knowledge to write some components you can use in your Web applications. So start your Java Development Environment and continue with the next chapter.

Chapter 9. Developing Components for Cocoon

Cocoon provides a wide variety of components, such as the various transformers and generators, that can be used to build different types of applications. As you have seen throughout this book, these applications can range from simple web pages to quite complex solutions such as a news portal, where data is accessed over the Internet and the user configuration is stored in a database. However, there will be times when you need a component that is not in the standard distribution. One of Cocoon's great advantages is that it can be easily extended with additional components.

One of the first Cocoon-based applications we designed required the integration of a legacy system via a proprietary protocol. This was not provided for by the available components in the Cocoon distribution. So, in order to still use Cocoon as the publishing engine, we wrote our own generator that could connect to the new system. Because we wrote a component that implemented the given Cocoon interface, we could integrate it into Cocoon just by configuring the details in the cocoon.xconf file. So adding new components is just a matter of configuration (after you've written and tested them, of course).

By the end of this chapter, you will have developed new components that extend Cocoon with functionality that isn't included in the standard distribution. These components range from the simple, such as a reader, to the more-complex, such as a transformer. Apart from sitemap components, you will also see how to develop a new protocol for Cocoon and how to centralize functionality in Cocoon by developing a mail component that can be used by other components.

What Is Needed to Develop Cocoon Components

Before starting to write a particular component, you first need to answer the most important question: What do you need in order to write your own components for Cocoon? Well, first of all, Cocoon. You already know that, so let's move on. Actually, you need only one other thing: a Java Developer Kit (JDK).

Because Cocoon is based on the Java 1.2 platform, you need a developer kit that conforms to this version. Actually, you can use either JDK version 1.2 or 1.3. JDK 1.4

recently was released. Unfortunately, it doesn't work with the version of Cocoon covered in this book. So, if you want to use JDK 1.4, you will have to get a more recent version of Cocoon from the Cocoon web site. You need JDK and not just the Java Runtime Environment (JRE). You can download versions of the JDK from places such as the Sun web site.

Theoretically, that's all you need—that is, if you like starting the Java compiler from a command-line interface. However, it is a lot easier to use an integrated development environment (IDE). Regardless of whether you prefer the CLI solution or the IDE one, you must add all the necessary Java archives (JARs) to the classpath when building new components. When you installed Cocoon into the servlet engine, all the necessary JAR files were copied to the WEB-INF/lib directory. Refer to your IDE user manual or to the documentation of the command-line tools for more information on how to add all the necessary JAR files to the classpath so that they are available to the compiler.

After you have successfully compiled a component, you need to "deploy" it. By deploying, we mean adding your new component to the installed Cocoon servlet so that it can be used in a pipeline. Deploying a new component in a Cocoon environment is the same for each type of component, so we will describe the necessary steps first, before looking at the individual components. Remember that this is not anything specific to Cocoon, but rather the standard way of packaging Java classes for deployment. Here are the steps:

1. Build a JAR file containing the new component(s).
2. Shut down the servlet engine.
3. Add the new JAR file to the WEB-INF/lib directory.
4. Configure the component either into the sitemap or into the cocoon.xconf.
5. Create a pipeline containing the component.
6. Start the servlet engine.
7. Test the component.

After you have set up your development environment, it's "tool time." But what can you develop? There are so many possibilities with Cocoon that it is very difficult to choose. So let's briefly recap the types of components you can develop to extend Cocoon.

Sitemap Components

First of all, there are the sitemap components: actions, generators, transformers, serializers, matchers, and selectors. You used these various component types when we looked at Cocoon from the user perspective in Chapters 4 and 6. Depending on the

application, you might need to write an action, generator, reader, transformer, or sometimes maybe even a selector. But in the case of serializers and matchers, the situation is slightly different, because Cocoon already provides common implementations. During the work you have done with Cocoon, you have not yet had to write a serializer or a matcher.

The component most likely to be written is the action. This type of component is ideal for integrating functionality that might already exist.

You will start by writing a simple action. Then you will build a reader and a generator. After that, you will increase the complexity by writing your own transformer. You will finish with a simpler component type: your own selector. As soon as you understand the basics of Cocoon component development, it is easy to extend your knowledge to the component types not covered in this chapter.

But let's first have a look at some general rules and remember some important points from Chapter 8, "A Developer's Look at the Cocoon Architecture." We know this might disappoint you, but trust us. You will benefit from the next few pages, because they set some general guidelines for writing robust and easy-to-debug components.

General Hints

Cocoon is based on the Avalon framework and adheres to the Avalon component model. Writing a sitemap component therefore includes writing an Avalon component. So it's worth repeating the different life-cycle interfaces that Avalon supports:

- `Loggable`: Provides a logger for the component.
- `Contextualizable`: Provides the context.
- `Composable`: Passes the component manager to the component.
- `Configurable`: Sets a hierarchically structured configuration.
- `Parameterizable`: Sets a flat configuration.
- `Initializable`: Initializes the component.
- `Disposable`: Cleans up the component.
- `ThreadSafe`: The component is thread-safe.
- `Poolable`: The component is not thread-safe but should be pooled.
- `Recyclable`: The same as `Poolable`, but the component can clean itself up when it is returned to the pool.

See Chapter 8 for in-depth coverage of these interfaces. We recommend that you always implement the `Loggable` interface so that the component receives a logger. If the component does not need to inherit from another class, it can simply inherit from the `AbstractLoggable` class that is also contained in Avalon. This abstract class implements the `Loggable` interface and offers the `getLogger` method to get the logger instance. If the component cannot inherit from this abstract class, it has to implement the `setLogger` method of the `Loggable` interface itself.

If the new component needs to look up other components, such as a parser, it must implement the `Composable` interface. If the component requires any configuration, you have to choose between `Configurable` and `Parameterizable`. Whenever the components only need key-value pairs for configuration, you can use `Parameterizable`. On the other hand, the `Configurable` interface gives you the full power of XML-based configuration.

If the component needs any further initialization, the `Initializable` interface is the way to go. To clean up when the component is destroyed, you can implement the `Disposable` interface. For all these interfaces, it is advisable not to implement a lifecycle interface if the component does not really require the functionality. This keeps the components as small as possible and makes them perform a bit faster.

Sooner or later you will face a situation in which after you have written your complex component, it doesn't work the way you expect when you test it. In such a situation, you will benefit from compact code, because this minimizes the lines of code you need to scan for the bug.

Another technique that helps in the debugging process is using tracing or logging features. It is a good idea if a method outputs the incoming parameters and the output result. Because you can use the provided logger for this, an implementation for a method could look like this:

```
public String getValueFor(String contextName, Object context) {
    if ( this.getLogger().isDebugEnabled() ) {
       this.getLogger().debug("getValueFor invoked with contextName=" +
     contextName
                          +"context = " + context);
    }
    String result;
    ... do whatever is needed and store it in 'result'.

    if ( this.getLogger().isDebugEnabled() ) {
       this.getLogger().debug("getValueFor() returns: " + result);
    }
    return result;
}
```

This technique helps a lot in debugging, because it is possible to have a look at the log and see which methods were called with what parameters and what the result of the method was. The examples in this book do not contain logging statements to keep them as small as possible but still understandable. But the complete source code is also contained on this book's companion CD. That version uses the logging mechanisms we have just described.

Another point we need to be clear about before we start is a component's lifetime. If possible, you should make your component thread-safe. If not, it should at least be poolable or recyclable.

For the sitemap components, Cocoon provides several abstract classes you can use to implement your own components. Some of these abstract classes already implement different life-cycle interfaces. They all implement at least the `Loggable` interface. Some of them implement the `Recyclable` interface. In these cases, it is important to not extend from any other lifetime interface, such as thread-safe. This is not allowed and would result in an exception from the component manager when such a component were looked up.

The problem with the `Recyclable` interface is that it defines a method called `recycle` that your new component must implement. So if you inherit from a `Recyclable` class, you have to take care of at least two things. First, you need to implement the `recycle` method in the component and clean up any objects you might have allocated. Second, you must also call the super implementation. The usual implementation of the `recycle` method resets all instance variables to `null` to indicate that the referenced objects can be released into garbage collection. If any of the referenced objects were components you looked up from the component manager yourself, these have to be released at this time.

But you must be careful not to reset any instance variables that were set by some of the other life-cycle interfaces, such as the `Contextualizable` or `Composable` interfaces. When a component is recycled, it is used the next time without these life-cycle interfaces being called.

Another topic you have to deal with is exceptions. In most cases, the methods raise an `org.apache.cocoon.ProcessingException`. This is a Cocoon-specific exception that is raised whenever a problem occurs during the processing. The `ProcessingException` can be used to wrap another exception. Most of the interfaces used in Cocoon can often throw only a `ProcessingException`. If you implement one of these interfaces, and if you have to deal with other exceptions inside your implementation, you have to catch the exception and throw a `ProcessingException` instead that wraps the original exception:

```
public void method()
throws ProcessingException {
   try {
      ... some statements eventually raising a ProcessingException, an
IOException and
          any other Exception as well
   } catch (IOException ioe) {
      throw new ProcessingException("IOException", ioe);
   } catch (ProcessingException pe){
      throw pe;
   } catch (Exception any) {
      throw new ProcessingException("Exception", any);
   }
}
```

From this example, you can see two important things:

- If you catch an exception and raise a new one, you must never lose the original exception. You need to pass it on with the new exception whenever possible (an example is `IOException` and `Exception`).
- If you use a type of exception that wraps other exceptions, you need to make sure that you do not wrap them unnecessarily again. This would be the case if the function you call also throws a `ProcessingException`.

Three other kinds of exceptions are commonly used inside Cocoon:

- `SAXException`: Whenever something is wrong with the SAX events, this exception is used. It is also a wrapping exception.
- `IOException`: Used whenever an I/O operation fails.
- `ResourceNotFoundException`: This is used to indicate that a source has not been found—such as if a generator tries to read an XML document that is unavailable.

If you look through the Cocoon source code (which is also on the CD), you will see a lot of try/catch blocks like the one just shown. If you strictly follow the rule of wrapping the original exception, you will always see in the top exception handler what the real cause of the exception was. If you forget somewhere in between to include the original exception, this important information gets lost, and it will be very hard to find the real bug or problem.

Enough theory for now. Let's get practical and write your own sitemap component: an action.

Actions

What can your action do? Because an action can perform an arbitrary task, there are really no limits. The one exception is that an action cannot directly influence the SAX stream of the XML processing pipeline, as you saw in Chapter 8. To keep the example as small as possible and in order to show the implementation of an action, you will develop a simple action that produces a random number.

Your random action can be used in the sitemap to generate a random number that can be used to form the name of a stylesheet. Each time a user requests a document, the background color could be different. Or, if you are interested in putting banners on your document, you can alternate between different banners by using this action.

As a starting point, we present the `Action` interface in Listing 9.1. The whole interface consists of only one method—the `act` method. You need to add your functionality here. This method gets several parameters: a redirector for possible redirect-ing, the source resolver, the object model for the current request, the value of the source attribute from the `map:act` statement, and the sitemap parameters from the sitemap. As explained in

Chapter 8, the source resolver and the object model are two important objects. The source resolver helps in obtaining any resource, such as an XML document, an image, and so on. The object model describes the current request-response cycle.

Listing 9.1 The *Action* Interface

```
package org.apache.cocoon.acting;

import org.apache.avalon.framework.component.Component;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.environment.Redirector;
import org.apache.cocoon.environment.SourceResolver;

import java.util.Map;

public interface Action extends Component {

    String ROLE = "org.apache.cocoon.acting.Action";

    Map act(Redirector redirector, SourceResolver resolver,
           Map objectModel, String source, Parameters par)
    throws Exception;
}
```

On successful completion of the task, the action returns a `Map` instance that is a collecttion of key-value pairs. Remember that if the action returns a `map` object, the sitemap statements inside the `map:act` are executed. If the action returns `null`, these are skipped. In addition, the values contained in the map can be used from other sitemap components through value substitution if the other components know the key under which a value is stored.

But even if an action does not provide any values for other sitemap components to use, it should return a `map` object to the sitemap. This `map` object can simply be empty.

So, your action generates a random number and returns this number in the map using the key "number." When the action is used, it is possible to provide a range for the random number by giving a minimum and a maximum value. If these ranges are not provided, the generated number is between 0 and 100.

Cocoon already provides some abstract classes from which your little action can inherit: `AbstractAction` and the `ComposerAction`. `AbstractAction` already implements the `Loggable`, `Configurable`, and `Disposable` interfaces. `ComposerAction` extends `AbstractAction` by also implementing the `Composable` interface. Because your action needs neither a component manager nor a configuration, you will not inherit from these abstract classes. Because you want a logger for debugging, you inherit from the `AbstractLoggable` class. In addition, your action is thread-safe, so you extend that interface, too. In fact, because an action consists of only one method, there are really rare situations in which the action is not thread-safe. Listing 9.2 shows the code.

Listing 9.2 The *RandomAction* Class

```
package cxa.acting;

import org.apache.avalon.framework.activity.Initializable;
import org.apache.avalon.framework.thread.ThreadSafe;
import org.apache.avalon.framework.logger.AbstractLoggable;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.acting.Action;
import org.apache.cocoon.environment.Redirector;
import org.apache.cocoon.environment.SourceResolver;

import java.util.HashMap;
import java.util.Map;
import java.util.Random;

public class RandomAction
  extends AbstractLoggable
  implements Action, Initializable, ThreadSafe {

    protected Random generator;

    public void initialize() {
        this.generator = new Random( System.currentTimeMillis() );
    }

    public Map act(Redirector redirector,
                   SourceResolver resolver,
                   Map objectModel,
                   String src,
                   Parameters parameters)
 throws Exception {
     int min = parameters.getParameterAsInteger("min", 0);
     int max = parameters.getParameterAsInteger("max", 100);

     int random = this.generator.nextInt(max - min + 1) + min;

     // build resulting map
     Map map = new HashMap(1);
     map.put("number", "" + random);
     return map;
    }
}
```

Listing 9.2 contains your action. As you can see from the `initialize()` function, it also implements the `Initializable` interface. When the action is created, it gets a `java.util.Random` object, which generates the random numbers.

The rest of the action consists of the implementation of the method `act`. The action usually calculates random numbers between 0 and 100. By specifying two parameters, `min` and `max`, you can override these ranges. So the first two lines evaluate any passed parameters.

The next statement generates the random number in the given range, and then the resulting `map` object is created. The map contains one key called `number`, which points to the generated number.

And that's all. The next step is to "deploy" this class as we explained earlier and to configure it in the sitemap. Add the following line to the `map:actions` section of the sitemap:

```
<map:action name="random" src="cxa.acting.RandomAction"/>
```

As an example, put a number of images into Cocoon's context directory, and name them starting with 0.jpg, 1.jpg, up to the number of images available. Then add the following pipeline to the sitemap and make sure that you set the parameter `max` according to the number of images you have:

```
<map:pipeline>
    <map:match pattern="random">
        <map:act type="random">
            <map:parameter name="max" value="5"/>
            <map:read src="{number}.jpg"/>
        </map:act>
    </map:match>
</map:pipeline>
```

Now start Cocoon, point your browser to `http://localhost:8080/cocoon/random`, and reload this page several times. Isn't it a good feeling to see your first new sitemap component in action?

Writing an action is a piece of cake. But actions are important because they allow you to write a component that does not need to know anything about XML or understand such things as SAX events. The same is true of the next component we will look at: readers.

Readers

A reader is a sitemap component that streams a document directly to the output. This document is represented by bytes, so this could be any binary information or any text information, such as a JavaScript file.

The Cocoon distribution already contains the resource reader, which can read any file from the local hard drive or by using any provided protocol, such as HTTP or FTP. So you can deliver any file on the hard drive to the client. In addition, you can, for example, use the `resource` protocol, which was explained in Chapter 6, "A User's Look at the Cocoon Architecture." With this protocol, you can extract documents from the JAR files in the classpath.

But what if you have other archives, such as ZIP files, on the hard drive, and you want to deliver a file from within such an archive to the client without wanting to unpack the archive beforehand? As you might guess, the answer is simple: Just use the reader you will now write!

Your reader will get a URI that points to a ZIP file as input, such as archives/images.zip or `http://www.newserver.com/news_2001.zip`. Because the reader must know which document it should extract from the archive, this path is appended to the URI, so the full URI for your reader will look something like this: archives/images.zip/people/matthew.jpg.

The reader uses this URI, opens the archive, and extracts the document specified by the path. The opening of an archive and the extraction of a document from an archive is very simple in Java. The java.util.zip package contains everything you need for these operations.

The most important point is the name of your reader. Let's call it the zip reader. But before we have a look at the implementation part, we should start with some general notes about a reader. Like any other sitemap component, a reader is defined by a Java interface: `org.apache.cocoon.reading.Reader`. As shown in Listing 9.3, the `Reader` interface inherits from the interfaces `SitemapModelComponent` and `SitemapOutputComponent`. The listing contains an extended Java syntax to provide all information from these three interfaces in one listing.

Listing 9.3 The *Reader* Interface

```
package org.apache.cocoon.reading;

import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.environment.SourceResolver;
import org.apache.cocoon.sitemap.SitemapModelComponent;
import org.apache.cocoon.sitemap.SitemapOutputComponent;
import org.xml.sax.SAXException;

import java.io.IOException;
import java.io.OutputStream;
import java.util.Map;

public interface Reader
    extends interface org.apache.cocoon.sitemap.SitemapModelComponent {

        void setup(SourceResolver resolver, Map objectModel,
                   String src, Parameters par)
        throws ProcessingException, SAXException, IOException;
    }
    extends interface org.apache.cocoon.sitemap.SitemapOutputComponent {
        void setOutputStream(OutputStream out) throws IOException;

        String getMimeType();
        boolean shouldSetContentLength();
```

```
    }

    String ROLE = "org.apache.cocoon.reading.Reader";

    void generate()
    throws IOException, SAXException, ProcessingException;

    long getLastModified();
}
```

When the Cocoon processor processes the request, the component manager looks up the reader and configures it according to the life-cycle interfaces it implements. Following this initialization, the processor gives additional information to the reader; the `setup` method shown in Listing 9.3 is invoked first. With this method, the reader gets the important source resolver instance, the object model for the current request, the source information from the sitemap, and all the possible sitemap parameters specified for the reader.

During this setup phase, a reader can perform any initialization it requires based on what it receives. Your zip reader does not require anything to be done during this part of the initialization.

After the setting up of the reader, the Cocoon processor asks for the document's mime type. If the reader is able to detect the document's mime type, it should return it. If this is not possible, the reader can simply return `null`. In this case, the Cocoon processor uses information from the `map:read` instruction in the sitemap. Remember that you can specify a document's mime type using an attribute of the same name: `<map:read mime-type="image/jpg" src="images/hello.jpg"/>`.

Your zip reader will not provide a document's mime type for the Cocoon processor. Instead, you need to hard-code the correct mime type into the sitemap, as just shown. The reason for this is obvious: The requested document can have any mime type, and the archive does not contain any information about the mime-type or format of an archived file. So, in order to provide this information, your reader must detect a file's format by itself.

The next piece of information that the Cocoon processor requests from the reader is the last modification date of the document in question. If the reader can provide this information, the Cocoon processor can make use of this information in order to speed up the document generation. The HTTP protocol provides special headers that are then sent from the client with the request. The server—or, in this case, Cocoon—can make use of this information and check whether the version on the client is still valid. If the client version is valid, the server informs the client of this without the need to regenerate the document. If the client version is no longer valid, the server generates the new document and returns it.

Validity checking is exactly the purpose of the `getLastModified` method. If the reader returns a nonzero value, Cocoon uses this information to check the document's

validity. If the reader returns 0, it can't determine when the document was last modified. In this case, a new document is generated on each request.

If the document still needs to be generated, the Cocoon processor invokes the `shouldSetContentLength` method. If the reader returns `true`, the processor sets the correct content length for the response. If `false` is returned, the response doesn't contain any information about the document's length. Some formats, such as PDF, require a correct content length to be set in the response.

The last information the reader gets from the processor is the output stream the reader can write the document to. This is set by the `setOutputStream` method.

After the reader is set up correctly and everything has been checked, the processor invokes the `generate` method. This starts the real process of reading the document. In the case of your zip reader, the reader gets the archive, extracts the document in question, and writes this document to the provided output stream.

Let's start with your reader. For the easy writing of readers, Cocoon already provides an abstract class that your reader can inherit from. Then you only have to focus on the really important methods. The abstract class is called `AbstractReader`. Listing 9.4 shows this class and the provided functionality. The class inherits from `AbstractLoggable` and `Recyclable`. The component manager passes a logger into the reader. The reader is pooled, and when it is passed back into the pool, the `recycle` method is called.

Listing 9.4 The *AbstractReader* Class

```
package org.apache.cocoon.reading;

import org.apache.avalon.excalibur.pool.Recyclable;
import org.apache.avalon.framework.logger.AbstractLoggable;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.environment.SourceResolver;
import org.xml.sax.SAXException;

import java.io.BufferedOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Map;

public abstract class AbstractReader
extends AbstractLoggable
implements Reader, Recyclable {

    protected SourceResolver resolver; // The source resolver
    protected Map objectModel;        // The object model
    protected Parameters parameters;   // The sitemap parameters
    protected String source;          // The source information
    protected OutputStream out;       //  The output stream
```

```
    /** Store the parameters in resolver, objectModel, source
       and parameters */
    public void setup(SourceResolver resolver, Map objectModel,
                   String src, Parameters par);

    /** Store the output stream in out */
    public void setOutputStream(OutputStream out);

    /** Return null */
    public String getMimeType();

    /** Return zero */
    public long getLastModified();

    /** Return false */
    public boolean shouldSetContentLength();

    /** Reset instance variables */
    public void recycle();
}
```

For each object set by the Cocoon processor, the abstract reader has an instance variable that stores the source resolver, the object model, the source information from the sitemap, the sitemap parameters, and the output stream. All these objects are cleaned up in the `recycle` method.

Furthermore, the `abstract` class already implements most of the additional methods. The `getLastModified` method always returns 0, the `shouldSetContentLength` method returns `false`, and the `getMimeType` method returns `null`.

So the only method missing is the `generate` method, which does the real work. The first thing you have to do is extract the URI for the ZIP archive from the source information passed to the reader. You do this by searching for the file extension .zip in the URI:

```
int position = this.source.indexOf(".zip");
if (position == -1) {
    throw new ProcessingException("The URI does not point to a ZIP archive.");
}
String zipURI = this.source.substring(0, position + 4);
```

If the URI does not point to a ZIP archive, you throw a `ProcessingException`. The next step is to extract the document name from the original URI. Remember that a URI for your zip reader looks like this: archive.zip/*documentname*. So you extract everything after .zip/ from the URI:

```
String documentName = this.source.substring(position + 5);
```

Now you have all the information you need to process the request. You now ask the source resolver for a `source` object for the archive:

```
Source archive;
try {
    archive = this.resolver.resolve( zipURI );
    ...
} finally {
    if (archive != null) archive.recycle();
}
```

What follows is nothing specific to Cocoon but everyday Java code. The `source` object provides an input stream you can use to read from the archive. You use some classes of the java.util.zip package to read this stream and search for the desired entry. When the entry is found, it is read byte by byte and is written directly to the output stream. You can see this very straightforward code here:

```
ZipInputStream zipStream = new Z
ZipEntry document = null;
boolean found = false;
do {
    document = zipStream.getNextEntry();
    if (document != null) {
        if ( document.getName().equals(documentName) ) {
            found = true;
        } else {
            // go to next entry
            zipStream.closeEntry();
        }
    }
} while (document != null && found == false);

if (document == null)
    throw new ResourceNotFoundException("The document " + documentName + "
     is not in the
archive);

byte[] buffer = new byte[8192];
int length = -1;

while ( zipStream.available() > 0 ) {
    length = zipStream.read(buffer, 0, 8192);
    if ( length > 0 ) {
        this.out.write(buffer, 0, length);
    }
}
zipStream.close();
this.out.flush();
```

The `do-while` loop reads one entry after the other from the archive and compares the name of the entry with the name of the desired document. The loop is finished either if

the document is found or if there are no further entries. In this case, a `ResourceNotFoundException` is raised.

When the document is found, the entry from the archive is read in chunks of 8192 bytes and is written directly to the output stream. After the entry is finished, everything is closed, and the reader has finished its work. If you are unfamiliar with the java.io or java.util.zip packages, just browse through the JDK documentation to see what we have done here.

Congratulations! This is your first working reader. As you can see from the number of lines of Java code, it is really very easy to write your own reader. The interfaces presented by Cocoon in addition to the provided abstract classes let you focus on the important parts of the reader.

Let's now put this component into action by first deploying it in the Cocoon web application. Then you have to add the following line to the sitemap's `map:readers` section:

```
<map:reader name="zip" src="cxa.reading.ZipReader"/>
```

Put a zipped archive with, for example, some images into Cocoon's context directory, and name this archive images.zip. Now add the following pipeline to the sitemap, and choose a document name that the zip reader should return from the archive:

```
<map:pipeline>
    <map:match pattern="zipread">
       <map:read type="zip" mime-type="image/jpg"
     src="images.zip/test.jpg"/>
    </map:match>
</map:pipeline>
```

Now start Cocoon and invoke `http://localhost:8080/cocoon/zipread`. This reads the image test.jpg from the archive images.zip. Note that you explicitly set the document's mime type to indicate that this is an image, because your reader cannot test the mime type.

We suggest that you play around with this component and perhaps extend it. The current implementation assumes that the archive name ends with .zip. What about making the ending configurable using a parameter? Just experiment a bit with the component. When you feel comfortable developing a component such as a reader, you can move on to the next sitemap component in line.

Generators

You've managed to write your own reader, so let's increase the complexity of your components a little and write a generator.

From an implementation point of view, a generator is very similar to a reader. Both generate a document. Whereas a reader can deliver any document, even binary information, a generator can send only SAX events.

You already have a reader that extracts a document from an archive. You will stick to this sample scenario and build a generator that extracts a document from an archive but also assumes that this document contains XML. The zip generator will parse the document and send the SAX events to the pipeline for further processing.

So, again, let's first have a look at the generator interface. It is shown in Listing 9.5. This interface is somewhat simpler than the reader interface. But it also extends the `SitemapModelComponent` interface and the `XMLProducer` interface.

Listing 9.5 The *Generator* Interface

```
package org.apache.cocoon.generation;

import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.sitemap.SitemapModelComponent;
import org.apache.cocoon.xml.XMLConsumer;
import org.apache.cocoon.xml.XMLProducer;
import org.xml.sax.SAXException;

import java.io.IOException;

public interface Generator
    extends org.apache.cocoon.xml.XMLProducer {
        void setConsumer(XMLConsumer consumer);
    }
    extends org.apache.cocoon.sitemap.SitemapModelComponent {
        void setup(SourceResolver resolver, Map objectModel,
                   String src, Parameters par)
        throws ProcessingException, SAXException, IOException;
    }

    String ROLE = "org.apache.cocoon.generation.Generator";

    void generate()
    throws IOException, SAXException, ProcessingException;
}
```

During the processing and after the generator and the other pipeline components are looked up, the Cocoon processor initializes the generator by calling the `setup` method. By invoking this method, the generator gets the source resolver, the object model describing the request, the source information from the sitemap, and the sitemap parameters.

In addition, the `setConsumer` method is invoked from the processor. With this method, the generator gets the next component of the XML processing pipeline. The generator sends all SAX events to this consumer.

The last method a generator has to implement is the `generate` method. Similar to readers, this is the main part of the generator. It reads whatever documents are required and generates the SAX events from the data.

In the case of your zip generator, the `generate` method opens the archive and extracts the XML document. This extracted document is passed to the XML parser, and the parser sends the SAX events to the XML consumer.

Let's start implementing. Again, Cocoon already offers some abstract classes you can inherit from. A generator has at least two: `AbstractGenerator` and `ComposerGenerator`. `ComposerGenerator` extends `AbstractGenerator` by also implementing the Avalon `Composable` interface to get a component manager. Because you need an XML parser to parse the document, you need a component manager. Therefore, you extend `ComposerGenerator`, as shown in Listing 9.6.

Listing 9.6 The *ComposerGenerator* Class

```
package org.apache.cocoon.generation;

import org.apache.avalon.excalibur.pool.Recyclable;
import org.apache.avalon.framework.component.ComponentException;
import org.apache.avalon.framework.component.ComponentManager;
import org.apache.avalon.framework.component.Composable;
import org.apache.avalon.framework.logger.AbstractLoggable;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.environment.SourceResolver;
import org.apache.cocoon.xml.XMLConsumer;
import org.apache.cocoon.xml.XMLProducer;
import org.xml.sax.SAXException;
import java.io.IOException ;
import java.util.Map;

public abstract class ComposerGenerator
extends AbstractGenerator
implements XMLProducer, Composable, Recyclable {

    protected ComponentManager manager; // The component manager
    protected SourceResolver resolver;  // The source resolver
    protected Map objectModel;         // The object model
    protected Parameters parameters;    // The sitemap parameters
    protected String source;           // The source information
    protected XMLConsumer xmlConsumer;  // The SAX consumer

    /** Store the component manager into manager */
    public void compose(ComponentManager manager) throws
     ComponentException;

    /** Store the parameters in resolver, objectModel, source
       and parameters */
    public void setup(SourceResolver resolver, Map objectModel,
                   String src, Parameters par);
```

```
    /** Store the consumer in xmlConsumer */
    public void setConsumer(XMLConsumer consumer);

    /** Reset instance variables */
    public void recycle();
}
```

The `ComposerGenerator` implements all the methods required for a generator except the `generate` method. All the important information is stored in instance variables: the component manager, the source resolver, the object model, the source information from the sitemap, the sitemap parameters, and, last but not least, the XML consumer.

The `ComposerGenerator` implements the Avalon `Loggable`, `Composable`, and `Recyclable` interfaces. This means that you get the logger for debugging, the generator is pooled, and the `recycle` method is called to allow the generator to clean up.

Because the zip generator is very similar to the zip reader, we will not explain it line by line. We will focus on the differences. Listing 9.7 shows the complete code for the zip generator.

Listing 9.7 The *ZipGenerator* Class

```
package cxa.generation;

import org.apache.avalon.excalibur.pool.Poolable;
import org.apache.avalon.framework.component.ComponentException;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.ResourceNotFoundException;
import org.apache.cocoon.components.parser.Parser;
import org.apache.cocoon.environment.Source;
import org.apache.cocoon.generation.ComposerGenerator;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

public class ZipGenerator
  extends ComposerGenerator
  implements Poolable {

    public void generate()
    throws SAXException, IOException, ProcessingException {
        int position = this.source.indexOf(".zip");

        String zipURI = this.source.substring(0, position + 4);
        String documentName = this.source.substring(position + 5);

        Source archive = null;
```

```
    try {
        archive = this.resolver.resolve( zipURI );

        ZipInputStream zipStream = new
  ZipInputStream( archive.getInputStream() );
        ZipEntry document = null;
        boolean found = false;
        do {
           document = zipStream.getNextEntry();
           if (document != null) {
              if ( document.getName().equals(documentName) ) {
                found = true;
             } else {
                // go to next entry
                zipStream.closeEntry();
             }
           }
      } while (document != null && found == false);

      if (document == null)
         throw new ResourceNotFoundException(documentName + " not
  found.");
      // now we will extract the document and write it into a byte array
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      byte[] buffer = new byte[8192];
      int length = -1;

      while ( zipStream.available() > 0 ) {
         length = zipStream.read(buffer, 0, 8192);
         if ( length > 0 ) {
            baos.write(buffer, 0, length);
         }
      }
      zipStream.close();
      baos.flush();

      // now we create the InputSource object for the parser
      ByteArrayInputStream is = new
  ByteArrayInputStream( baos.toByteArray() );
      Parser parser = null;
      try {
         InputSource source = new InputSource(is);

         parser = (Parser) this.manager.lookup (Parser.ROLE);
         parser.setConsumer( this.xmlConsumer );
         parser.parse( source );
      } catch (ComponentException ce) {
         throw new ProcessingException("Component Parser not found.",
  ce);
      } finally {
         this.manager.release( parser );
         is.close();
      }
  } finally {
      if (archive != null) archive.recycle();
  }
}
```

```
}
```

Can you spot the differences between the reader and the generator? The implementation of the `generate` method starts with the same lines until the entry for the document is found in the archive. Whereas the reader reads the document and then streams it to the output stream, the generator does not know anything about an output stream.

Instead, the generator writes the extracted document to a `ByteArrayOutputStream` in order to get a byte array from the document. This byte array is then fed into a `ByteArrayInputStream`. This stream can then be used for the XML parser. So the last part of the generator implementation looks up the parser component and creates an `InputSource` object from the input stream. The parser parses this input source and sends the SAX events to the XML consumer, which is the next component in the XML processing pipeline.

You know that using a `ByteArrayOutputStream` and then a `ByteArrayInputStream` is not the fastest way to do this, nor is it as elegant as it could be. However, this chapter is not about writing high-performance, state-of-the-art code. It's about writing easy-to-understand components for Cocoon. So we lay the focus on Cocoon and not on Java. However, you can use this part of the generator for your own experiments and modify the code.

Anyway, that is all you have to write for your generator. Again, the implementation is very simple and straightforward. So let's deploy the component and add the following line to the sitemap's `map:generators` section:

```
<map:generator name="zip" src="cxa.generation.ZipGenerator"/>
```

In order to test the generator, you have to put an archive into Cocoon's context directory, which contains an XML document. The following pipeline assumes that this archive is named documents.zip and that it contains a document called a.xml. If you have other names, you should change the sample pipeline:

```
<map:pipeline>
    <map:match pattern="zipgenerate">
        <map:generate type="zip" src="documents.zip/a.xml"/>
        <map:serialize type="xml"/>
    </map:match>
</map:pipeline>
```

Start Cocoon and go to `http://localhost:8080/cocoon/zipgenerate`. You will see the XML document in your browser. Again, we suggest that you now experiment a bit with the generator and extend it in additional ways in order to get a better idea of how to write generators. If you are tired of generators and want to learn more, we will now discuss the most complex sitemap component: the transformer.

Transformers

The component you will probably end up writing the most is a transformer. A transformer is an XML `pipe` object: It receives SAX events from a previous component in the pipeline and also sends SAX events to the following component.

The usual behavior of a transformer is that it triggers on some special XML elements. For example, the sql transformer listens for a query element in order to execute a SQL command. It's common for each transformer to use its own namespace to avoid conflicts. This means that a transformer listens for SAX events that contain its namespace. All other events are forwarded unchanged to the next component in the pipeline.

Whenever a transformer receives an element it understands, this element and its XML subtree are consumed by the transformer. It usually records the information contained in the subtree, processes the information, and then sends new SAX events containing the result of the process to the next component in line. For example, the sql transformer waits for the `query` element and records the SQL statement to be processed. When the transformer has all the relevant information, the SQL command is executed, the result is transformed to XML, and the correct SAX events are generated.

During the next few pages, you will develop your mail transformer. This transformer can collect the necessary information it needs to send an email from the SAX events. When the transformer has all the information, such as the subject, the receiver, and the body of the email, it sends the email. After the email is sent, the transformer creates some SAX events indicating either the success or failure of the sending process.

Let's start by looking at the `Transformer` interface that a custom transformer has to implement. Listing 9.8 contains the interface. It extends the `XMLPipe` interface. Therefore, a transformer is an XML producer like a generator and an XML consumer implementing the `ContentHandler` and the `LexicalHandler` interface. In addition, a transformer, like a reader, is a `SitemapModelComponent`. So the `setup` method is called by the processor to initialize the transformer for the current request.

Listing 9.8 The *Transformer* Interface

```
package org.apache.cocoon.transformation;

import org.apache.avalon.framework.parameters.Parameters;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.environment.SourceResolver;
import org.xml.sax.SAXException;
import java.io.IOException;
import java.util.Map;

public interface Transformer
    extends org.apache.cocoon.xml.XMLPipe,
    extends org.apache.cocoon.sitemap.SitemapModelComponent {
        void setup(SourceResolver resolver, Map objectModel,
                   String src, Parameters par)
```

```
        throws ProcessingException, SAXException, IOException;
    }

    String ROLE = "org.apache.cocoon.transformation.Transformer";
}
```

As with all the other components, Cocoon provides an abstract class—the `AbstractTransformer` class. Listing 9.9 shows this class and its methods. It inherits from the `AbstractXMLPipe` class, which implements all the methods of the `XMLPipe` interface and also implements the `Loggable` interface. In addition, an XML pipe is `Recyclable`.

Listing 9.9 The *AbstractTransformer* Class

```
package org.apache.cocoon.transformation;

import org.apache.cocoon.xml.AbstractXMLPipe;

public abstract class AbstractTransformer
extends AbstractXMLPipe
implements Transformer {
}
```

Because an XML pipe is a producer, it needs an XML consumer. This is stored in the instance variable `xmlConsumer`, which contains the next pipeline component. The XML pipe is also an XML consumer, so it must implement the content handler and lexical handler interface. The implementation of `AbstractXMLPipeline` forwards all received SAX events to `xmlConsumer`.

The only method not implemented by the `AbstractTransformer` class is the `setup` method. So you have to implement this yourself. But before we go into the details of implementing the mail transformer, let's look at an example of how to use it. The following XML sends an email when it is processed by the mail transformer:

```
<?xml version="1.0"?>
<document xmlns:mail="http://cxa/cocoon/sendmail">
    <mail:sendmail>
       <mail:mailto>myemail@mywebaddress.com</mail:mailto>
       <mail:mailsubject>Test Email</mail:mailsubject>
       <mail:mailbody>
          This is a test email.
       </mail:mailbody>
    </mail:sendmail>
</document>
```

The transformer listens for elements having the namespace `http://cxa/cocoon/sendmail`. The sending of an email is indicated by the element `sendmail`. The required information for an email—the receiver, the subject, and the body—are given by the three XML elements `mailto`, `mailsubject`, and

`mailbody`. All three elements can contain arbitrary text that provides the required information.

Your mail transformer collects this information. It starts collecting when it receives the `startElement` event for the `sendmail` element. When it receives the `startElement` event for the `mailto` object, it records all the incoming character events to get the recipient until the `endElement` event for `mailto` arrives. The same applies for the `mailsubject` and `mailbody` elements. When the transformer receives the `endElement` event for the `sendmail` element, it sends the email with the provided information.

From this description, you see that you have to keep status information inside the transformer. Using this status information, the transformer knows what information it is currently receiving: the subject, the receiver, the body, or any information that is outside the `sendmail` element. So let's start your implementation. Listing 9.10 contains the initialization of the transformer.

Listing 9.10 The Initialization of *SendMailTransformer*

```
package cxa.transformation;

public class SendMailTransformer
extends AbstractTransformer
implements Parameterizable, Poolable {

    public static final String NAMESPACE = "http://cxa/sendmail";
    public static final String SENDMAIL_ELEMENT = "sendmail";
    public static final String MAILTO_ELEMENT = "mailto";
    public static final String MAILSUBJECT_ELEMENT = "mailsubject";
    public static final String MAILBODY_ELEMENT = "mailbody";

    protected static final int MODE_NONE = 0;
    protected static final int MODE_TO = 1;
    protected static final int MODE_SUBJECT = 2;
    protected static final int MODE_BODY = 3;

    protected int mode;
    protected StringBuffer toAddress;
    protected StringBuffer subject;
    protected StringBuffer body;

    protected String mailHost;
    protected String fromAddress;

    public void parameterize(Parameters parameters)
    throws ParameterException {
        this.mailHost = parameters.getParameter("mailhost");
        this.fromAddress = parameters.getParameter("from");
    }

    public void setup(SourceResolver resolver,
                      Map          objectModel,
```

```
                String        src,
                Parameters     par)
    throws ProcessingException, SAXException, IOException {
        this.mode = MODE_NONE;
        this.toAddress = new StringBuffer();
        this.subject = new StringBuffer();
        this.body = new StringBuffer();
    }

}
```

The `SendMailTransformer` class inherits from the `AbstractTransformer` class. It also implements the `Parameterizable` interface, because the transformer needs some configuration details. In the `parameterize` method, the transformer gets the mail host and the from address used to send emails.

Because the transformer listens for elements with a distinct namespace, you define a constant called `NAMESPACE`. You also define a constant for each element name the transformer is interested in. As we discussed, your transformer needs to keep a status of which information it is currently in the process of receiving. So you define constants for each possible status and an instance variable called `mode` that contains this status.

During the `setup` method, you set this status to "outside the `sendmail` element." In addition, you have three instance variables for the mail receiver, the subject, and the body. All these are initialized during the `setup` method.

The remaining portion of the transformer is the code that receives the `startElement`, `characters`, and `endElement` events. Let's start with the `startElement` and `characters` events. The implementation of the corresponding methods is shown in Listing 9.11.

Listing 9.11 The *startElement* and *characters* Events

```
public void startElement(String uri, String name, String raw,
                     Attributes attr)
throws SAXException {

    if (uri != null && uri.equals(NAMESPACE) ) {
        if (name.equals(SENDMAIL_ELEMENT) == true) {
            // No need to do anything here
        } else if (name.equals(MAILTO_ELEMENT) == true) {
            this.mode = MODE_TO;
        } else if (name.equals(MAILSUBJECT_ELEMENT) == true) {
            this.mode = MODE_SUBJECT;
        } else if (name.equals(MAILBODY_ELEMENT) == true) {
            this.mode = MODE_BODY;
        } else {
            throw new SAXException("Unknown element " + name);
        }
    } else {
        // Not for us
```

```
        super.startElement(uri, name, raw, attr);
    }

}

public void characters(char[] buffer, int start, int length)
throws SAXException {
    switch (this.mode) {
        case MODE_NONE : super.characters(buffer, start, length);
                         break;
        case MODE_TO : this.toAddress.append(buffer, start, length);
                       break;
        case MODE_SUBJECT : this.subject.append(buffer, start, length);
                       break;
        case MODE_BODY : this.body.append(buffer, start, length);
                       break;
    }
}
```

In the `startElement` event method, you test for the namespace of the mail transformer. If the current element has the correct namespace, you test for one of the four possible elements and set the `mode` instance variable according to this information. If this element does not have the correct namespace, you simply call the super implementation of the `startElement` event method. This sends the SAX event to the next component in the pipeline.

The `characters` event method is called for all text information. If the transformer is currently in the mode of recording information, the incoming text is appended to either the subject, the body, or the receiver. Otherwise, the SAX event is sent to the next component in the pipeline.

The last remaining task is to implement the `endElement` method. The implementation is shown in Listing 9.12. This method's layout is very similar to the `startElement` event method: It tests for the namespace of the transformer and then for one of the four possible elements. If the element is either the `subject`, `body`, or `receiver` element, the transformer's mode is reset.

If the element is the `sendmail` element, the transformer starts its real work by sending the email. The following code uses the JavaMail API provided by Sun to send an email. Please refer to the JavaMail API documentation for more information.

Listing 9.12 The *endElement* Events

```
public void endElement(String uri, String name, String raw)
throws SAXException {
    if (uri != null && uri.equals(NAMESPACE) ) {
        if (name.equals(SENDMAIL_ELEMENT) == true) {
            String text;
            try {
                Properties props = new Properties();
                props.put("mail.smtp.host", this.mailHost);
```

```
            Session mailSession = Session.getInstance(props, null);

            MimeMessage pm = new MimeMessage(mailSession);

            // set from
            pm.setFrom(new InternetAddress( this.fromAddress ));
            // set to
            pm.setRecipients(Message.RecipientType.TO,

    InternetAddress.parse( this.toAddress.toString() ));
            // set subject
            pm.setSubject( this.subject.toString() );
            // set date
            pm.setSentDate(new Date());
            // set content
            pm.setText( this.body.toString() );
            // send mail
            Transport trans = mailSession.getTransport("smtp");
            Transport.send(pm);
            // success message
            text = "Sending mail to " + this.toAddress + " was successful.";
         } catch (Exception any) {
            this.getLogger().error("Exception during sending of mail",
    any);
            // failure message
            text = "Sending mail to " + this.toAddress + " failed!";
         }
         // create SAX events for success/failure
         super.startElement(NAMESPACE, "sendmail", "sendmail", new
    AttributesImpl());
         super.characters(text.toCharArray(), 0, text.length());
         super.endElement(NAMESPACE, "sendmail", "sendmail");

      } else if (name.equals(MAILTO_ELEMENT) == true) {
         // mailto received
         this.mode = MODE_NONE;
      } else if (name.equals(MAILSUBJECT_ELEMENT) == true) {
         this.mode = MODE_NONE;
      } else if (name.equals(MAILBODY_ELEMENT) == true) {
         this.mode = MODE_NONE;
      } else {
         throw new SAXException("Unknown element " + name);
      }
   } else {
      // not for us
      super.endElement(uri, name, raw);
   }
}
```

After the email is sent, the transformer creates some new SAX events that are sent to the next component in the pipeline. These events create a `sendmail` element. It has either a message that the sending of the email was successful as content or a failure message in case an error happened.

So, in fact, the mail transformer consumes the elements it understands and replaces them with a result. As you can tell from the number of lines required to implement the transformer, it is simple and straightforward to write a transformer that acts this way.

Now, let's use the transformer. Before you can compile the transformer, you need the JavaMail API and the JavaBean Activation Framework. Both are available as downloads from Sun or might already be available, depending on which servlet engine you use. Refer to Appendix C, "Links on the Web," for the URI where you can download these components. Also refer to your servlet engine's documentation for additional information. Place the two JAR files from the two packages into your classpath/IDE, and then compile the transformer. Add the following lines to the `map:transformers` section of the sitemap and change the configuration settings according to your environment:

```
<map:transformer name="mail"
     src="cxa.transformation.SendMailTransformer">
    <parameter name="mailhost" value="mymailhost"/>
    <parameter name="from" value="from@myhost.com"/>
</map:transformer>
```

Restart your servlet engine, and add a pipeline to the sitemap that contains the mail transformer. Use the file generator to read an XML document similar to this one:

```
<?xml version="1.0"?>
<document xmlns:mail="http://cxa/sendmail">
    <mail:sendmail>
        <mail:mailto>myemail@mywebaddress.com</mail:mailto>
        <mail:mailsubject>Test Email</mail:mailsubject>
        <mail:mailbody>
            This is a test email.
        </mail:mailbody>
    </mail:sendmail>
</document>
```

When you use the xml serializer, the resulting document should look like this (perhaps with a different message):

```
<?xml version="1.0"?>
<document xmlns:mail="http://cxa/sendmail">
    <mail:sendmail>
        Sending mail to myemail@mywebaddress.com was successful.
    </mail:sendmail>
</document>
```

Experiment with the transformer. Try adding new elements interpreted by the transformer, and perhaps send more new events after the email is sent or when an error occurs. Because transformers are one type of component that is often needed, we suggest that you perhaps write further transformers that react to different tags and perform other tasks. How about a transformer that receives mail for a particular user from a mail server? After

having enough of transformers, you might then want to find out about other Cocoon components. Perhaps a selector is more the component you need, so let's move on.

Selectors

Implementing a transformer is the hardest development task when it comes to building sitemap components, so if you are still with us, be assured that things will be getting easier from now on. Next in line is the selector. The interface is shown in Listing 9.13.

Listing 9.13 The *Selector* Interface

```
package org.apache.cocoon.selection;

import org.apache.avalon.framework.component.Component;
import org.apache.avalon.framework.parameters.Parameters;

import java.util.Map;

public interface Selector extends Component {

    String ROLE = "org.apache.cocoon.selection.Selector";

    boolean select (String expression, Map objectModel, Parameters
     parameters);
}
```

As the listing shows, a selector is just as simple as an action. It consists of exactly one method that must be implemented: the `select` method. Therefore, a selector is, in most cases, thread-safe and should extend the `ThreadSafe` interface.

When the `select` method is called, it gets three arguments: the current object model, the expression from the `map:when` clause, and possibly some parameters specified for the selector. The methods simply return `true` if the test is successful; otherwise, `false` must be returned. For each `map:when` in the sitemap, the `select` method is called, until one invocation returns `true`. If all calls return `false`, the `map:otherwise` branch is evaluated. So the `select` method might be called more than once for one `map:select` statement evaluation in the sitemap.

The question now is what information you want your new selector to select. We thought a season selector, which selects based on the current season of the year, might be an interesting addition. This information/selection can then be used to display the documents with a winter or summer look and feel. Here is the usage of the selector:

```
<map:generate src="document.xml">
<map:select type="season">
    <map:when test="winter">
        <map:transform src="doc2html_wintertime.xsl"/>
    </map:when>
    <map:when test="summer">
```

```
      <map:transform src="doc2html_summertime.xsl"/>
   </map:when>
   <map:otherwise>
      <map:transform src="doc2html.xsl"/>
   </map:otherwise>
</map:select>
<map:serialize/>
```

Your selector tests for five identifiers: `summer`, `winter`, `spring`, `fall`, and `autumn`. When this information is passed into the `select` method, the current date is checked to see if it is in this season. The result of this test is returned. The complete code is shown in Listing 9.14.

Listing 9.14 The *SeasonSelector* Class

```
package cxa.selection;

import org.apache.avalon.framework.logger.AbstractLoggable;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.avalon.framework.thread.ThreadSafe;
import org.apache.cocoon.selection.Selector;
import java.util.Calendar;
import java.util.Map;

public class SeasonSelector
  extends AbstractLoggable
  implements Selector, ThreadSafe {

    public boolean select(String expression, Map objectModel, Parameters
     parameters) {
       boolean result;
       // get the current month
       Calendar currentDate = Calendar.getInstance();
       int month = currentDate.get(Calendar.MONTH);

       // compare
       if ( "summer".equalsIgnoreCase(expression) ) {
          result = (month >= Calendar.JUNE && month <= Calendar.AUGUST);
       } else if ("autumn".equalsIgnoreCase(expression)
                 || "fall".equalsIgnoreCase(expression) ) {
          result = (month >= Calendar.SEPTEMBER && month <=
     Calendar.NOVEMBER);
       } else if ("winter".equalsIgnoreCase(expression) ) {
          result = (month == Calendar.DECEMBER || month <=
     Calendar.FEBRUARY);
       } else if ("spring".equalsIgnoreCase(expression) ) {
          result = (month >= Calendar.MARCH && month <= Calendar.MAY);
       } else {
          // unknown expression
          result = false;
       }

       return result;
    }
}
```

The season selector does not perform 100% correctly when testing for the season. It checks only for the month, not for the date. For example, June, July, and August are considered summer. Now let's configure your new selector in the sitemap inside the `map:selectors` section:

```
<map:selector name="season" src="cxa.selection.SeasonSelector"/>
```

You can test the selector with a pipeline like the one we showed at the introduction to selector components. You will get your document layout for the current season. However, if you want to test the other seasons, you might have to wait a few months. In the meantime, let's develop some more components for Cocoon.

Isn't it simple to write a selector? We suggest that you now try to write your own selector that selects based on the current time of day so that your document can use darker colors at night and brighter ones during the day. With the selector component, we complete our developer's tour through the sitemap components. We will now get on with some advanced topics.

Advanced Components

Congratulations! You have written your own first components and extended Cocoon by adding your own sitemap components. But believe us: It gets even more exciting. In the previous examples, you developed some components that read documents from an archive: the zip reader and the zip generator. They are both very helpful components, but they have at least two disadvantages: They do not support the Cocoon caching algorithm, and it is not possible for other components such as transformers to also access an archive.

We will first show you how you can make the sitemap components cache-aware. Afterwards, you will develop your own `zip` protocol. With this protocol you can access documents from within an archive everywhere a URI is allowed. But first, here are some notes about caching.

Making Components Cacheable

We explained the basic caching algorithm in Chapter 6. We will now only repeat the important pieces of information you need to implement cacheable sitemap components. The caching algorithm builds a unique key for each request. Using this key, the algorithm can detect whether the response has already been cached. If it was cached before, the key points to a collection of `validity` objects and the cached response itself. The `validity` objects are compared with the newly generated `validity` objects to test whether the cached response is still valid.

The caching algorithm needs two pieces of information: a unique key and a `validity` object. These two pieces of information are provided by each sitemap component that supports caching. In particular, this means either a reader, a generator, a transformer, or a

serializer. If such a component is cache-aware, it implements the `Cacheable` interface, as shown in Listing 9.15.

Listing 9.15 The *Cacheable* Interface

```
package org.apache.cocoon.caching;

public interface Cacheable {

    long generateKey();

    CacheValidity generateValidity();
}
```

This interface consists of two methods: `generateKey`, which provides a unique key for the current instance of the component, and `generateValidity`, which returns a `validity` object that can be used to test whether the cached content is still valid.

To explain these two methods, let's have a look at a real-world example of a cacheable component: the file generator. The key generated by the generator is required to be unique within the space of this component type. This means that the file generator must provide different keys if it reads different source documents. Different types of generators can generate the same key without any conflict. The caching algorithm automatically distinguishes between keys from a file generator and, for example, keys from an html generator.

The component should build the key by assembling all the information required for the current usage of this component in the pipeline. In other words, the key must be built using all the input information for this component. In the case of the file generator, this is the URI of the XML document. So the file generator builds its key from the URI by hashing the string.

However, the html generator does not have only the URI as input. It also has two parameters. Therefore, the html generator hashes not only the URI but also the values of these two parameters.

The second piece of information required for the caching algorithm is a `CacheValidity` object. The interface is shown in Listing 9.16. The interface consists of only one method: `isValid`. As just explained, the caching algorithm stores the `validity` object together with the key. When a cached document is requested the next time, the sitemap components producing the document provide new `validity` objects for the request. These objects are then compared to the cached versions.

Listing 9.16 The *CacheValidity* Interface

```
package org.apache.cocoon.caching;

public interface CacheValidity extends java.io.Serializable {
```

```
    boolean isValid(CacheValidity validity);
}
```

In the case of the file generator, this means that the generator stores the last modification date of the document in the cache `validity` object. So the cached document always contains a `validity` object with the last modification date of the read XML document. The next time the same document is requested, the file generator provides a new cache `validity` object with the current last modification date of the XML document. The two `validity` objects are then compared. If the last modification date is the same, the cached response is valid.

Cocoon already contains some implementations of the `CacheValidity` interface:

- `AggregatedCacheValidity`: This can be used if several `validity` objects must be combined to check the validity.
- `CompositeCacheValidity`: This is the same as `AggregatedCacheValidity`, except that only two cache `validity` objects can be combined.
- `NOPCacheValidity`: This `validity` object is always valid. It can be used whenever an up-to-date check is not required. This is the case if the component does not change the content of the generated document, like most serializers do.
- `TimeStampCacheValidity`: This `validity` object compares timestamps. It's used the most often, because it is used by many components to hold the last modification date of a document, such as the read XML document or the used XSLT stylesheet.

For many components, the implementation of the `Cacheable` interface is very similar. The generated cache key is produced by hashing a URI, and the cache validity is built by getting the last modification date of that URI.

The caching algorithm can work only if the sitemap components know about their input before the caching algorithm calls the `generateKey` or `generateValidity` method. The first method called by the caching algorithm is always the `setup` method of the `SitemapModelComponent` interface. With this method, the component gets all the information it requires to build the key.

The next method invoked is the `generateKey` method. It might be possible that a component normally is cacheable, but for some reason it isn't at this particular time. For example, perhaps it can't get all the information required to generate the key or the `validity` object. In this case, the component should return 0 for the key. This indicates to the caching algorithm that the request currently isn't cacheable.

However, if the key is not 0, the caching algorithm calls the `generateValidity` method next. Again, if the request currently isn't cacheable, the component has a chance to return `null`, indicating this.

If the response is cached and this cached response is valid, the sitemap component is not used anymore; it is put back into the pool without the invocation of any other method. If the response is not cached or is invalid, the Cocoon processor calls the component's methods as explained throughout this chapter.

So, each component taking part in the document's processing can support the caching by looking at its input. This simple statement needs some further investigation. What is a component's input?

For a `reader` component, this question is easy to answer. A reader's input is the source information from the `map:read` statement and possibly the parameters. This information is passed to the component during the setup phase, so the reader can easily generate a key and a `validity` object from it.

For a serializer, the process is a bit more complicated. On the one hand, a serializer is not allowed to get source information or any parameters from the sitemap. So there is no input to a serializer from this side. But the serializer is an XML consumer, so it gets XML or SAX events as input. Because the methods of the `Cacheable` interface are called after the component is set up but long before the XML processing is started, the serializer does not know anything about its input.

This is obviously a required feature. The caching algorithm can be effective only if the XML processing does not need to take place in order to check whether a cached document is valid. If it would require that the document be generated from scratch only to test the validity, it wouldn't make sense to take the document from the cache. There wouldn't be any performance improvements, making the cache absolutely useless.

The answer to this problem is very simple. The SAX events coming into the serializer are not assumed to be input when it comes to caching. Imagine a pipeline with a generator, a transformer, and a serializer. The generator provides a starting point for the XML processing and sends the starting SAX events to the transformer. Each time the generator reads or produces the same XML document, the SAX events sent to the transformer are the same. Because the transformer always receives the same SAX events and does the same transformations each time, the transformer sends the same set of SAX events to the serializer each time the pipeline is processed.

In other words, if the previous stages of the pipeline produce the same XML, the serializer gets the same SAX events as input, and therefore the serialized result is the same.

This is asserted by the caching algorithm as it first asks the generator for a unique key and a `validity` object. Only if a response is cached with this key and only if it is valid is the next component in the XML pipeline asked. If the key isn't found or if the validity check isn't successful, the remaining components in the pipeline are never asked for their keys or `validity` objects.

Let's assume that the generator provides a cached key and a valid `validity` object. The transformer is then asked for its key and `validity` object. Only if this key is found and the validity check is successful does the caching algorithm know that the same XML stream comes from this transformer. And only then can the last component in the pipeline, the serializer, be queried.

Cocoon's caching algorithm is very smart. It can use the provided keys and `validity` objects to check whether the XML stream between the components will be the same as during the cached invocation. This leads us to the statement that the input of a sitemap component is any information apart from the SAX events.

Now let's come back to the question of what the input for a serializer is. The answer is simple: The serializer has no input (other than SAX events). So, 99.9% of the time, a serializer is cacheable, and it can return a constant for the unique key and a `NOPCacheValidity` object as the `validity` object. If the serializer uses some external information, such as the current date and time, to produce the output, it is not so easy to determine the cacheability.

In the first run, this algorithm seems very clever, but the world isn't that perfect. So there are some drawbacks to the caching algorithm. We stated that if the incoming SAX events for a transformer are the same during two requests, the output SAX events are also the same.

Unfortunately, this is not always true. For example, if you use the sql transformer, the incoming SAX events contain a database query. If during the two requests the content of the database changes, the output SAX events of the sql transformer will differ even though the received SAX events were the same. Because during the processing of the caching algorithm the sql transformer does not know anything about the SQL commands it will receive, it can calculate neither a unique key nor a `validity` object. So the conclusion here is to not make the sql transformer cacheable. But this prevents a pipeline containing the transformer from being cacheable. In such a case, we say that the input of the transformer is not only the information passed on from the sitemap but also any external information, such as the fetched SQL data.

When making a sitemap component cacheable, you have to think about what your component's input really is and what it must use to determine whether it is cacheable. Let's now make your reader and generator cacheable.

A Cacheable Reader

We will start with the reader. You make a component cacheable simply by implementing the `Cacheable` interface. You must figure out what the input of your reader is. The answer is very simple: It is the URI containing the archive and the document to be extracted.

You will build a unique key based on this information by simply hashing the URI. Cocoon provides a powerful hashing algorithm in the `HashUtil` class. You will use this algorithm to hash the URI.

The second point for caching is the `validity` object. When is the cached content invalid? When the extracted document changes. How do you find out whether this is the case? First you could ask the archive about the last modification date of the extracted document. But because the archive might be very large, asking for this information could take a long time, so you use a simpler approach:You just check the archive's last modification date. If the document inside the archive changes, the archive itself changes.

Unfortunately, the opposite is not always true. If the archive changes, the document in question might not have been changed. But we will neglect this point here.

So, to summarize what we have discussed so far, your cacheable reader generates a hash of the URI used to extract the document, and it uses `TimeStampCacheValidity` with the archive's last modification date.

There are at least three places where the reader needs a `Source` object: to calculate the hash, to get the last modification date, and to read from the archive. Resolving via the `Source` object might be a time-consuming task, because a connection to a different server is established each time to get the required information. You should avoid doing this three times for only one read process. In order to achieve this, you use the reader's `setup` method to get a `Source` object for the archive. This object is used throughout the whole read process. But where can you recycle this object? This is exactly one of the purposes of the Avalon `Recyclable` interface: The component can clean up. So you use the `recycle` method to recycle the `source` object for the archive. Listing 9.17 contains the complete cacheable reader.

Listing 9.17 The *CacheableZipReader*

```
package cxa.reading;

public class CacheableZipReader
  extends AbstractReader
  implements Recyclable, Cacheable {

    /** The source for the archive */
    protected Source archive;

    /** The document name */
    protected String documentName;

  public void setup() {
       super.setup(resolver, objectModel, src, parameters);

       int position = this.source.indexOf(".zip");
       if (position == -1) {
```

```
            throw new ProcessingException("The URI does not point to a ZIP
     archive.");
          }
          String zipURI = this.source.substring(0, position + 4);
          this.documentName = this.source.substring(position + 5);

          if ( this.getLogger().isDebugEnabled() ) {
              this.getLogger().debug("Reading " + documentName + " from zip
     archive " +
zipURI);
          }
          this.archive = resolver.resolve(zipURI );
       }
      /**
       * Recycle
       */
      public void recycle() {
         if (this.archive != null) {
            this.archive.recycle();
            this.archive = null;
       }
      }

      /**
       * Generate the unique key.
       * This key must be unique inside the space of this component.
       * This method must be invoked before the generateValidity() method.
       *
       * @return The generated key or <code>0</code> if the component
       *             is currently not cacheable.
       */
      public long generateKey() {
         if (this.archive.getLastModified() != 0) {
            return HashUtil.hash(this.archive.getSystemId() + '/' +
        this.documentName);
          }
         return 0;
      }

      /**
       * Generate the validity object.
       * Before this method can be invoked the generateKey() method
       * must be invoked.
       *
       * @return The generated validity object or <code>null</code> if the
       *         component is currently not cacheable.
       */
      public CacheValidity generateValidity() {
         if (this.archive.getLastModified() != 0) {
            return new
        TimeStampCacheValidity(this.archive.getLastModified());
          }
         return null;
      }

      /**
       * Generates the requested resource.
```

```
    */
   public void generate()
   throws SAXException, IOException, ProcessingException {
      ZipInputStream zipStream = new
     ZipInputStream( archive.getInputStream() );
      ZipEntry document = null;
      boolean found = false;
      do {
        document = zipStream.getNextEntry();
        if (document != null) {
            if ( document.getName().equals(documentName) ) {
               found = true;
            } else {
               // go to next entry
               zipStream.closeEntry();
            }
        }
     } while (document != null && found == false);

     if (document == null)
        throw new ResourceNotFoundException("The document " + documentName
     + " is not in
the archive " + zipURI);

     byte[] buffer = new byte[8192];
     int length = -1;

    while ( zipStream.available() > 0 ) {
        length = zipStream.read(buffer, 0, 8192);
        if ( length > 0 ) {
           this.out.write(buffer, 0, length);
        }
     }
     zipStream.close();
     this.out.flush();

  }

}
```

As shown in Listing 9.17, the `setup` method parses the URI and creates a `source` object for the archive. This is later cleaned up by the `recycle` method. You might wonder why you don't recycle the `source` object in the `generate` method. The reason is simple: It is not guaranteed that the `generate` method will be called. First, your reader is cacheable, so in the case of a cached document, the `setup` method is called, but not the `generate` method. Second, if an error or exception occurs during the processing, it might be that the `generate` method is not called due to the exception. But because the `recycle` method belongs to the component life-cycle methods, it is called in all circumstances.

The implementation of the `generate` method is very similar to the one used for the noncacheable version. The two methods of the `Cacheable` interface are also very straightforward. The `generateKey` method hashes the complete URI containing the

archive and the document name. The `generateValidity` method uses the last modification date of the `source` object to initialize a `TimeStampCacheValidity` instance.

You can see from this cacheable reader that it is very easy to write a cache-aware component for Cocoon, because only two methods must be implemented. For performance reasons, you should make your component cacheable whenever possible.

A Cacheable Generator

Making your `zip generator` cacheable is very similar to making the reader cacheable. You implement the `Cacheable` interface and the methods. Again, the generator must generate a unique key by hashing the URI, and the `validity` object contains the archive's last modification date. So actually, the generator uses exactly the same implementation as the reader.

We do not provide the solution for the generator in this book because we think this is a good practical lesson for you. Just look at the cacheable reader again and try to make the zip generator cacheable by changing the implementation in exactly the same way you changed the reader. You will see that making a component cacheable is not that difficult, provided you can work out what the input is.

When you have finished your implementation of the cacheable zip generator and it seems to be working correctly, check out the companion CD. It contains the complete source code of the cacheable generator as a reference.

You have just developed some high-performance cache-aware sitemap components. But a more general approach for dealing with archives in Cocoon is to create a custom protocol.

Creating a Protocol

If you look closely at your components—the zip generator and the zip reader—you can see that they are very similar. The only real difference is that the generator uses a parser to generate SAX events. But apart from that, they are identical.

This means that every time you want to access a zip file from one of your components, you would be duplicating the code. Cocoon is flexible enough to offer a better solution:You can create your own protocol. Instead of using custom components whenever you want to extract something from an archive, you can use this new protocol with each existing sitemap component.

After you have developed a suitable protocol, you can write `zip://people/matthew.jpg@pictures.zip` or `zip://financenews.xml@http://www.newserver.com/news_2001.zip`. This protocol extracts the URI for the archive, accesses the archive, and fetches the document

to be extracted. The URI format you use is `zip://`, followed by the path inside the archive, followed by `@`, and ending with the URI for accessing the archive.

By writing your own protocol, you can extract files from archives in every sitemap component. You could then use the new protocol with the file generator, the resource reader, or even the xslt transformer. Of course, this makes the two new components you developed earlier redundant. However, they are still good examples for learning how to develop sitemap components.

All source resolving takes place using the provided `SourceResolver`. This central component is passed to all sitemap components. It provides a single method, `resolve`, which has exactly one argument: a string containing the URI. The result of this method invocation is a `Source` object. For more information on source resolving, see Chapter 8.

How does this source resolver get to know about a custom protocol? The source resolver uses a `SourceHandler`. This source handler helps the source resolver when it comes to custom protocols. The source handler knows all the custom protocols, because these protocols are registered with the handler.

What do you need to do to write your own protocol? You need to write your own Avalon component that implements the `SourceFactory` protocol, shown in Listing 9.18. This factory must then be registered with the source handler. This is done in cocoon.xconf. But before you configure something, let's look at the implementation.

Listing 9.18 The *SourceFactory* Interface

```
package org.apache.cocoon.components.source;

import org.apache.avalon.framework.thread.ThreadSafe;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.environment.Environment;
import org.apache.cocoon.environment.Source;

import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;

public interface SourceFactory extends ThreadSafe {

    Source getSource(Environment environment, String location)
    throws ProcessingException, MalformedURLException, IOException;

    Source getSource(Environment environment, URL base, String location)
    throws ProcessingException, MalformedURLException, IOException;
}
```

As you can see from the listing, the source factory interface is very simple. It consists of two methods, both called `getSource`, returning a `Source` object. They differ only in the number of parameters.

The second method with three parameters gets the current environment, a base URI, and a string containing a relative URI to resolve. Actually, this is a legacy method that is usually used to resolve relative locations to a base URI. But because the base URI is a `URL` object that doesn't know anything about your `zip` protocol, this method can never really be invoked. However, because the source factory interface still contains this method, you implement it simply by calling the other `getSource` method from within this implementation.

All the work is done in the first `getSource` method with two parameters: the current environment and the URI to resolve. Your source factory parses the URI and checks to see whether it is valid. This means it must start with `zip://` and that the `@` character must be used to separate the document name from the archive location. Because the archive's URI can be any URI, even HTTP or FTP, it must be resolved using the current source resolver. The current source resolver is passed to the method by the `environment` object, because in this case the environment is also a source resolver. The whole factory is shown in Listing 9.19.

Listing 9.19 The *ZipSourceFactory* Class

```
package cxa.components.source;

import org.apache.avalon.framework.component.ComponentManager;
import org.apache.avalon.framework.component.Composable;
import org.apache.avalon.framework.logger.AbstractLoggable;
import org.apache.avalon.framework.thread.ThreadSafe;
import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.components.source.SourceFactory;
import org.apache.cocoon.components.source.URLSource;
import org.apache.cocoon.environment.Environment;
import org.apache.cocoon.environment.Source;
import org.xml.sax.SAXException;

import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;

public class ZipSourceFactory
extends AbstractLoggable
implements Composable, SourceFactory, ThreadSafe {

    protected ComponentManager manager;

    public void compose(ComponentManager manager) {
        this.manager = manager;
    }

    public Source getSource(Environment environment, String location)
    throws ProcessingException, MalformedURLException, IOException {
        int separatorPos = location.indexOf('@');
        if (separatorPos == -1)
            throw new MalformedURLException("@ required in URI: " + location);
        int protocolEnd = location.indexOf("://");
```

```
      if (protocolEnd == -1)
         throw new MalformedURLException("URI does not contain '://' : "
    + location);

      String documentName = location.substring(protocolEnd+3,
    separatorPos);
      Source archive;
      if (environment == null) {
         archive = new URLSource(new
    URL(location.substring(separatorPos+1)),
                  this.manager);
      } else {
         try {
           archive =
    environment.resolve(location.substring(separatorPos+1));
        } catch (SAXException sax) {
           throw new ProcessingException("Unable to resolve ");
        }
     }
     return new ZipSource(archive, documentName, this.manager);
  }

  public Source getSource(Environment environment, URL base, String
     location)
  throws ProcessingException, MalformedURLException, IOException {
    return this.getSource(environment, new URL(base,
    location).toExternalForm());
  }
}
```

You use the environment to resolve the archive's URI. The result is a `Source` object. This `Source` object, along with the name of the document to be extracted and the component manager, are used to create a `Source` object that can be used in every component. This `Source` object is implemented by the `ZipSource` class. We will not provide you with the whole class, but we will show some pieces of it and explain them directly. For the complete class listing, we suggest checking out the CD.

Listing 9.20 shows the constructor and the body of the class. The `ZipSource` class implements the `Source` interface. It gets the document name, the archive, and the component manager from the source factory when a new `ZipSource` instance is created. All the information is stored in separate instance variables. One method of the `Source` interface is the `getSystemId()` method, which returns the unique URI of this source. This system identifier is created in the constructor by concatenating the archive's URI and the path of the document to be extracted. The complete identifier is stored in an instance variable, too.

Listing 9.20 The *ZipSource* Class

```
package cxa.components.source;

public class ZipSource implements Source {
```

```
    private String systemId;
    private Source archive;
    private ComponentManager manager;
    private String documentName;

    public ZipSource(Source archive,
                  String documentName,
                  ComponentManager manager)
    throws IOException {
        this.manager  = manager;
        this.systemId = archive.getSystemId() + '/' + documentName;
        this.archive  = archive;
        this.documentName = documentName;
   }

   public String getSystemId() {
      return this.systemId;
   }
}
```

Listing 9.21 shows some more methods of the `Source` interface implemented by the `ZipSource` class. Because you do not know the length of the document beforehand, you simply return -1 for the content length. The object needs no cleanup after it is used, so the implementation of the `recycle` method is empty. The `getLastModified` method should return the date when the source was last changed. You return the date when the archive was last modified. Because the archive is a `source` object itself, you need to see if the `source` object is a `ModifiableSource`, which indicates whether the archive can change. If it is a modifiable source, you have to call the `refresh` method on the archive before you can ask for the last modification date. You put the test for a modifiable source and the call of the `refresh` method in a separate method called `update`, because you have to call the `refresh` method on the archive each time you do something with the archive.

Listing 9.21 Standard Methods of *ZipSource*

```
protected void update() {
    if (this.archive instanceof ModifiableSource) {
        ((ModifiableSource)this.archive).refresh();
    }
}

public long getLastModified() {
    this.update();
    return this.archive.getLastModified();
}

public long getContentLength() {
    return -1;
}

public void recycle() { }
```

Only three methods are needed to complete the `Source` interface: `getInputStream`, `getInputSource`, and `toSAX`. Let's start with the last two. They are shown in Listing 9.22. The implementation of `getInputSource` is very simple. It creates a new `InputSource` object and passes the system identifier and the input stream to this new object.

The `toSAX` method streams the content of the extracted document to a parser that creates the SAX events. This code is very similar to that used for the zip generator. The only difference is that the `toSAX` method has only a `ContentHandler` as the parameter. This method tests to see whether this object is an `XMLConsumer` or a `LexicalHandler`. The input for the parser is an `InputSource`. It is created by calling the `getInputSource` method.

Listing 9.22 XML Support for the *ZipSource* Class

```
public InputSource getInputSource()
throws IOException, ProcessingException {
    InputSource newObject = new InputSource(this.getInputStream());
    newObject.setSystemId(this.systemId);
    return newObject;
}

public void toSAX(ContentHandler handler)
throws SAXException, ProcessingException
{
    Parser parser = null;
    try {
        parser = (Parser)this.manager.lookup(Parser.ROLE);

        if (handler instanceof XMLConsumer) {
            parser.setConsumer((XMLConsumer)handler);
        } else {
            parser.setContentHandler(handler);
            if (handler instanceof LexicalHandler) {
                parser.setLexicalHandler((LexicalHandler)handler);
            }
        }
        parser.parse(this.getInputSource());
    } catch (ProcessingException e){
        // Preserve original exception
        throw e;
    } catch (SAXException e) {
        // Preserve original exception
        throw e;
    } catch (Exception e){
        throw new ProcessingException("Exception during processing.", e);
    } finally {
        if (parser != null) this.manager.release(parser);
    }
}
```

The remaining method, `getInputStream`, is shown in Listing 9.23. It first calls the `update` method to update the archive `source` object and then uses the same code you used for the zip generator to extract the document from the archive. The data of the extracted document is again read into a byte array, and a `ByteArrayInputStream` reading from this byte array is returned.

Listing 9.23 The *getInputStream* Method

```
public InputStream getInputStream()
throws IOException, ProcessingException {
    this.update();

    ZipInputStream zipStream = new
     ZipInputStream( this.archive.getInputStream() );
    ZipEntry document = null;
    boolean found = false;
    do {
        document = zipStream.getNextEntry();
        if (document != null) {
            if ( document.getName().equals(this.documentName) ) {
                found = true;
            } else {
                // go to next entry
                zipStream.closeEntry();
            }
        }
    } while (document != null && found == false);

    if (document == null)
        throw new ResourceNotFoundException("The document is not in the
     archive.");

    // now we will extract the document and write it into a byte array
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    byte[] buffer = new byte[8192];
    int length = -1;
    while ( zipStream.available() > 0 ) {
        length = zipStream.read(buffer, 0, 8192);
        if ( length > 0 ) {
            baos.write(buffer, 0, length);
        }
    }
    zipStream.close();
    baos.flush();

    // return an input stream
    return new ByteArrayInputStream( baos.toByteArray() );
}
```

That's all you need to implement. Let's review what you have done so far. You developed a `SourceFactory` that provides the new `zip` protocol to the source resolver. The new `ZipSourceFactory` creates a `ZipSource` object that implements the `Source` interface.

Now each time a URI with the zip protocol is resolved by the source resolver, it returns a `ZipSource` object. This can then be used by the calling component to create SAX events, to read from the input stream, and so on. But before you can test your component, there is one thing left to do: configuration.

You have developed a new protocol, and now you have to tell Cocoon that it can use this protocol. You do this by adding the protocol to the configuration of the `SourceHandler` in cocoon.xconf:

```
<source-handler>
   <protocol name="zip" class="cxa.components.source.ZipSourceFactory"/>
</source-handler>
```

The source handler is an Avalon component that can be configured in cocoon.xconf. The configuration consists of a list of custom protocols that Cocoon supports. With the `protocol` element, you define your new protocol. The `name` attribute defines the protocol's scheme name, and the `class` attribute points to a class that implements the `SourceFactory` protocol. That's all you have to do.

You can test the protocol by using it, for example, with the file generator to read XML documents from an archive. Or you can use it with the resource reader to read images from the archive, or use it with the xslt transformer, and so on. The following lines show the same example you used for your custom zip reader, but now with the use of the new protocol:

```
<map:pipeline>
   <map:match pattern="zipread">
      <map:read mime-type="image/jpg" src="zip://test.jpg@images.zip"/>
   </map:match>
</map:pipeline>
```

With the protocol, you added the same functionality you had beforehand with the custom reader. But the protocol can now be used everywhere. Because most Cocoon components are already cache-aware, you do not need to worry about caching with a custom protocol, because it works out of the box. For example, the file generator and the resource reader use the last modification date provided by the `Source` object for caching.

So you might well get the idea that everything should be implemented as a protocol and that writing additional generators or readers is a very rare task. A custom protocol should be preferred over a generator or reader only if the protocol can be used for more than one component type. For example, if a protocol can be used only for a generator and not for a transformer or a reader or any other component, it is better to write a custom generator/reader instead.

You have learned from this example that extending Cocoon is quite simple in most cases. It often comes down to implementing an interface and configuring something somewhere.

Now that you have added a component (or better, a protocol) to the system, we will look at adding your own custom component.

Writing a Mail Component

When we developed the `zip` protocol, we mentioned that one main advantage of creating a protocol is that it can be used everywhere a URI is allowed. In fact, a protocol is a simple example of Separation of Concerns (SoC). The zip reader and the zip generator were developed for one special purpose: to read from an archive. You soon discovered that this resulted in duplicate code and very restricted use of this feature, because it is unavailable for other components such as transformers.

The reader and the generator have at least two concerns: their main function as a reader or generator and interpreting a new URI scheme. By providing the custom protocol, you separated the concerns and minimized the effort and the components needed for this purpose.

Another example of a not-so-sophisticated software architecture is the mail transformer you developed in this chapter. This transformer also has two purposes: interpreting the tags of the SAX stream to collect the information for sending an email, and sending of the email itself. If you want to send an email from any other component, such as an action, you have to duplicate code and configuration. Remember that you need at least the mail host to send emails.

This leads us to the next example: a mail component. This is a custom component that, like the parser, is available to all the other components in Cocoon. This component's main task is sending emails. Later in this chapter you will adapt the transformer to use this mail component. By providing this centralized instance, you separate concerns and make the configuration a lot easier, because there is only one place to configure the parameters needed to send emails. In addition, you get the benefit of using this mail component in other components as well, such as in an action.

The Mail Component

Let's start with the design of the mail component. It will consist of only one method, `sendMail`. This method has three parameters: the email's receiver, subject, and body as strings. Listing 9.24 shows the interface for this component.

Listing 9.24 The Mail Component

```
package cxa.component.mail;

import org.apache.avalon.framework.component.Component;

public interface MailComponent
extends Component
{
   String ROLE = "cxa.component.mail.MailComponent";
```

```
   boolean sendMail(String receiver, String subject, String body);
}
```

Because the component has only one method, the implementation can be `ThreadSafe`. The component needs a configuration for the mail host and the from address to use for sending emails. So the component implements the `Parameterizable` interface. Listing 9.25 shows the whole implementation of the mail component.

Listing 9.25 The Mail Component Implementation

```
package cxa.component.mail;

import org.apache.avalon.framework.parameters.Parameterizable;
import org.apache.avalon.framework.parameters.Parameters;
import org.apache.avalon.framework.parameters.ParameterException;
import org.apache.avalon.framework.thread.ThreadSafe;
import java.util.Properties;

import javax.mail.*;
import javax.mail.internet.*;

public class MailComponentImpl
implements MailComponent, Parameterizable, ThreadSafe{

    protected String mailHost;
    protected String fromAddress;

    public void parameterize(Parameters parameters)
    throws ParameterException {
        this.mailHost = parameters.getParameter("mailhost");
        this.fromAddress = parameters.getParameter("from");
    }

   public boolean sendMail(String receiver, String subject, String body) {
        try {
            Properties props = new Properties();
            props.put("mail.smtp.host", this.mailHost);
            Session mailSession = Session.getInstance(props, null);

            MimeMessage pm = new MimeMessage(mailSession);

            // set from
            pm.setFrom(new InternetAddress( this.fromAddress ));
             // set to
             pm.setRecipients(Message.RecipientType.TO,
                              InternetAddress.parse( receiver ));
             // set subject
            pm.setSubject( subject );
            // set date
            pm.setSentDate(new Date());
            // set content

     pm.setText( body                                                    );
            // send mail
```

```
            Transport trans = mailSession.getTransport("smtp");
            Transport.send(pm);
            // success
            return true;
        } catch (Exception any) {
            // failure
            return false;
        }
    }
}
```

The code for the sending of the email is transferred directly from the mail transformer to the component. Neither the interface of the mail component nor the implementation is very sophisticated. For a real-world custom component, the interface is really unsuitable, because it is not flexible enough. But as soon as you know more about using custom components in Cocoon, you can turn your mail component into something more powerful. After all, the components developed in this chapter are now yours to use and adapt as needed.

Adding Roles

Now you have developed the component. But how do you tell Cocoon about it? Do you remember the introduction of roles from Chapter 8? We discussed that the configuration of a component consists of two parts. The first one makes the role available to the application, and the second one sets the used implementation for a role.

You already know that this second configuration takes place in cocoon.xconf. But where can you add new roles? Most roles that are available in Cocoon are defined in a file called cocoon.roles. This file is contained in Cocoon's JAR file. It is an XML document that looks like this:

```
<role-list>
 <role name="org.apache.cocoon.components.parser.Parser"
      shorthand="parser"
      default-class="org.apache.cocoon.components.parser.JaxpParser"/>

...more roles...
</role-list>
```

For each role, an element with the name `role` is listed in this file. The attribute `name` is the name of the role, and the `default-class` is the default implementation for this component. This can be overridden in cocoon.xconf. The attribute `shorthand` is also used in cocoon.xconf as a unique identifier to address this role:

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

All you have to do is add your role to the list. But wait a minute. The cocoon.roles file is inside a JAR file in the Cocoon distribution. Altering this file would require you to alter the JAR file and to directly manipulate the Cocoon distribution. Because this is not a viable solution, there is, of course, a better one: It is possible to tell Cocoon to look for a second file containing role definitions. You do this by adding the attribute `user-roles` to the root element of cocoon.xconf:

```
<cocoon version="2.0" user-roles="myroles.xconf">
   ...
</cocoon>
```

Now you can change the file myroles.xconf and add all your custom roles here. Let's start with a role for the mail component:

```
<role-list>
    <role name="cxa.component.mail.MailComponent"
          shorthand="mail"
          default-class="cxa.component.mail.MailComponentImpl"/>
</role-list>
```

Now you can use this role with the shorthand `mail` to configure the component in cocoon.xconf by adding these lines and changing the settings to your environment:

```
<mail>
    <parameter name="mailhost" value="mymailhost"/>
    <parameter name="from" value="from@myhost.com"/>
</mail>
```

After restarting Cocoon, you can use this mail component like any other component. You can look it up using `MailComponent.ROLE` and can use it to send emails.

Using the Mail Component

The simplest way to test the mail component is to change the mail transformer. Just add an import statement for the `cxa.component.mail` package to the transformer and change the method for the `endElement` events, as shown in Listing 9.26.

Listing 9.26 Using the Mail Component

```
public void endElement(String uri, String name, String raw)
throws SAXException {
    if (uri != null && uri.equals(NAMESPACE) ) {
        if (name.equals(SENDMAIL_ELEMENT) == true) {
            String text;
            MailComponent mail = null;
            try {
                mail = (MailComponent)
     this.manager.lookup(MailComponent.ROLE);
```

```
    ➥if (mail.sendMail(this.toAddress.toString(),
                        this.subject.toString(),
                        this.body.toString()) ) {
                    text = "Sending mail to " + this.toAddress + " was
    successful.";
                } else {
                    text = "Sending mail to " + this.toAddress + " failed!";
                }
                // create SAX events for success/failure
                super.startElement(NAMESPACE, "sendmail", "sendmail", new
AttributesImpl());
                super.characters(text.toCharArray(), 0, text.length());
                super.endElement(NAMESPACE, "sendmail", "sendmail");

            } finally {
                this.manager.release(mail);
            }
        } else if (name.equals(MAILTO_ELEMENT) == true) {
            // mailto received
            this.mode = MODE_NONE;
        } else if (name.equals(MAILSUBJECT_ELEMENT) == true) {
            this.mode = MODE_NONE;
        } else if (name.equals(MAILBODY_ELEMENT) == true) {
            this.mode = MODE_NONE;
        } else {
            throw new SAXException("Unknown element " + name);
        }
    } else {
        // not for us
        super.endElement(uri, name, raw);
    }
}
```

By using the mail component, you have created cleaner code and can change the implementation of the email component to meet new challenges.

Developing and using custom components with Cocoon is simple. It consists of these steps:

1. Defining the role by specifying a Java interface.

2. Creating a default implementation for the role.

3. Adding the role to the user roles file.

4. Configuring the role in cocoon.xconf.

5. Using the component.

This is only a checklist, not a strict set of rules. There might be components that have only one static implementation, so you do not need to create an interface for it. Instead, you can use the implementation directly as the role definition as well. There are other

cases, such as utility classes, in which you might not need a configuration or the Avalon component life-cycle, so you would not need an entry in cocoon.xconf. A good start for writing roles is to have a look at the existing roles and their implementations inside Cocoon. Or you can check the Avalon home page. It contains some notes and a tutorial for writing components.

Wrapping Up the Developer Perspective

Throughout this chapter, you have developed several components for Cocoon. You started by exploring the sitemap components. This covered actions, selectors, generators, readers, and transformers. Then you fine-tuned these components by writing cache-aware components and providing your own protocol to use with Cocoon. You finished by writing and adding your own mail component.

As you saw in this chapter, developing components for Cocoon is often very simple. In most cases, it comes down to implementing an interface and, even simpler, extending a provided abstract class.

The best way to get to know Cocoon is to look into the provided source code. Just start by exploring some simple classes such as `FileGenerator` and see how they are implemented. From there, start to explore the classes and components used by the file generator and so forth. You can learn a lot from the source code. It helps you understand the concepts behind Cocoon. You will get a better idea of when to use what and also be able to decide if you really need a custom component or if the needed component is already contained in the Cocoon distribution. Developing additional components will also hopefully lead to your becoming a member of the Cocoon developer community and donating some of the things you have developed.

In the next chapter, we will return to our sample application, the news portal, and will put what you have learned to work. We will also delve into a topic that will help you extend your application with new functionality, Extensible Server Pages (XSP).

Chapter 10. Cocoon News Portal: Advanced Version

The last two chapters showed you how to develop new Java components for Cocoon. Adding components lets you extend Cocoon in many different ways to meet your specific needs. Because of how Cocoon is designed, it is very easy to drop these new components into a given Cocoon installation just by adding them to the sitemap and deploying the compile Java classes. However, as is often the case with Cocoon, there is another way to extend Cocoon: XSP.

The first part of this chapter looks at XSP. In the second part, you use this knowledge to extend the portal from Chapter 7, “Cocoon News Portal: Extended Version.” The goal of this chapter is to enhance the previous version of the portal and build a more-advanced version.

Up to this point, the different portal versions were built without your having to code anything in Java. The advanced version of the portal requires some knowledge of Java. When you build a portal for a real-world scenario, you need to adapt to the particular environment the portal is running in. The most common way of doing this is by writing new components in Java. As you will see, XSP provides a way of writing Java code using a scripting-based approach.

In addition to writing additional code in Java, an advanced portal also needs a way to track the user, so we will look at how to use a session to store information about the portal user. We will also introduce two functions that you often find in a portal: the last-logged-in function and a greeting message. This message will be a randomly selected tip on using Cocoon, taken from the FAQ.

Extensible Server Pages (XSP)

If you’re already familiar with Cocoon, especially with the earlier versions, you might still be waiting for one important topic: Extensible Server Pages (XSP). Because you need knowledge of Java in order to use XSP, we did not think this topic was suitable for the more user-oriented chapters, because it is possible to develop web applications with Cocoon without any knowledge of Java.

Because XSP is a complex topic that could fill a whole book by itself, we decided that the easiest way to present the concepts behind XSP was to use the portal as an example. After an introduction to XSP, we will provide some information on when you could or should use XSP.

So what is XSP? XSP is a method for creating dynamic content. It is very similar to Java Server Pages (JSP) and Active Server Pages (ASP), but it aims to be a scripting language that is tightly integrated into Cocoon. The dynamic content is described by an XSP document, which is an XML document with special markup elements.

The XSP document is processed by a special Cocoon generator: the serverpages generator. So an XSP document is the starting point for an XML processing pipeline. The serverpages generator transforms the special markup into a Java class that implements the `Generator` interface. After the component has been created from the XSP markup, it is treated as a normal generator.

The next time the same XSP document is processed, the generated generator is used, so the XSP actually is compiled only once.

Hello World with XSP

Before we look more closely at XSP processing, let's start with a simple example—the great Hello World example we have neglected over the last few chapters! Here it is written in XSP:

```
<xsp:page xmlns:xsp="http://apache.org/xsp">
   <xsp:logic>
      String text="Hello World";
   </xsp:logic>
   <document>
      <xsp:expr>text</xsp:expr>
   </document>
</xsp:page>
```

After saving the XML into a document called helloworld.xsp in the context directory of the Cocoon web application, you need to add the following pipeline to your sitemap. After you restart the servlet engine, point your browser to `http://localhost:8080/cocoon/helloworld`. If everything works correctly, you should see those familiar words.

```
<map:pipeline>
   <map:match pattern="helloworld">
      <map:generate src="helloworld.xsp" type="serverpages"/>
      <map:serialize type="xml"/>
   </map:match>
</map:pipeline>
```

XSP uses its own namespace: http://apache.org/xsp. The special markup for XSP is connected to this namespace. From now on, you will use the short form of prefixes to indicate that an element is in the XSP namespace, such as `xsp:logic`. An XSP document always starts with the root tag `xsp:page`.

The `xsp:logic` element wraps the actual Java code. This element can occur as often as is needed inside an XSP file. For example, it is possible to declare some variables inside one `xsp:logic` block and then use them later in the XSP document. You have to distinguish between top-level `xsp:logic` elements and nested ones. A top-level element that is a direct child of the `xsp:page` element can declare variables. If you want to compare it with a usual Java class, all this "code" is outside any method.

A nested `xsp:logic` element runs in the context of a method, so it is possible to use any Java statement there, such as loops and method calls.

The `xsp:expr` element evaluates the enclosed expression like the `toString()` method usually does for objects. The expression can even consist of complex calculations such as `((a * 15) / 3) + 5`.

Let's look at another example. You will count from a minimum value to a maximum value and put each number in its own `number` element. At the end, you will add an element called `date` with the current date and time:

```
<xsp:page xmlns:xsp="http://apache.org/xsp">
   <xsp:logic>
      int min = 1;
      int max = 5;
   </xsp:logic>
   <document>
      <xsp:logic>
         for(int i = min; i <= max; i++) {
            <number><xsp:expr>i</xsp:expr></number>
         }
      </xsp:logic>
      <date><xsp:expr>new Date()</xsp:expr></date>
   </document>
</xsp:page>
```

The top-level `xsp:logic` element defines two constants, `min` and `max`. Inside the `document` element is a second `xsp:logic`, which contains a loop starting at the value given by the variable `min` and ending with `max`. Remember that this is an XML document, so you have to write the comparison `<=` as `<=` (or you could use a CDATA section). Inside the loop is the `number` element, and inside this element `xsp:expr` evaluates the current value of the index variable `i`.

The output XML of this XSP document is the root element `document`, which has five child nodes containing the element `number` with the numbers 1 to 5. The root element contains the `date` element with the current date and time.

Defining dynamic content this way can get complicated and verbose very quickly. To write maintainable and perhaps reusable XSP documents, it's important to understand the concept of logicsheets.

What Are Logicsheets?

A logicsheet is a library of custom XSP elements. Elements from the library can be referenced inside an XML document, much in the same way transformer functionality is referenced using tags that the transformer understands.

When the logicsheet is applied to an XML document, the referenced elements are interpreted (or better transformed) by the logicsheet. Whenever the logicsheet encounters such an element, the element is consumed. The element is removed from the SAX stream and is replaced with the result of the corresponding function from the library.

Whereas a transformer is written totally in Java, an XSP logicsheet is implemented as a stylesheet. This stylesheet can be changed during runtime without any problems, and XSP can use it immediately. In addition, the logicsheet can be registered, so it is applied automatically to the XSP document. In contrast, a transformer needs to be explicitly added to the XML processing pipeline in the sitemap.

Let's develop a simple logicsheet with two functions. The `random` function calculates a random number between a minimum and maximum value (similar to the action from the preceding chapter). The second function inserts the current date and time. Here is the stylesheet that implements the functionality:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsp="http://apache.org/xsp"
    xmlns:cxa="http://cxa/logicsheet">

<xsl:template match="cxa:date">
    <xsp:expr>new Date()</xsp:expr>
</xsl:template>

<xsl:template match="cxa:random">
       <xsp:logic>
           int min = <xsl:value-of select="@min"/>;
           int max = <xsl:value-of select="@max"/>;
           java.util.Random generator = new
     java.util.Random( System.currentTimeMillis()
);
           int random = generator.nextInt(max - min + 1) + min;
       </xsp:logic>
       <xsp:expr>random</xsp:expr>
```

```
</xsl:template>

<xsl:template match="@*|*|text()|processing-instruction()">
    <xsl:copy>
       <xsl:apply-templates
     select="@*|*|text()|processing-instruction()"/>
    </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

This logicsheet matches against two nodes, both having the namespace `http://cxa/logicsheet`. The stylesheet replaces the `date` element with the current date and time. The `random` element is replaced with a random number. This number is between the minimum value set by the attribute `min` and the maximum value set by the attribute `max`. The last rule in the stylesheet simply copies all the other elements unchanged. Let's have a look at an example using this logicsheet:

```
<xsp:page xmlns:xsp="http://apache.org/xsp"
         xmlns:cxa="http://cxa/logicsheet">
   <document>
      <random>
         <cxa:random min="1" max="100"/>
      </random>
      <date>
         <cxa:date/>
      </date>
   </document>
</xsp:page>
```

If you want to use a logicsheet, you must know which namespace it uses. So you add the namespace prefix `cxa`. In the XSP document, you simply use the two elements `cxa:random` and `cxa:date`. But how does the XSP engine know about your stylesheet? There are two possibilities. The first one is to register the logicsheet as a built-in logicsheet. The second solution is to add a reference to the logicsheet to the XSP document that will use the logicsheet. We will look at this solution later.

So how do you define a built-in logicsheet? cocoon.xconf contains the configuration for the various components used in Cocoon. Among these components is a component responsible for processing XSP documents—`xsp-language`. You can find this component in cocoon.xconf inside the `markup-languages` element:

```
<markup-languages>
   <xsp-language name="xsp">
     <parameter name="prefix" value="xsp"/>
     <parameter name="uri" value="http://apache.org/xsp"/>

     <!-- Defines the XSP Core logicsheet for the Java language -->
     <target-language name="java">
       <parameter name="core-logicsheet"
```

```
value="resource://org/apache/cocoon/components/language/markup/xsp/java
     /xsp.xsl"/>

       <builtin-logicsheet>
         <parameter name="prefix" value="xsp-request"/>
         <parameter name="uri"
     value="http://apache.org/xsp/request/2.0"/>
         <parameter name="href"

     value="resource://org/apache/cocoon/components/language/markup/xsp/
     java/
       request.xsl"/>
     ➥</builtin-logicsheet>
       <builtin-logicsheet>
         <parameter name="prefix" value="session"/>
         <parameter name="uri"
     value="http://apache.org/xsp/session/2.0"/>
         <parameter name="href"

     value="resource://org/apache/cocoon/components/language/markup/xsp/
     java/
session.xsl"/>
       </builtin-logicsheet>
       ...
    </xsp-language>
</markup-languages>
```

Among the configurations for the `xsp-language` itself is a list of the built-in logicsheets, each one indicated by the element `builtin-logicsheet`. The preceding code excerpt lists two logicsheets that are in the Cocoon distribution. We will have a closer look at them later in this chapter. But for now, let's add the logicsheet by adding these lines:

```
<builtin-logicsheet>
  <parameter name="prefix" value="cxa"/>
  <parameter name="uri" value="http://cxa/logicsheet"/>
  <parameter name="href" value="context://logicsheet.xsl"/>
</builtin-logicsheet>
```

Each built-in logicsheet gets three parameters. The first two are the namespace prefix (`cxa`) and the namespace URI (`http://cxa/logicsheet`) used in the logicsheet for the handled elements in the logicsheet. The third parameter is the location of the logicsheet. This example uses the special `context` protocol, which searches for the logicsheets in the context directory of the Cocoon web application. And that's all you have to configure. From now on, your logicsheet is available in all XSP documents.

Let's test this example. Save the XSP document as logicsheettest.xsp and the logicsheet as logicsheet.xsl in the Cocoon context directory, add the following pipeline to the sitemap, and invoke the example:

```
<map:pipeline>
```

```
    <map:match pattern="logicsheettest">
       <map:generate src="logicsheettest.xsp" type="serverpages"/>
       <map:serialize type="xml"/>
    </map:match>
</map:pipeline>
```

Because built-in logicsheets are available to all XSP documents, they should be generic enough that they can be used by several documents. If you want to create special-purpose logicsheets that should be used by only one or two documents, you can specify these logicsheets inside an XSP document instead of adding built-in logicsheets.

The processing instruction `xml-logicsheet` at the beginning of an XSP document has one attribute, `href`, that points to the location of the logicsheet. It is possible to add as many processing instructions as required:

```
<?xml-logicsheet href="logicsheet.xml"?>
```

Instead of adding your logicsheet to cocoon.xconf, you could use this processing instruction at the beginning of the XSP document and create the same effect. Of course, you can use any URI in the `href` attribute, including custom protocols such as `context:` and `resource:`.

As you saw in cocoon.xconf, Cocoon already contains some built-in XSP logicsheets. Using these libraries, it is very easy to access request information, maintain sessions, and query a database. You will use some of these logicsheets later in this chapter. First, let's take a closer look at XSP's pros and cons.

Lights, Camera, Action: When to Use XSP

In earlier versions of Cocoon, XSP was the only alternative for defining dynamic content. Therefore, it still creates some interest in the Cocoon community. But with Cocoon's current architecture and the many different types of components, such as transformers and actions, XSP is losing some of this interest.

In fact, XSP has some disadvantages. One of them is that the skill required to use XSP is greater than for using other components, such as transformers. XSP requires at least basic knowledge of Java programming. Because a special generator is responsible for interpreting the XSP tags, it is very easy to mix content and logic into a single XSP document.

For production environments, XSP is not the best solution when it comes to performance. As a default, it does not support the caching algorithm. Because it is possible to write any Java statement inside the XSP document, the serverpages generator cannot decide which XSP document is cache-aware. Consider a simple XSP document that "prints" the current date and time. If this were cached, all clients would see only the date and time of the first invocation. However, if you are an experienced XSP and Cocoon developer, you can

make your XSP document cache-aware by implementing the `Cacheable` interface, introduced in the preceding chapter.

Another minor performance problem is compiling the XSP document into a generator. This process is rather slow. But because this happens only once for each XSP document, it is unimportant.

A logicsheet has one advantage over a custom transformer: It is applied automatically to the XSP document. You do not have to include a custom transformer in the processing pipeline. Using a new logicsheet inside an XSP document does not require the administrator to change the sitemap or any other configuration file. It can simply be used. However, if a new transformer is used, the sitemap containing the XML processing pipeline has to be adjusted.

On the other hand, a logicsheet has two disadvantages compared to a transformer. First, an XSP logicsheet requires the serverpages generator. Without this special generator, the logicsheet cannot be applied to the elements. So it is not possible to use a logicsheet with a different generator that limits the use of logicsheets. Second, writing a logicsheet is a subtle task, because it is very easy to create conflicts between logicsheets that make the XSP document unusable. For example, if an XSP document uses two logicsheets, and both declare a variable with the same name, a compilation error occurs, because a variable can be declared only once.

“So why use XSP, then?” you might ask. If you don’t want to program Java, some things are currently possible only using XSP, such as maintaining a session for the current user. For certain functionality, you can therefore either use available XSP logicsheets or write your own generator, transformer, or action in Java.

Another advantage of XSP is rapid prototyping. The XSP document is transformed dynamically during runtime into a Java class or, more precisely, into a generator. So writing your own generator and changing it are possible with XSP without the need to deploy Java classes and restart the servlet engine. A simple change in the XSP document is sufficient.

Developers use this approach to create their own generator. They use XSP to prototype it. When they are happy with the result, they use the generated class and maintain that from then on as a custom generator and forget about the XSP document.

When compared to other scripting languages, XSP has the advantage that the markup is not hard-coded into the scripting engine. Instead, even the core markup, such as `xsp:page`, is transformed by a stylesheet. This loose coupling makes this approach very flexible and maintainable.

Whether you want to use XSP is up to you. It probably depends on the exact solution you are building. Here are some additional points you should consider when deciding if XSP suits your needs:

- Imports can't be specified in the logicsheets, so classes should be named to include their package names. Otherwise, a compile error will occur.
- Exceptions other than `ProcessingException`, `SAXException`, and `IOException` must be caught explicitly by including a `try-catch` block in the XSP code.
- Errors aren't known until runtime, when the generated Java file is compiled. With custom generators, errors are known at compile time.

We should also point out that some of the functionality provided by XSP is slowly being replaced by components that do the same thing. As we will show you in Chapter 12, "Cocoon: Weaving the Future," there is an alternative to XSP when it comes to session handling. But session handling is a good reminder of what this chapter is about: developing your portal. So let's start with this now.

Extending the Extended Portal

In Chapter 7, you developed a simple portal where the user can log in and receive a personalized news feed. In this chapter you will use this portal as a base and extend it by introducing the concept of a session to store information about the user on the server. You will also rewrite most of the portal to use XSP and then later add new functionality to the portal.

Adding Sessions

A common feature of web solutions is the concept of *sessions*. A session allows the application on the server to store information about the current user accessing the application. Applications are forced to do this themselves, because the HTTP protocol is stateless.

When a web application built in Cocoon decides it needs session storage, it asks Cocoon to create a session for the current user. If no session currently exists for this user, a new session is created. Otherwise, the old one is used instead.

This session is a Java object that offers methods to store and retrieve any other Java object using a key. The stored objects are called *attributes*, so the corresponding methods of the session are named `getAttribute()` and `setAttribute()`.

Your portal will use the session to store the user's name and his preferences after he logs in. This eliminates the need to fetch the data from the database each time the user requests the portal.

After a session is created, Cocoon can check an incoming request to see if it is from a user who already has a session. If the user has a session, the web application can directly use this session and get the information stored in it from an earlier request.

When the session is no longer needed, it has to be destroyed programmatically by the web application. Usually when the user invokes the "log off" function, the server frees

the information. From now on, each incoming request is no longer connected to a session, because it has been deleted.

If the user does not select the “log off” function, Cocoon offers the possibility of defining a period of inactivity after which Cocoon automatically destroys the session.

How can Cocoon detect whether a request belongs to a session? Because Cocoon runs in an environment such as a servlet engine, it uses the features of this environment. Servlet engines generally use two methods to maintain sessions: cookies or URI rewriting.

The default is usually to use cookies, because the developer of the web application does not need to do anything special to use them. When a new session is created, a cookie is sent to the client that contains a unique handle for the session. Because a cookie is sent back to the server by the client with each new request, it is very easy for the server to then find the correct session.

The second method, URI rewriting, has the advantage that no information is stored on the client in the form of a cookie. But it has the disadvantage that the developer of the web application has to take care of the URI rewriting. When this method is used, each link or reference contained in the document that is sent back to the client is postfixed with a unique handle for the session. For example, when the link “portal” is used in an HTML document, it is rewritten by the web application as something like `portal;jsessionid=524AF3`. So when the user selects this link, the URI with this special information is sent back to the server, which extracts the information to get the corresponding session.

This method is called *URI rewriting*, because the correct URI “portal” must be rewritten by the web application as `portal;jsessionid=524AF3`. Unfortunately, this does not happen automatically. Do you remember the `Response` object that is available for each request inside Cocoon? This object offers a method that gets a URI as the argument and returns the rewritten URI.

Your portal example will not use URI rewriting, because it is complex enough. So this version will use cookies. But this could be the first part where you can practice later on yourself.

But even to create and maintain a session inside your portal, you need to write Java code, because the session is a Java object. But don’t fear. You will not develop new components, as you did in the preceding chapter. Instead, you will use XSP. You’ll get a little help from XSP, because Cocoon already offers a logicsheet for session handling: the session logicsheet. You will use two functions of this built-in logicsheet.

The `session:get-attribute` function gets the object stored in the session attribute under the given name and converts it to XML. The value of the attribute replaces the `session:get-attribute` element. The `session:set-attribute` function sets the value for the attribute with the given name.

The following fragment shows these two functions, a third one for ending a session, and the namespace and prefix used for this built-in logicsheet. Of course, the session logicsheet offers more functions than these two. You can find more information in the provided documentation on the CD and in the logicsheet's source, also contained on the CD.

```
Namespace: http://apache.org/xsp/session/2.0
Prefix: session:

<session:get-attribute name="name of attribute"/>
<session:set-attribute name="name of attribute">
    value
</session:set-attribute>
<session:invalidate/>
```

Cocoon offers a second logicsheet that helps you: the request logicsheet. As the name implies, this logicsheet contains some functions for getting information from the request: the request parameters, the URI requested from the client, and so on. The following code shows the used namespace and prefix for this built-in logicsheet:

```
Request Logicsheet
Namespace. http://apache.org/xsp/request/2.0
Prefix: xsp-request

<xsp-request:get-parameter name="name of request parameter"/>
```

Again, the request logicsheet offers more than just the `xsp-request:get-parameter` function. You will find more information in the documentation and source on the CD.

So you already know two logicsheets that will help you. With these two, you will develop most parts of the session and request handling. But Cocoon offers even more logicsheets.

Database Queries with XSP

Whenever you fetched data from a database in the last chapters, you used the sql transformer and a database connection configured in cocoon.xconf for this purpose.

You will now use a different approach. Cocoon already contains a logicsheet that performs nearly the same tasks as the sql transformer. It's called the esql logicsheet. This logicsheet is very complex and powerful. It offers even more possibilities than the sql transformer. We will focus on only some of the functions. The complete description can be found in the documentation on the CD or in the logicsheet's source. The following shows a standard use of the logicsheet for a database fetch:

```
Namespace: http://apache.org/cocoon/SQL/v2
Prefix: eqsl
```

```
<esql:connection>
    <esql:pool>portal</esql:pool>
    <esql:execute-query>
        <esql:query>
            SQL STATEMENT
        </esql:query>
        <esql:results>
            <esql:row-results>
                <eqsl:get-string column="id"/>
            </esql:row-results>
        </esql:results>
    </esql:execute-query>
</esql:connection>
```

The `esql:connection` element starts the database command. The nested `esql:pool` element tells the esql logicsheet which connection it should use. The element's content is the name used for the configuration in cocoon.xconf.

The actual database fetch starts with the `esql:execute-query` element. This is very similar to the sql transformer notation. The nested `esql:query` element contains the SQL statement. If a result is fetched, the esql logicsheet evaluates the branch contained in the `esql:results` element. If no result is fetched, the elements contained there are never evaluated.

For each fetched row, the `esql:row-results` element is executed, and the contained elements are included in the XML document. The preceding example uses the `eqsl:get-string` element to fetch the value of one distinct column named `id`. So, for each fetched row, the value of this column is added to the generated document.

If you are interested in the complete functionality available using esql, we refer you to the documentation contained in the Cocoon distribution.

Building the Portal with XSP

Now you have everything ready to start your advanced portal. To remind you of what you want to do, here is an overview. You will develop a login document in which the user can enter his name and password. This document invokes the portal document.

The portal document authenticates the user against a SQL database. If the user is known and if the password is valid, the portal document displays this user's portal. If for any reason the user cannot be authenticated, a message is displayed stating this.

From the portal document, the user can either log out of the portal or enter the editfeeds document. The editfeeds document displays all the currently selected news-feeds, and the user can delete a news feed or add new ones.

The Start Document

You remember the start document from Chapter 7. It contains a form with a text field for the username and a text field for the password. In addition, the portal document is invoked when this form is submitted from the client.

You can use the XML and XSL documents from Chapter 7 with two slight modifi-cations. The XML document is now an XSP document, so you have the root element `xsp:page`. But you don't use any XSP functionality.

```
<xsp:page xmlns:xsp="http://apache.org/xsp">

<start>
    <pipeline>portal</pipeline>
    <idfield>id</idfield>
    <passfield>password</passfield>
</start>

</xsp:page>
```

The second modification is a message for a new predefined user to the stylesheet. We do not show the stylesheet here because it is basically the same as in Chapter 7. (But you can find the whole portal you are building in this chapter on the CD.)

The start document invokes the portal document and sends the values of the form with the request to authenticate the user.

Extending the Custom Logicsheet

You want to use sessions to store information about an authenticated user, so you need a way to create a session. Unfortunately, the session logicsheet does not help you here, so you have to write this function yourself.

For this, you will extend the logicsheet started earlier in this chapter. You will add a new function called `cxa:create-session`. The template looks like this:

```
<xsl:template match="cxa:create-session">
   <xsp:logic>

    org.apache.cocoon.environment.ObjectModelHelper.getRequest( objectM
    odel
).getSession(true);
   </xsp:logic>
</xsl:template>
```

You simply get the `Request` object from the object model and get the session object using the `getSession` method with the Boolean argument `true`. This argument indicates

that if no session exists for the user, a new one is to be created. In case you have forgotten the details of the object model and the request object, take a look at Chapter 8, "A Developer's Look at the Cocoon Architecture." We will be here waiting when you get back.

In addition to authenticating the user, you want to protect your documents. If a user is not authenticated, it should not be possible to get at the content of the portal or the editfeeds document. You will implement this by testing inside these two documents to see if a session for the user exists.

If a session exists, the user is authenticated, and you can display the portal or the edit document. If the user is not authenticated, you will only display a message that he is not allowed to see this document.

So, how do you test to see whether a session exists? In Java code, this is very easy. You can simply ask the `Request` object if a session is present by calling the `getSession()` method with the Boolean argument `false`. So you could simply use this inside your XSP documents to test whether a session exists.

But to show how you can use XSP logicsheets, we have added another element to the logicsheet that does the tests for you:

```
<xsl:template match="cxa:if-session">
  <xsp:logic>
     <xsl:choose>
        <xsl:when test="not(@test) or @test = 'true'">
           if (org.apache.cocoon.environment.ObjectModelHelper.
              getRequest( objectModel ).getSession(false) != null) {
        </xsl:when>
        <xsl:otherwise>
           if (org.apache.cocoon.environment.ObjectModelHelper.
              getRequest( objectModel ).getSession(false) == null) {
        </xsl:otherwise>
     </xsl:choose>
  </xsp:logic>

  <xsl:apply-templates/>

  <xsp:logic>
     }
  </xsp:logic>
</xsl:template>
```

The `cxa:if-session` function tests whether a session exists and executes the elements contained in this element if the test is successful. The function is implemented by getting the `Request` object from the object model and trying to get the session from the `Request` object. Let's have a look at a short example:

```
<cxa:if-session>
   Yes
</cxa:if-session>
<cxa:if-session test="false">
  No
</cxa:if-session>
```

When this code is processed by your custom logicsheet, the whole block is replaced with either the text `Yes` or `No`, depending a session's existence. With the attribute `test` and the value `false`, the logic is reversed, which means that the nested elements are executed if no session exists.

With these two functions, you can convert the portal example to XSP, so let's continue with the portal document.

The Portal Document

In Chapter 7, you used a combination of several stylesheets, the sql transformer, and the cinclude transformer to authenticate the user, fetch the news feeds, and include the news in the portal view. You will now rewrite this rather complex XML pipeline as a very simple one that uses XSP and the previously introduced logicsheets.

The first thing you do is test whether a session already exists, because you want the authentication to be done only if no session is available. You use the `cxa:if-session` element to do this:

```
<xsp:page xmlns:xsp="http://apache.org/xsp"
        xmlns:xsp-request="http://apache.org/xsp/request/2.0"
        xmlns:esql="http://apache.org/cocoon/SQL/v2"
        xmlns:session="http://apache.org/xsp/session/2.0"
        xmlns:cxa="http://cxa/logicsheet">

<document>
   <cxa:if-session test="false">
      <esql:connection>
         <esql:pool>portal</esql:pool>
         <esql:execute-query>
            <esql:query>
               select * from PORTALUSER_TABLE
               where id = '<xsp-request:get-parameter name="id"/>'
                    and password = '<xsp-request:get-parameter
    name="password"/>'
            </esql:query>
            <esql:results>
               <cxa:create-session/>
               <esql:row-results>
                  <session:set-attribute name="id">
                     <esql:get-string column="id"/>
                  </session:set-attribute>
                  <session:set-attribute name="color">
                     <esql:get-string column="color"/>
```

```
                    </session:set-attribute>
                </esql:row-results>
            </esql:results>
        </esql:execute-query>
    </esql:connection>
</cxa:if-session>
</document>
</xsp:page>
```

If no session is available, you use the esql logicsheet to fetch the data from the database. Because this document is invoked from the start document, the request contains two parameters, `id` and `password`. You will use this to look up the user in the database.

If a result is available, which means that the username and password are valid, a session is created by calling `cxa:create-session`. Each value fetched from the database is written to a separate attribute in the session, and the `id`, `password`, and `color` are stored.

The first time this portal document is requested, or when no session is available, this document tries to authenticate the user and then stores the information in the session.

The next step for the portal document is to present the portal, which means that it has to fetch the configured news feeds from the database and get the news from `moreover.com`. You get the news using the `SourceResolver`, as you did in Chapters 8 and 9.

```
<cxa:if-session test="true">
    <xsp:logic>
        org.apache.cocoon.environment.Source source;
        org.apache.cocoon.xml.IncludeXMLConsumer consumer =
            new org.apache.cocoon.xml.IncludeXMLConsumer(xmlConsumer);
    </xsp:logic>
    <user>
        <name>
            <session:get-attribute name="id"/>
        </name>
        <background>
            <session:get-attribute name="color"/>
        </background>
        <feeds>
            <esql:connection>
                <esql:pool>portal</esql:pool>
                <esql:execute-query>
                    <esql:query>
                        SELECT newsfeed from MOREOVER_TABLE
                        where name = '<session:get-attribute name="id"/>'
                    </esql:query>
                    <esql:results>
                        <esql:row-results>
                            <xsp:logic>
                                source = null;
                                try {
                                    source = resolver.resolve(
```

```
      "http://p.moreover.com/cgi-local/page?index_"+<esql:get-string
    column="newsfeed"/
>+"+rss");
                              source.toSAX( consumer );
                           } catch (Exception ignore) {
                        </xsp:logic>
                        <p>Newsfeed <esql:get-string column="newsfeed"/>
    currently
not available.</p>
                        <xsp:logic>
                           } finally {
                              if (source != null) source.recycle();
                           }
                        </xsp:logic>
                     </esql:row-results>
                  </esql:results>
               </esql:execute-query>
            </esql:connection>
         </feeds>
   </user>
</cxa:if-session>
```

First you declare two variables for your XSP code: a `Source` object and a `consumer`. Both are used to resolve and include the news in the current document. You resolve a news feed by using the `SourceResolver`. Because the resolved `Source` object always streams a full XML document containing the `startDocument` and `endDocument` SAX events, you need to filter these events. That's the purpose of `IncludeXMLConsumer`.

You generate an XML document containing information about the user, such as the name and the color. These values are stored in the session, so you only need to get them out of the session. The `feeds` section is more complex. You use the esql logicsheet to fetch the data from the database. For each fetched row or, more precisely, each fetched news feed, you write some Java code.

This code resolves the news feed and includes it in the current document. The code is very similar to the code you used in Chapters 8 and 9 to resolve documents. If an exception occurs during the processing, you include a message stating that the news feed is currently unavailable.

The last code you add tests whether the authentication was unsuccessful:

```
<cxa:if-session test="false">
   <p>Please try again.</p>
</cxa:if-session>
```

If a user who is not authenticated requests the portal document, he sees this message. Together with the preceding pieces, you have a complete XSP document that covers the same functionality as the portal in Chapter 7. Because the XML produced by this XSP is very similar to the XML of the extended portal, the final stylesheet to present the portal

in HTML is similar, too. We do not repeat it in this chapter, but you can have a look at it on the book's companion CD.

The Edit Document

The remaining part of the portal is the editfeeds document, which allows the user to select new news feeds or remove a news feed from his portal.

This XSP document is very similar to the portal XSP document. It tests whether a session exists in order to verify whether the user is authenticated. If the user is not authenticated, a message is displayed stating this. However, if the user is authenticated, the esql logicsheet is used to fetch the different news feeds from the database for the user. The name and ID of the feeds are included in the generated document. A stylesheet uses this information to present an HTML document similar to the one from Chapter 7.

If the form is submitted, the same technique is used to update the database as in Chapter 7: An action-set evaluates the request parameters and either adds a new news feed or deletes an old one.

Because the XSP document and the stylesheet do not present anything new, we refer you to the CD to have a look at them.

Adding New Features

You have completed the first step of your advanced portal:You have converted the extended news portal to XSP and have added a session to track the user and store information. You have a nice portal that is a little faster because of the sessions than the version in Chapter 7. But to be honest, you haven't made real progress in adding user functionality.

But don't be disappointed. This chapter is twofold. The first part had the aim of introducing XSP, and by converting the portal, you achieved this goal. The next good news is that the second part of this chapter deals with extending the portal! Because you now are using XSP as the base for your advanced portal, you can easily integrate new functions.

The Logout Document

Because you have created a session for the user, you should finish the session. It's best to do this when the user logs out.

So you add a new link to the portal document that allows the user to log out. This link requests the logout document. This XSP document is shown here:

```
<xsp:page xmlns:xsp="http://apache.org/xsp"
        xmlns:session="http://apache.org/xsp/session/2.0">
```

```
<document>
   <session:invalidate/>
   <p>You are logged out now.</p>
</document>

</xsp:page>
```

This is a very simple XSP document that uses the session logicsheet to end the session. It then displays the message `You are logged out now`. A stylesheet that you will find on the CD transforms the output of this XSP document to HTML.

Last Logged In

A common feature of a "system" in which a user has to log in is that these "systems" display when the user was last logged in. Believe it or not, you will add this to your portal, too.

For this, you add a new column for each user to your database containing the date he last logged in. When the user is authenticated, this value is fetched from the database and stored in the session. The value is updated to the current date, so the next time the user logs in, this date is presented.

You store the value of the database column in the session using the following statement:

```
<session:set-attribute name="last-logged-in">
   <esql:get-string column="last_logged_in"/>
</session:set-attribute>
```

After the authentication procedure, you test whether a session exists to detect whether the authentication was successful. In this case, you send a SQL `update` command to the database, updating the last-logged-in date to the current time:

```
<cxa:if-session test="true">
   <esql:execute-query>
      <esql:query>
         update PORTALUSER_TABLE set last_logged_in='<cxa:date/>'
                where id = '<xsp-request:get-parameter name="id"/>'
      </esql:query>
   </esql:execute-query>
</cxa:if-session>
```

You use your custom logicsheet to get the current date using `cxa:date`. The rest is the usual SQL update command for updating one column.

In addition, you have to include the last login date in the XML document in order for the stylesheet to present the user with this information in HTML:

```
<last-logged-in>
   <session:get-attribute name="last-logged-in"/>
</last-logged-in>
```

Because the value is already stored in the session, you only have to get it and include it. The stylesheet for the portal picks up this value and displays `You were last logged in on...`.

With this little function, your portal looks a bit more professional. With the next feature, you will add a function that is unavailable in most portals.

Random Cocoon Tip

You know many programs that display a “tip of the day” each time the program is started. With Cocoon, you can do this even better:You can present a random tip each time the portal is requested.

You might wonder where you get all these tips. They are contained in the Cocoon web application inside the online documentation (the documentation has a FAQ). This FAQ is—guess what—an XML document found in the documentation/xdocs directory inside the Cocoon context directory. This document is called faq.xml.

So you have a FAQ from which to select a tip, and you have your XSP logicsheet that can generate random numbers. So the only thing remaining is to connect these two in your portal document:

```
<faq-section>
  <select><cxa:random min="1" max="50"/></select>
  <xsp:logic>
     source = null;
     try {
        source =
    resolver.resolve("context://documentation/xdocs/faq.xml");
        source.toSAX( consumer );
     } catch (Exception ignore) {
     } finally {
        if (source != null) source.recycle();
     }
  </xsp:logic>
</faq-section>
```

First you calculate a random number between 1 and 50, and then you use the `SourceResolver` to include the FAQ XML document in your generated XML. Note that you include the complete document, not only the tip you are interested in. These two pieces of information (the random number and the complete FAQ) are then evaluated by the portal stylesheet:

```
<xsl:template match="faq-section">
   <xsl:variable name="select">
      <xsl:choose>
         <xsl:when test="normalize-space(select) > count(faqs/faq)">
            <xsl:value-of select="select mod count(faqs/faq) + 1"/>
         </xsl:when>
         <xsl:otherwise>
             <xsl:value-of select="normalize-space(select)"/>
         </xsl:otherwise>
       </xsl:choose>
    </xsl:variable>
    <hr/>
    <h2>Cocoon Random Tip(No. <xsl:value-of select="$select"/>)</h2>
    <xsl:apply-templates select="faqs/faq[position() = $select]"/>
</xsl:template>
```

The first part of this stylesheet template tests whether the random number is higher than the number of tips contained in the FAQ. If this is the case, the random number is corrected. With the now-correct number, `<xsl:apply-templates/>` is invoked with ➥ only the node of the tip in question. So the stylesheet filters the FAQ, and only one FAQ entry remains in the resulting HTML document.

Because the whole FAQ is not formatted in HTML, you added some more templates to the stylesheet to present the FAQ entry in a more HTML-style format.

Now that you have all the pieces implemented, let's put them to work.

Running the Portal

Your portal consists of several pieces. You have the SQL database, the XSP logicsheet, and the various XSP and XSL documents. Of course, because these are all contained on the CD, you only have to copy the contents of the example folder from the CD into the Cocoon context directory. So you have logicsheet.xsl and the portal directory directly beneath the Cocoon context directory.

The next task is to configure the HSQL database. Because you modified one table, you have to delete the entries for the portal demo from Chapter 7 first. Then you add the new configuration to cocoondb.script, which can be found in Cocoon's \WEB-INF\db directory. This file can be edited with any text editor. The following lines then need to be appended to the end of the file:

```
CREATE TABLE PORTALUSER_TABLE(ID VARCHAR,PASSWORD VARCHAR,COLOR VARCHAR,
    LAST_LOGGED_IN
VARCHAR,UNIQUE(ID))
CREATE TABLE MOREOVER_TABLE(ID INTEGER,NAME VARCHAR,NEWSFEED
    VARCHAR,UNIQUE(ID))

INSERT INTO PORTALUSER_TABLE VALUES('matthew','wizard','yellow','never')
```

```
INSERT INTO PORTALUSER_TABLE
    VALUES('carsten','professional','red','never')
INSERT INTO PORTALUSER_TABLE VALUES('cocoon','magic','white','never')
INSERT INTO MOREOVER_TABLE VALUES(1,'matthew','banking')
INSERT INTO MOREOVER_TABLE VALUES(2,'cocoon','usa')
INSERT INTO MOREOVER_TABLE VALUES(3,'cocoon','banking')
INSERT INTO MOREOVER_TABLE VALUES(4,'carsten','computer')
```

Next comes the connection configuration for the database in cocoon.xconf. If you still have the configuration from the extended portal, you don't have to add it again. If you don't, you need to create a new database connection. You do this by adding a few lines to cocoon.xconf. Open the file with your favorite XML editor (or a simple text editor if you prefer). Find the <datasources> section of the file. This part of cocoon.xconf contains the configured database connections, and it is where you will add the new one. Add the following lines inside the <datasources> tags:

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

Make sure that you still have configured your own logicsheet as a built-in logicsheet in cocoon.xconf. If you don't, have a look at the beginning of this chapter to see how you can add it.

As always, the last part is to add the entries to the sitemap. As in Chapter 7, you have three parts to add. First are two custom actions:

```
<map:action name="add-feed"
    src="org.apache.cocoon.acting.DatabaseAddAction"/>
<map:action name="del-feed"
    src="org.apache.cocoon.acting.DatabaseDeleteAction"/>
```

Then comes the new action-set:

```
<map:action-set name="portal">
  <map:act type="add-feed" action="Add"/>
  <map:act type="del-feed" action="Delete"/>
</map:action-set>
```

Last are the pipelines:

```
<map:pipeline>
   <map:match pattern="newsportal">
      <map:redirect-to uri="portal/user/start"/>
   </map:match>
```

```
    <map:match pattern="portal/user/*">
       <map:act set="portal">
          <map:parameter name="descriptor"
     value="context://portal/resources/
dbfeeds.xml"/>
       </map:act>
       <map:generate src="portal/resources/{1}.xsp" type="serverpages"/>
       <map:transform src="portal/styles/{1}.xsl">
          <map:parameter name="use-request-parameters" value="true"/>
       </map:transform>
       <map:serialize type="html"/>
    </map:match>
</map:pipeline>
```

Make sure that you remove all configuration from the extended portal beforehand. So, that's it. Now start Cocoon and invoke `http://localhost:8080/cocoon/newsportal`. You will see the portal's login page. Log in and enjoy the new version!

Of course, some of the statements from Chapter 7 are still valid. This is not an award-winning solution. But we think it is a good starting point for you to experiment with the wonderful world of Cocoon. So you can either look at Chapter 7, where we suggested some improvements, or you can use your imagination.

Conceiving and Designing a Cocoon Application

This finishes your full-featured Cocoon application. First you converted the extended portal from Chapter 7 to XSP, and then you added some more functions, such as a last-logged-in function and a random tip.

It is obvious from the sample portal that using a session is very useful in developing web applications. Unfortunately, currently the only way to deal with sessions is to use XSP or to write custom components. Chapter 12 presents some alternatives currently in development for Cocoon. These alternatives include a complete portal framework and complete session handling without the need to program Java or XSP.

This sample portal included all the important information in this chapter, especially a concept of what you wanted to build and what Cocoon provides that allows you to develop the solution.

In real life, however, you rarely are presented with a finished concept. More often, a problem is presented that needs to be solved. It's your job to first design the concept and then build the web application defined by this concept. The next chapter goes over some tips on conceiving and building advanced XML web applications with Cocoon.

Chapter 11. Designing Cocoon Applications

The previous chapters discussed how Cocoon provides a complete XML platform for building applications. We looked at how solutions developed with Cocoon can meet the challenges facing today's modern application architectures. We also presented some examples for small applications and built a personalized news portal using Cocoon concepts and technologies.

Cocoon is not a platform specifically aimed at only one application area, such as a portal. Cocoon can be used to build a variety of applications and solutions. Because we have been using Cocoon as a base for the paid work we do, we have built web sites and portals and have also used Cocoon to build front ends for databases, XML processing systems, and integration systems for different hosting environments, such as those used for Application Service Providing (ASP).

It is our experience that learning to use Cocoon to build these types of applications takes time, because the philosophy behind the solution is different from the way Internet applications are commonly built, using scripting languages such as ASP and JSP or dedicated software solutions built using servlets or other components.

We have included this chapter to provide additional background information and tips that we hope will help you if you want to develop a more advanced Cocoon application, such as an Internet portal. A lot of this information will not be completely new if you have read through the book. However, we have often heard people say, "There is so much in Cocoon. What do I actually need if I want to build a certain type of application?" The aim of this chapter is to provide this information in a different context so that you can then go back to where we originally explained it for the full details.

Before getting into the different types of applications you can build with Cocoon, we will start with some general points that are important when designing any type of software solution. Although this might seem to be a long list of things to think about, remember that you will probably only need to look at individual points when you start building real applications, such as a new Internet portal for a major client. That being said, it is always a good idea to start with a concept of what you will do.

The Application Concept

Few people can cook exotic meals without a recipe. The recipe gives you an idea of what the result will look like, what ingredients you need, and how you should prepare the dish. Using a recipe as a concept for your meal is common sense.

When you build an application with Cocoon, a concept that includes the points discussed in the following sections helps you plan your solution and prevents you from making some of the more common mistakes. The following sections define the system functionality, the application architecture, and aspects such as performance and presentation design.

While thinking about these points, you can also try to work out which of the described Cocoon technologies will be important for what you want to do and whether you perhaps need to write additional components. We will also provide some guidance for the times when, even after you've done all this, things still don't work as you expected. We will start with probably the most common question asked of any application: "What's it supposed to do?"

General Functionality

The first step is to define the functionality of the system you will build. Most systems built with Cocoon publish data in some way. In addition, there might be functions that allow the user to interact with the application in some form. Depending on the type of application, it might be necessary to define areas of information that are then combined into the complete application.

As an example, imagine that you are building a web site application for an imaginary company that produces Rewinders (don't ask us what these are; it's imaginary). Obviously you need functions that allow general information about your firm to be published. However, you have several different areas of information you want to publish, so you need to structure the application. Here are a few areas you might want to define:

- General information about Rewinders
- News about the company (Rewinder Inc.)
- Industry news
- Products offered
- Jobs
- Information for employees only

If you check out some company web sites, you will see that most have this sort of structure. Each area in your web site will have subareas that provide more detailed information. An area such as "Products" will contain all the different types of Rewinders that are offered. The section called "Employees Only" will provide special information about upcoming Rewinders. This information should be available only to someone who has logged on to the system.

After you have designed the application's structure, it is time to think about any interactive components you might need. Perhaps you will need an application form in the "Jobs" area or a feedback form in the "Products" area. In addition, you will need some form of login page for the "Employees Only" area. You also want to know when someone looks at the new "Cool Blue Rewinder," so you specify that you want an email to be sent when that document is viewed.

Depending on the type of application you are building, you might need only publication functions. If your solution is aimed more at processing information, you need more functions that allow interaction with your system.

After you have set up the application's structure, you must work out how navigation through the system is possible. After someone enters the "Products" area, what other areas can he access from there? What happens if he accesses the "Employees Only" area? Working out the navigation and flow can be one of the most time-consuming jobs when designing the application.

A typical application will be a combination of published data and data that flows from the user to the application. After you have defined the site's structure, it is time to think about the content.

The data you want to publish must come from somewhere. Either it is already stored in a file or database, or it will be obtained from external sources at runtime. An area such as "Jobs" will access the current job openings from a database. An application area such as "Industry News" will probably access a news provider to obtain news about the current state of the Rewinder industry. The authentication data is also probably contained in a database. You will need access to it to check such things as the password when a user wants to access the "Employees Only" area.

As soon as you know where the data you want to publish comes from, you need to determine what format it is available in. Of course, it is ideal if the data is supplied in an XML format.

Next you need to define your output formats. Your imaginary company wants to publish its web site in HTML first. In addition, some of the documents are to be in PDF, and you want to offer product descriptions in WML.

Notice that we have not yet talked about a specific technology. Indeed, a first concept does not require any knowledge of how you will realize your application. As soon as you have the concept in place, you can decide which technology to use (in this case, Cocoon) and then move on to defining the actual system architecture.

Application Architecture

After you have defined and documented the points just discussed, you can start building the actual architecture for your application using Cocoon. You need to define the various

documents you want to publish through the web site and work out what sort of pipelines you need in order to generate the different formats.

Here are some of the types you need for Rewinders Inc.:

- Pipelines that obtain data from a file and format that data in HTML or WML (depending on the browser)
- An additional pipeline that sends an email if a particular document is chosen (such as the Cool Blue Rewinder)
- Pipelines that access data from a database and format it in PDF (for online product handbooks)
- A pipeline that accesses online industry data and formats it in HTML
- A pipeline that receives the incoming forms data from the feedback form and saves it to a database

As soon as you have laid out the types of pipelines you need and have decided how many of each you require, you might need to think about splitting them between subsitemaps to ease maintenance of the complete site. Another alternative might be to use content aggregation to combine separate pipelines into a single pipeline that is then formatted for output.

Because a complete application architecture is seldom confined to just one area, such as what you build with Cocoon, you also need to think in advance about things such as bottlenecks that might occur when you roll out your solution:

- What would happen if all 30,000 employees accessed the “Employees Only” page at the same time?
- How will the system react when all the customers hit the “Cool Blue Rewinder” page at exactly the same time?
- Will the email system be able to cope with all the emails?

These are the sorts of questions you should ask yourself while designing the application architecture. This brings up one of the most important aspects of such a system: performance.

Performance and System Environment

We have been building Internet applications for quite a few years, and it is our experience that one of the most common problems is that after it is installed, the solution is always too slow. This is something that all applications suffer from, as you can see in the various discussion forums of any software product.

This does not always mean that the programming is sloppy (although perhaps often it is). There is often a great difference in the speed an application can actually achieve and the perceived performance that the end-user might experience. Also, a system that performs

well when only one user accesses it might collapse if several users send requests at the same time.

A system that integrates many different data sources might suffer from bad performance even though the actual portal application might be fast enough. A portal's speed is defined to a great extent by the speed at which data from external systems is delivered. So a portal will be slow if one of the data sources is slow. Unfortunately, no one will care that it is not your fault if it takes minutes for the portal to appear in the browser.

When designing a complex software solution, it is always best to define performance expectations beforehand and to test for performance as early as possible. This sounds simple, but this point is often forgotten until it is too late. If nobody takes the time at the beginning of the project to define the expected performance, the system will always be too slow. It is a lot more difficult to correct performance problems after the solution is in a production environment.

When we installed our first online Internet banking solution, very few people accessed their accounts via the Internet. The application worked well and delivered the web pages quickly enough. However, no real stress testing was performed at the beginning, so we did not really know how many requests our system could handle. Over the months the application was installed, the number of requests grew slowly but steadily. Still, no stress testing was done. After all, the system ran OK didn't it? Then, for some strange reason, the number of people using the system suddenly exploded overnight! Needless to say, the whole system collapsed under the load. It was far worse having a nonfunctional system in this situation than it would have been when Internet banking was still an exotic application.

How do you define a system's expected performance? It depends on what the system is supposed to do. The first thing you can do is check out the data sources and decide what sort of performance you can expect from them. If you are integrating standard data sources (such as a standard database), you can often obtain performance data from the vendor. Get that data, but take in the information with a grain of salt. To really check, you need to run your own isolated tests against the single system if you can. It is much more difficult to find bottlenecks after everything is integrated.

Before you start evaluating the performance of individual systems, make sure you also define your computing environment. What's the good of testing the system on some high-powered system if it will actually be running on a low-end box? Also make sure you test on the same operating system and using the same hosting software (such as a servlet engine). The servlet API might be standardized, but in reality you will find that life is not so simple. And it is a lame excuse to say, "We didn't test on that system" when a complaint comes in.

Another way to find out what to expect from your system is to check out other solutions that might do the same thing you are planning on doing. See how fast they run, and try to obtain some information on how they work. Check out case studies, often published on

web sites, to find out the architecture used to build the application. You might also be able to find out by asking whoever built the system.

As soon as you are satisfied that you know what to expect of your application, here are some tips on what you can use in Cocoon to achieve the fastest possible application:

- Use the built-in Cocoon caching whenever possible when building your pipelines.
- If you need to write your own components, make sure they support the caching interfaces in Cocoon if possible.
- Stress-test your application using an available tool, and observe how the performance changes if you adjust the pooling of Cocoon components.
- Make sure you are running your application with the lowest level of trace, where only errors are logged.

Another piece of advice when writing components that connect to a specific data source (especially if it is not your data source) is to make sure you add a time trace. In other words, trace when you connect to the external data source, and trace when the data is returned. That is the time someone else has to worry about.

If, after testing with stress tools, you find that your system performance is not good enough, you will want to look into what else you can do to improve the response time. Obviously it is a good idea to make sure the system has enough memory and the processor is fast enough. If you are running in a servlet environment, you might want to try an alternative servlet engine to see if you can get better performance.

You might also want to look into front-side and back-side caching. A front-side cache is placed between Cocoon and the Internet. Any client program requesting a particular document receives it from the cache, not from Cocoon itself. The cache can store the complete document and request it from Cocoon only if it has expired. Cocoon then generates the new document and serves it to the cache to be stored. Look into how you can control the expiration of generated documents using the appropriate HTTP headers in your documents. There are several ways of doing this. For example, the Cocoon reader component allows you to set HTTP headers. Another way is to write your own component, such as an action that sets headers when used in a pipeline.

If you are accessing an external data source that is too slow, you might need to implement a backside cache. This type of cache sits between Cocoon and the external data source. The pipeline requests the data from the cache, not from the data source itself. There are various ways of implementing the cache. You can look at the description of how Cocoon caches pipelines to get some ideas on how to implement your own.

It is a good idea to provide the user with some visual feedback to show what is going on. If the user cannot see anything happening on the screen, he will perceive system performance as being too slow, even though it might not be. One way of doing this is to load an intermediate page that says something like “Please wait; your data is being fetched” and then let this page call the function on the server that does this. Presenting

the user with something to read while the work goes on in the background means that by the time the user has finished reading, part or all of the data will have been retrieved. Look into redirects and metatags to do this if you are building a site in HTML.

When designing HTML web sites, one of the mechanisms used most often is frames. Although this is not a book on magical HTML design, here's a piece of advice: Remember that each part of a frame causes a new request to be sent to the server. So if you have a page containing four different parts (header, footer, navigation, and actual content), that is a total of five requests to the server and five pipeline calls in Cocoon. Try to reduce the use of frames if possible. One way is by using Cocoon's content aggregation to aggregate the different parts of a page and then use a stylesheet to format the output.

In addition to the tips just discussed, there are additional areas you will want to check when you design the output format—which brings us to presentation.

Presentation

Most applications have some form of presentation. Because presentation in Cocoon is done using XSL stylesheets, you need a working knowledge of this technology to be able to author your presentation. You will also want to look at tools that help you author stylesheets.

One of the major steps is deciding what presentation format you need. Of course, the advantage of Cocoon is that you can add further types of presentations by adding stylesheets as you need them. However, this should not keep you from planning your presentation carefully.

Decide whether you want to support each client application (such as the different browsers) individually or whether you want to go for a format that suits both. Be aware that by the time you have finished your application, a yet-unknown browser might be the market leader.

Design your presentation for speed. This point is not necessarily limited to Cocoon applications, but it is worth stressing. If you plan on presenting your data in HTML, make sure you follow the guidelines as to how you should construct HTML pages for maximum speed when you author your stylesheets. This can depend on the browser type, so refer to available information on this subject.

Make sure you follow the Cocoon paradigm of separating concerns. Even though Cocoon offers you ways of splitting layout and content, it does not force you to. We have seen Cocoon applications built where XHTML was used as the format for the data. Although this might seem like a good idea to start with, after all, XHTML is an XML format. Imagine trying to then provide a presentation layer in WML. As mentioned in Chapter 2, "Building the Machine Web with XML," extracting the actual data from a format like XHTML is quite difficult.

Decide whether your presentation is static or whether it offers personalization of some sort. Check out the later section "Portals" for more information on using personalization to influence the output of your application.

Think about seasonal changes to your presentation. Make your application interesting by making small changes to the web site's appearance, depending on the current season. For example, you could give your site a Christmas feeling during November and December. Write a component such as a selector that provides you with this information.

If you already have HTML pages that you want to reuse in your Cocoon application, this is also possible. You would use the HTML generator to read the HTML and then have a stylesheet format the XHTML into the format you require. This is a way of easing the migration path to a complete XML/XSL-based solution. Another way of migrating is to have the Cocoon solution run in parallel to the application you already have. Cocoon can then generate parts of your site for you. Any new HTML pages can be authored using stylesheets, and the existing site can be served as before.

Even though you might have authored your HTML documents using stylesheets, there will be times when you need to include technologies such as JavaScript in your pages. Another technology that is often used with HTML is Cascading Style Sheets (CSS). CSS is often used to achieve dynamic look-and-feel changes on HTML pages. All of this can be used (or reused) in a Cocoon environment. The site map must be configured to allow the JavaScript (.js) files and the CSS (.css) files to be served through Cocoon. Look into using a reader to do this. Alternatively, these files can be served directly from the web server.

It's possible to use other technologies inside your web pages in the same way. You can use Java applets inside web pages by using the appropriate tags to include them inside the generated HTML pages. Just make sure your .jar file can be served either through Cocoon or directly.

While someone is working on the presentation side of the application, someone else can be defining the content.

Know Your Content

We have already mentioned that the XML parser used in Cocoon can validate the XML data it parses. It can do this using DTDs or XML Schemas. When building the application, you will probably not yet have a DTD for all your data. This means that you cannot use XML validation in Cocoon, because you can only activate it for all the documents, not for an individual one. Even if you do not use the parser to validate the data, you should document your XML using either a DTD or an XML Schema before moving the application into a production environment. (Of course, the earlier the data's format is documented, the better.)

As more and more XML tools come onto the market, they begin to offer advanced features such as automatically validating the data you enter into, say, an editor. Now, suppose you have a Cocoon-based system and have authors who are writing content for that system. Often, they will use third-party tools to do this and then upload the content to the system or deploy it through some other means (perhaps saving it to a database). Obviously this is ideal if you can provide these authors with a DTD of the data. They can then use the DTD inside their editing program, and you know that the data they submit will be in a format you expect and have written stylesheets for.

While the designers are working on developing the stylesheets that will present the data, that data also needs to be defined and documented.

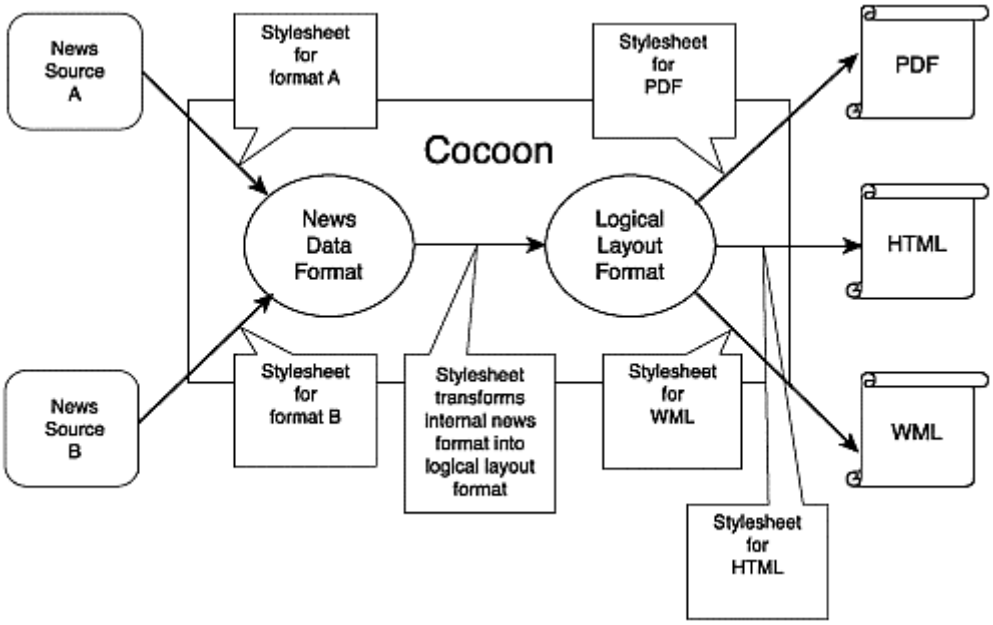
Document Your Data Sources

We talked briefly about external data sources when we discussed application performance. However, other factors also need to be taken into account when data is obtained from an external provider, such as a news feed.

Obviously, the most important fact is that you know exactly what format the data will be in. The best way to achieve this is if the data's format is documented in some way, such as in a DTD. You read about the various ways of documenting XML data in Chapter 2. It is an enormous advantage if your provider can send you the data in a standardized format. This becomes a great time-saver if you have to integrate several sources and they all can provide the data in the same format. It will then be possible to reuse the stylesheets. This is true of the news providers we looked at when building the Cocoon news portal in this book. Because the news is provided in RSS format, you could use the same stylesheet for several different feeds.

When designing the flow of data through your application, you need to consider two important points. The first point is the internal data definition. As shown in Figure 11.1, this is the format of the news data in your application. Every external data format needs to be converted into this format, so you need a stylesheet for every data source. Obviously it makes sense to choose a standardized format as your own internal format. This reduces the number of transformations you need, because not every external source that already supports your internal format needs a stylesheet transformation.

Figure 11.1. Format transitions using stylesheets.



The next step is to define a logical layout format. News data is not normally structured for presentation, so you need to think about defining a format that allows transformations into the end format, such as HTML or PDF. If your application is not limited to publishing just news data, but it also publishes other types of information, you will want to look into defining a logical layout format that is not data-specific. This lets you easily publish different types of data using the same stylesheets.

If you opt to use a standard format such as WML or XHTML as your logical layout format, make sure you will still be able to convert this format into a different layout, as shown on the right side of Figure 11.1.

This concept leaves you with three different transition areas:

- Incoming data must be transformed into your news data format.
- The news format must then be transformed into the logical layout format.
- The last area of transformation is into the regular output format.

Check to see whether your data source is always online. Nothing is more embar-rassing than finding out that your news provider is online only during the day when your news portal crashes the first night. Use appropriate selectors in the pipeline to ensure that you access the online server during the day and perhaps a database repository at night.

Make sure you can obtain the data you need with the least number of requests possible. We have seen a Cocoon-based application built to present stock information in which one block of information (such as an overview page) required the middleware solution to perform more than 20 requests against the data provider. Even worse, most of these requests had to be sent in order, because they were dependent on each other. The problem isn't that this can't be done with Cocoon—it can. But if you remember the earlier tips on performance, perhaps you will see why this point is worth stressing.

After you've defined the functions your system should have, the layout you want to present to the user, and the data format that is to be the core of your application, you need to look at the Cocoon components you can use to do all this.

Different Technologies

As mentioned at the beginning of this chapter, Cocoon provides many ways of solving certain problems. People new to Cocoon are sometimes overwhelmed by the many possibilities. Often, only one type of component is used to solve a problem when perhaps a different solution would have been better. As an example, when starting out with Cocoon, we often found ourselves writing new transformers when it would have been better to use an action or selector instead.

Here are some tips on when to use what:

- Using a given component is better than writing your own.
- Use generators when you have an identifiable data source that can be used as the starting point for your pipeline.
- Use transformers when you need to manipulate the XML data flowing through the pipeline.
- Use actions and selectors to influence the pipeline if their results do not need to manipulate the output document.
- Use an action if you want to execute a task that does not influence the XML processing pipeline.
- Use a selector if you want to choose between different processing pipelines.
- Use XSP for rapid development of a custom generator, and transform it later into a real generator.

This section has looked at a few aspects that are important when you design your Cocoon application. Performance is probably the key factor when the application is actually finished and installed. A well-thought-out concept is a necessary starting point for good design. "Program now; think later" is, in our opinion, not the way to build Cocoon applications. Unfortunately, even writing a great concept beforehand still might not prevent problems from occurring.

Solving Problems

So, you've written the concept, designed the architecture, written any needed components, and built the pipelines—and things still don't work as you expected. Here is a two-sentence answer to this problem:

Someone else has already solved your problem! All you need to do is find that person and solution.

Sounds simple, doesn't it? But for many cases, this is true. Problem solving has become easier with the Internet. When we first started using Usenet newsgroups (which were

exchanged using UUCP back in those days), we could post our problems—not just to our colleagues in Paderborn, Germany, but to the whole world! And the Internet has expanded this "knowledge base" so that now it is very probable that someone out there has already had the same problem you are trying to solve.

The Cocoon web site is a good starting place for finding information and help. There you can find mailing lists and archives of past list discussions. Chances are your question is there somewhere. Subscribe to the mailing lists and join the Cocoon community. Appendix C, "Links on the Web," lists links for the Cocoon web site.

Search engines are also a good choice when you are looking for a solution to your problem. However, if you query a search engine, you probably will be swamped with thousands of answers that don't really help. If you already know roughly the area your question applies to, perhaps checking one of the newsgroups is a better way to go. There are newsgroups for most of the subjects in this book, such as XML and XSL. However, there is as yet no newsgroup for Cocoon. Hopefully, you will be able to solve any problem that might arise using one of the listed methods.

Using the information discussed so far should allow you to complete your application concept and design the architecture of your solution, complete with the required Cocoon technologies. Even though most people who look at Cocoon and read this book will already have an exact idea of the type of application they want to build, it is always a good idea to see how other people are using the technology. The following examples might provide some additional ideas for the types of applications you can build with Cocoon.

Different Types of Applications

Cocoon lends itself to being used to build a variety of solutions. Although Cocoon is aimed primarily at the XML publishing sector, adding your own components lets you expand Cocoon into a complete middleware architecture.

In the past we have worked on building a commercial solution that provides additional (and sometimes customer-specific) components needed to provide a complete solution. We added components and functionality to Cocoon without throwing away a single Cocoon concept. This shows the extensibility of the architecture.

To give you some idea of what perhaps you can do to solve a specific problem, here are some of the extensions we have written to provide the various solutions we have built with Cocoon:

- Components for authentication and user administration
- Portal framework components
- A complete XML/XSL-based content management system
- Integration components for a commercial XML database
- System management components

Although these components were not written as part of the Cocoon project, some of them will find their way back into Cocoon and hopefully will be available in the not-too-distant future.

Using Cocoon and the additional components allows you to build applications such as portals, flexible publishing systems, and web sites. Because Cocoon can process XML data, you can also build solutions that can receive complete XML documents as input and process them using pipelines.

Let's look at some of these application types in more detail. The most common Internet application is the web site, where information is published as HTML. This type of application becomes more complex to develop when the information is stored in external systems such as databases and when additional formats such as PDF are required. The web site needs to be extended into a network publishing application to provide these advanced capabilities. When several different types of users are accessing the system, some form of personalization is called for. The term *portal* is often used to describe this type of application. This chapter concludes with a look at how to use Cocoon to build portals.

Using Cocoon to Build Web Sites

One of the most common uses of Cocoon is as a system for building web sites. After all, that is its main function. Many web sites already use Cocoon; they are listed on the Cocoon web site. We discussed a web site example earlier in this chapter. Now we will add to the information that was discussed there.

Remember that Cocoon organizes a web site's content using a sitemap. Although it is possible to define a pipeline for each document your web site will serve, this would result in a sitemap that becomes very hard to maintain. Therefore, you need to define pipelines that can handle similar types of content, perhaps split into different areas. Look into how you can use wildcards in the sitemap as a method of combining several documents into one pipeline.

Make sure the layout developer (the author of the stylesheets) uses a tool that can perform XSL transformations on some sample data for that format. You should provide the author with sample data to use. It will be easier for him to test individual stylesheets this way instead of having to use Cocoon each time.

Another important point is to make sure the layout deployers use a tool that either already uses the Xalan XSLT component or that lets you use it additionally. If the tool allows a version of Xalan to be used, make sure you use the same version as the one in the Cocoon you will be running. Which tool is best suited for the job depends largely on exactly who will be using it and for what purpose. We have provided a list of relevant links to tools in Appendix C.

Although your first-version web site might only read its content from XML files and publish to a single format such as HTML, one day you will want to use something more advanced to store your data, such as a database. You might also need to integrate external systems such as mainframes into your application. In addition, there might be demand for additional formats as users use devices such as mobile phones to access your solution. The web site must therefore be extended into a network publishing application.

Network Publishing Applications

Although this is only a different way of defining something, we use the term *publishing application* to emphasize that the data you want to display is actually stored somewhere, and we don't mean in a file. A publishing system might generate reports from data that is obtained from a database, for example. It then might manipulate the data in some way, perhaps to generate different views and then publish that data in one or more formats.

Areas you will want to look into include the Cocoon components that allow you to access data from a database or external systems such as a remote XML server via HTTP. You will also want to learn more about standards such as XSL:FO. After it is formatted this way, your data can be laid out in different output formats, such as PDF or PostScript.

Publishing systems might be the first time you need to publish data that is dependent on the type of end device. For example, you could allow mobile phone users to access only the most important information while allowing browser users to access the full beauty of your web site.

In our experience, using Cocoon as a publishing system for specific data is an ideal way to introduce the technology into a new area. Applications such as a report generator, which reads data from a database, consolidates it, and then presents that information in HTML and PDF, can be built in an isolated fashion that does not intrude on given software structures. The first little application we built with Cocoon was a front end to an internal database we had at that time containing work reports. The solution read the data from the database dependent on a query parameter and then presented an overview of the data in the various formats. As a prototype showing what could be done with Cocoon and how flexible it was, this was an ideal solution.

Publishing systems might be the first time you also need to integrate something like user authentication and personalization—allowing only certain people to access the data. This brings us to the next application form—the portal.

Portals

Although you probably think of something like myYahoo or myAOL when the term *portal* is used, portals can actually be a lot simpler. We refer to this type of application whenever some form of user authentication is necessary to access information or when information can be individually personalized. This personalization can range from

changing the color of a single document to configuring external news sources in a news portal.

In our portal example, built over several chapters, we have already seen how it is possible to build a portal using Cocoon. Nevertheless, and because we know that some readers might jump right to this section, we will go over some of the main points again and in a more general context.

In order for personalization to be possible, we need to be able to recognize the user when he accesses the portal. Most portals require some form of authentication, such as entering a user ID and password. This data is then matched against a repository, such as a database, and the user is rejected if there is no match. Each user therefore requires an entry in the database, and the application perhaps also needs to cater to an anonymous user (a user without a login). After the user is authenticated, the application will want to allow the user to access the different areas in the portal without having to log in again. Look into ways of creating a session when running inside a servlet engine in order to do this. It will also be necessary to recognize a returning portal user so that he does not have to log in each time he accesses some part of the portal. An appropriate action component can solve this problem.

Another important step is to define the portal structure. What information will be available to the user after he has logged in? Will each user have an individual profile, or will the portal cater to only specific groups of users? As soon as this has been decided, a suitable XML format for the profiles can be defined. The profile should then contain information relevant to the personalization (such as colors) or to the individual preferences in regard to the types of information to be displayed.

Therefore, the first step of building the portal is to define where the user data and the portal profile are to be stored. Then the application needs to define and set up a pipeline in Cocoon for the authentication. One way of doing this is to have an HTML form send the user ID and password to Cocoon and then use the `sql_transformer` to select the user and profile from the database.

If the portal profile contains data on the types of information that are to be displayed, this information must be fetched and integrated into the profile so that it is complete before it reaches the stylesheet. Look into using content aggregation as a way of doing this. Each different data source will then return information that is added to the user’s profile, so that the end result will be a complete portal in XML.

After the profile has been selected and all the data fetched from the various sources, the complete profile can then be transformed into a specific look and feel using a stylesheet. The stylesheet can access specific details contained in the individual profile and format the output as necessary.

If the personalization is based on the user who accesses the site, you need to define what types of information the user can change and how the presentation should be affected by,

say, his age. If you will be providing a different layout for teenagers than for middle-aged people, you will need to define the criteria by which this can be decided. Writing a new component such as a selector is an ideal way of doing this.

Think about whether you want to change the presentation dependent on other factors, such as the time of day or the weather. Say you are building a stock-quote portal and you present the current market chart (say NASDAQ) on your front page. After the NASDAQ closes for the day, it might be a good idea to present a different chart, such as from Asia. So if you want to switch content and presentation dependent on the time of day, look into the Cocoon selector component as a way of doing this.

If you are thinking about building a late-night portal, in which the presentation changes after a certain hour, remember that your user might be living in a different time zone, so it might be the middle of the day for him when you select the late-night presentation.

Summary

This completes this chapter on Cocoon application design. As we said at the beginning, you can build many different types of applications with the current version of Cocoon. Although Cocoon's main focus currently is on web sites, as more components are built that integrate into the Cocoon architecture, it will expand and become a platform for other types of applications as well.

This is one of the great advantages of using Cocoon as a base for XML applications. Because of the way new components can be easily added, there is really no limit as to how you can use Cocoon as the platform for your solution. As an open-source project, it has much support from individuals and companies. Several firms have donated components to the Cocoon project and in so doing have helped the software become better suited for application scenarios such as the network publishing system and portal described in this chapter. The next chapter outlines some of the directions Cocoon might go in as XML and XML applications become more widespread. It also provides some additional ideas as to where Cocoon can be used—perhaps in your particular environment.

Chapter 12. Cocoon: Weaving the Future

When we conceived this book on Cocoon, we decided that we would write this chapter last. This might seem obvious, because it *is* the last chapter in the book, but chapters are not necessarily written in the order they appear. This chapter would give us a chance to fold all the developments that happened while we were writing the book into a chapter that would also focus on what Cocoon's future might look like.

As is the case with writing a book, software development does not happen in a void. It is a very dynamic process. This is especially true of open-source projects that are developed under widespread public attention. Because of this, the Cocoon project is directly influenced by new technologies or by other projects that are being developed in close proximity. Many of the developers who work on the Cocoon project are also working on related projects such as Apache Avalon, so there is a lot of influence between the different projects.

Cocoon is based on several other Apache projects and focuses on technologies that are playing an increasing role in today's application architectures. As additional components are developed that extend the code base, Cocoon will become suitable for building applications such as portals or content-management solutions.

Cocoon's popularity is growing rapidly as we write this. The increased use of the platform is leading to additional requirements and a growing developer community. With additional components that have been donated to the project or developed by community members, Cocoon can provide far more than just a framework for publishing. A large development effort is currently going into adding increased support for building web applications. These additions range from support for team development to simplifying everyday tasks such as form handling to increasing scalability and performance. In fact, by the time you read this, there might already be a version of Cocoon available that incorporates some of the changes we have listed here.

The Evolving Cocoon Architecture

The Cocoon architecture changed abruptly from version 1.x when the current 2.0 version was designed. The first version of Cocoon, started in 1998, was based on the DOM

processing model. The XML files (the content) contained processing instructions that Cocoon used to apply the XSLT stylesheets.

As you saw when we looked at Cocoon from a user perspective, the processing flow is now defined by the pipelines in the centralized configuration file, the sitemap. The XML processing now takes place using SAX, so memory consumption is much lower than it was in the first version.

The sitemap is an XML file that is compiled into a Java class every time a change is detected. To be more exact, first a stylesheet is applied that produces a Java source. Then that class is compiled to produce what is loaded and processed by Cocoon to find the correct pipeline for, say, a request.

Many developments, such as the transformation of the XML configuration file, are constantly being refined in the Cocoon project. The project's openness allows alternatives to be discussed and then perhaps implemented if the project members reach a consensus. In turn, this helps Cocoon evolve as new technologies become available or new ideas emerge.

Some of the ideas being discussed, and, in part, already implemented in the current code base, are a way of interpreting the sitemap (instead of compiling it), adding advanced form handling, allowing actions to be more flexible, and extending how pipelines can be configured.

Interpreted Sitemap

The current way of compiling the sitemap into a Java class using a stylesheet has several drawbacks:

- The complex stylesheet that transforms the XML file into the Java class is now more than 80KB in size.
- Adding new features or finding bugs has become very difficult.
- Stylesheet transformation is slow compared to other solutions.
- This way of creating Java source and then compiling it requires a Java compiler that can be used from within Java code.

The last point is a potential risk for Cocoon, because currently there is no standard for using the compiler in this way, and several servlet engines have some class path problems. In addition, this way of using the Java compiler is deprecated in Sun JDK 1.4, so eventually it might even be removed altogether.

It was for these reasons that the concept of an interpreted sitemap was put forward as an alternative to the transformation and compilation model. In the new model, the sitemap is interpreted, and an object tree of Java objects is generated. Each statement or tag in the sitemap is then represented by an object. Each object knows what to do and when a

request enters Cocoon. This request is forwarded to the object tree that then performs the required task.

Advanced Action-Sets and Multiactions

Currently, actions can expose only a single functionality per Java class. This means that if you need several functions that might be best implemented as actions, you need a separate action for each function. This concept can cause a large number of actions to be implemented, where it would perhaps be better if a single action could expose several functions.

If you look at the actions you used in Chapter 7, "Cocoon News Portal: Extended Version," you can see that you had an action to insert data into the database and an action to delete data. However, this would be easier to maintain if you had a single database action that could provide a function for updating and a function for deleting data.

The concept of a "multiaction" has therefore been proposed. An `action` class contains more than one method, each of which implements one specific action. The current `act` method decides by looking at the value of a `request` parameter which real method should be called. So the multiaction acts as a dispatcher.

Apart from extending the action in this way, another proposed change is to provide an alternative to the `cocoon-action` parameter. As you have seen, the action selected from an action-set is dependent on the value of the parameter `cocoon-action`. When using several Submit buttons in a form, you must assign each button the name `cocoon-action` with a value of the action that is to be called. However, this means that the button's text is always the same as the configured value. This can cause several problems, especially when you're designing layouts for different languages.

A proposed alternative is to allow the Submit button to be called `cocoon-action-` followed by a postfix containing the actual value. An example would be `cocoon-action-add`. Cocoon would then interpret the name and not the value, avoiding the problems that exist currently.

Form Handling

A common task when writing a web application is form handling. The web application presents one or more documents in which the user of the application can type in some information and send it back to the server, where the data is processed. You saw examples of this when you built the extended version of the news portal in Chapter 7.

Due to the request-response cycle that web applications are based on, developing a form consists of two main steps.

The first step is developing the form that is presented to the user. This form is generated by the first request to the application. When the user submits the information to the web

application, a second request is sent that contains all the entered data. A typical example is the login page, where the user has to enter an ID and password.

The second main task for form handling is the extraction of the submitted values from the request and the validation by additional application logic.

Developing a form is therefore an error-prone task, because the developer has to synchronize two documents—one that generates the form, and one that validates and processes the submitted values.

Because form handling is a substantial part of a web application, currently several discussions and development efforts are ongoing—not only in the Cocoon community, but also in other open-source projects. They all have more or less the same goal: simplifying the development of forms and supporting the validation of data. The main idea is to define the data that should be editable in the form, together with the validation rules. This description is then used twice: to generate the form and to validate and store the submitted information.

With additions such as form handling, developing complex web applications with Cocoon will be a lot easier than it is today. And to be honest, this is one of the additions *we* are most interested in, because it will leverage Cocoon into areas where today many commercial products are used.

Modularization

Another development effort currently being discussed is to support the modularization of web applications. Using the version of Cocoon we describe in this book, it is rather difficult to coordinate the team development of a Cocoon-based application. This is due to the fact that some resources such as component configuration are maintained in a single configuration file.

The new concept of “blocks” is heading in the direction of modularization. As soon as this concept is implemented, it will be possible to develop several modules (or blocks) that can then be put together to build a complete web application. Each developer can define, develop, and configure each block individually.

This concept will also allow the reuse of parts of a web application. A generic solution can then be developed as a block. This block can be reused in other web applications, because it can be deployed with all the information it requires, such as configuration data.

Another advantage of the block concept is that every block can have its own description containing information on how the block is to be accessed and what data it requires. Having a description of the block available will allow tools to be built that can check for consistency between blocks. In addition, it will be easier to publish blocks—or collections of blocks—as web services, accessible via protocols such as SOAP or XML-RPC.

Pipeline Configuration

In a way, pipelines can themselves be considered components. However, the current version of Cocoon does not provide very much in the way of flexible configuration for these components. Take caching as an example. At the moment, it is only possible to turn on caching for all the pipelines or for none at all. However, it would be better if it were possible to override this configuration on a per-pipeline basis.

An administrator who designs the application by editing pipelines into the sitemap will know that there is no need for the extra caching overhead for individual pipelines, such as those that serve static documents or images. Pipelines that produce a very dynamic output or that use components where caching is not possible are other candidates for a parameter that defines caching on a per-pipeline basis.

In a web site scenario, it is quite common for a front-side cache to be placed in front of Cocoon. This cache then stores the generated pages (such as HTML documents) and can serve them to the client application faster than if they were generated dynamically. However, if it were possible to add an `expires` parameter to a pipeline, this would allow the validity of storage in the front-side cache to be defined individually for each pipeline.

As these examples show, Cocoon development takes place in a very dynamic surrounding. In part, this is because Cocoon can be used for many different types of applications, and it is not limited to only being a platform for XML publishing. We will now take a look at some of the scenarios and see how additional components are making it easier to use Cocoon to build these types of applications.

Cocoon Usage Scenarios

Even using the components that the standard Cocoon distribution provides, you have seen how to build different types of applications ranging from a picture gallery to a full-blown news portal using a database and additional components.

Although much of what is needed to build these types of applications is available, life would be easier for an application developer if more components were available for such things as authentication and if a portal concept were embedded into Cocoon that allowed for such things as user-enabled customization or embedded content (sometimes called a *portlet*).

The good news is that there is. Although they are not yet contained in a released version, additional components were recently donated to the Cocoon project that will allow this to happen. This chapter provides a brief overview of the areas these components cater to and why they are important if you want to use Cocoon to do more than just publish data.

Because these additions have just made it into the code base, we refer you to the Cocoon web site for additional information. In Chapter 3, “Getting Started with Cocoon,” we told you how you can obtain the newest version of Cocoon from the Cocoon web site.

Authenticating the User

When building an application that is accessed by different users in various roles, you must be able to define which pipelines can be accessed by different users. To do this, you need a way of authenticating the users against user storage such as a database.

In order to integrate the authentication into Cocoon, you need components that perform the following jobs:

- When a particular pipeline is accessed, check whether the user has been authenticated.
- If the user has not yet been authenticated, provide a login page.
- Authenticate the user's data (for example, ID and password) against a database or other storage.
- Protect access to pipelines dependent on the authentication and user's role.

Luckily, Cocoon has components you can use to perform various tasks in this list. For example, you can already use the sql transformer to authenticate a user against a database, as you saw in Chapter 7.

The authentication components that were added to the current (not yet released) code base provide a complete framework for authentication and pipeline protection.

Using authentication in your Cocoon application allows you to build solutions such as portals. You will now see what an XML-based portal with Cocoon looks like.

The Cocoon Portal

The previous chapters in this book looked at different ways of building a portal. You have seen what is available in Cocoon that allows you to do this.

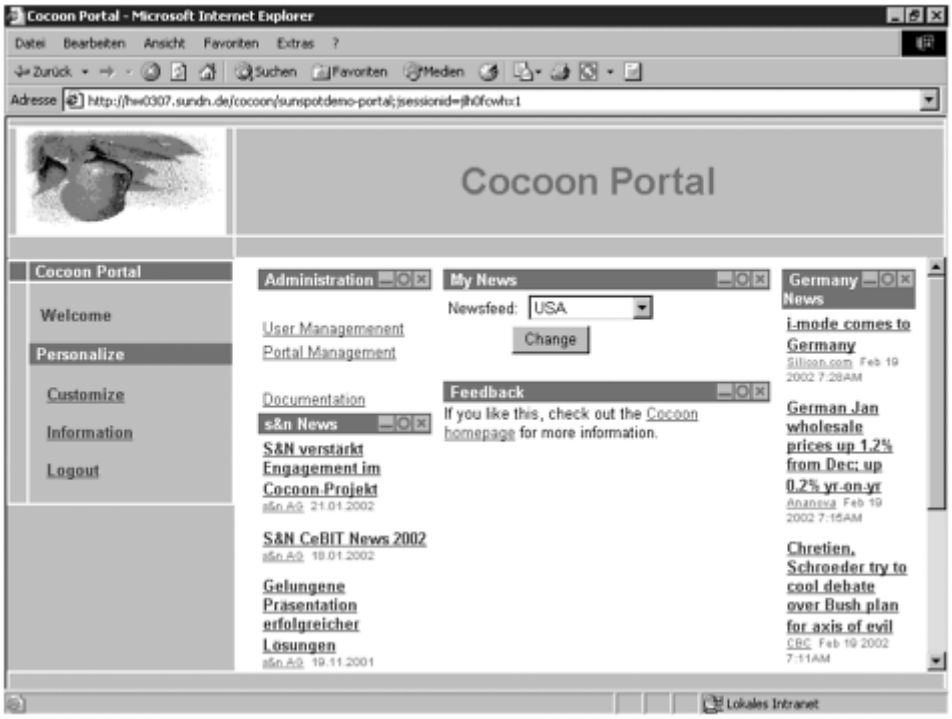
When you compare the news portal you built over three chapters to commercial portals that are available, it quickly becomes clear that a few things are still missing from making your solution an alternative to other offerings.

What requirements would you need from a complete portal based on Cocoon? Let's take a look at some of them. First, you would need an XML-based portal definition. This means that the portal administrator should be able to define the portal profile using XML. It should also be possible to define different portals for various groups of users. A user who belongs to the Management group should be able to see different information than a user who belongs to the Trainees group. In order to be able to provide different "blocks" of information, you would need a means of incorporating some notion of a portlet. Typically, this term is used to describe a block of information, such as a particular news feed integrated into the main portal page.

Cocoon already provides the means to access the different information feeds or data sources. It also provides much of the framework that is needed for a portal solution. The few components missing from the current release would have to be developed for each individual project.

As we mentioned at the beginning of this chapter, future Cocoon components will provide a more-advanced framework for portals than what we have shown in this book. Figure 12.1 shows how a portal based on these new components could look.

Figure 12.1. A Cocoon portal using additional components.



Portals based on Cocoon might be an area in which we will see a lot of growth over the next few months and, hopefully, years.

Other usage scenarios that also appear a lot on the Cocoon mailing list are the integration of XML databases and using Cocoon as a front end to content management systems (CMSs).

Integrating XML Databases

Native XML databases are becoming popular as a way of easily storing XML documents. Because Cocoon is fully based on XML, it makes sense to store the data Cocoon needs in a database of this type. Using XML databases has several advantages over using traditional relational databases in an XML environment:

- There's no need to map from XML to RDBMS.

- It can be used to store XML data *and* stylesheets.
- XML documents stored in an XML database can be accessed and searched via XML concepts such as XPath.

Several native databases are currently available, both commercial and open-source. Integrating the XML database into Cocoon depends on the interfaces and protocols that the database provides.

One common form of accessing the database is via the HTTP protocol, which is already possible using Cocoon components such as the file generator. Another form of integration—and one we discussed in this book—is by way of a Cocoon protocol that can be used wherever it is needed. For example, this would allow stylesheets to be stored in the XML database and then accessed directly using the xslt transformer.

As XML databases become more widespread, we are sure to see more and more installations in which the data and stylesheets are stored in an XML database and where the publication is done by way of Cocoon. This tandem of data storage and publishing forms part of an application type that is commonly called a content management system (CMS).

Content Management System

The term "content management system" means different things to different people. However, put simply, a content management system allows controlled access to a storage system that contains data. This data can be actual content, such as the latest news, or layout data, such as a stylesheet.

The idea behind a content management system is that it separates the concerns involving the maintenance of applications such as web sites. A web site contains data and layout information. Each different type is normally maintained by a different person or set of people. A user who submits content to the site does not normally want to be concerned with how the data should look when it is published. The content management system should therefore control who can do what while logged on to the system. Also, all changes to the data and layout should be recorded by the content management system, and it should be possible to retrieve older versions in case they are needed.

Cocoon already offers some of the components that can be used to build content management systems. By integrating a storage system such as an XML database, it is already possible to build simple systems. But quite a few features are still missing. Things such as versioning, staging, and the possibility of defining a work flow (such as edit, test, publish) would still have to be developed as additional components.

Therefore, one common view in the Cocoon community is that a given CMS should be integrated into Cocoon (or vice versa, depending on how you look at it). This would combine the flexibility of Cocoon's XML publishing with the additional features required of a CMS.

Unraveling Cocoon

When we first came up with the concept of a Cocoon book, "Unraveling Cocoon" was to be the title. However, as the concept evolved and we discussed it with our publisher and technical editors, we decided against using this title for two reasons.

The first reason was that somebody mentioned that if you unravel something, you destroy it. And we certainly did not want to destroy Cocoon. Secondly, we thought this book should be more than just a developer's guide to Cocoon. We wanted to motivate readers by using XML technologies to build different types of applications and show how Cocoon can make this process easier by taking care of the hard work for you.

This last chapter has hopefully given you some idea of Cocoon's ongoing development. There are many good and innovative ideas in the community. As Cocoon becomes more popular as an XML platform, the community is also growing, and additional contributions are being made. By reading about some of these new ideas, you might think that Cocoon is not finished or that it will change dramatically in the near future.

But when is software ever finished? The answer, of course, is never. There is always room for improvement. But the current version of Cocoon is finished in the sense that it is stable and that it is being used today to develop complex web applications. The Cocoon developers ensure that each new feature is introduced in a compatible way, so the things we presented in this book will be valid for a long time.

Using open-source projects as a base for customized solutions is becoming increasingly popular, thanks to projects such as Cocoon. When we first developed solutions for our customers in the early 1990s, the customer was willing to pay for a one-off solution built from scratch. But as developers will tell you, times have changed—so much so, in fact, that now sometimes it is the customer who tells the solution developer which technology he should be using. Today, that technology is increasingly XML-based.

As we complete our journey through the open-source XML publishing framework, we hope that we have unraveled some of the mysteries and provided some insight into the "why" and not just the "how" of using Cocoon. We invite you to continue your journey—on the web and in the mailing lists that *are* the Cocoon community.

Appendix A. Cocoon Components

This appendix lists the sitemap components delivered with the Cocoon distribution that you are most likely to use when developing a Cocoon application. The complete distribution is included on the companion CD, so additional documentation on components not listed in this chapter can be found there.

For each sitemap component type, we have provided a reference table that gives a compact overview of all the components of this type. An in-depth description of many of the components can be found in this book.

Some components can be configured, either by adding a configuration in the components section of the sitemap or by specifying parameters in the pipelines section. The reference lists all the possibilities for each component and gives a short explanation.

At the end of this appendix, we have listed some additional configuration details on components contained in cocoon.xconf. However, changing the configuration of these components is not as common as perhaps configuring a used generator in a pipeline.

Table A.1. Generators

Name	Description
file (default)	Reads an XML document from a URI. **Cacheable**: Yes
html	Reads an HTML document from a URI and converts the HTML format to XHTML. **Cacheable**: Yes **Parameters**: *`copy-parameters`* (default `false`)—If this is set to `true`, the parameters contained in the request are appended to the URI that the HTML is fetched from. *`xpath`* (optional)—If this is present, only nodes conforming to this XPath expression are included.

directory	Creates an XML document from a directory in the filesystem. **Cacheable**: No **Parameters**: *depth* (optional)—Sets how deep the directory generator should delve into the directory structure. The default value is `1`. *dateFormat* (optional)—Sets the format for the date attribute of each node. *root* (optional)—The root pattern. *include* (optional)—A pattern describing the files to be included. *exclude* (optional)—A pattern specifying the files to be excluded.
imagedirectory	Creates an XML document from a directory in the filesystem. Adds width and height information to images. **Cacheable**: No **Parameters**: Same as for the directory generator.
request	Creates an XML document from the incoming request. The document then contains all the request parameters, the client's user agent, and the HTTP headers. **Cacheable**: No
status	Creates an XML document from available system information. **Cacheable**: No
serverpages	Creates a new generator from a script document, such as an XSP document. When instantiated, the serverpages generator compiles the script into a Java class that is then used in the pipeline. So basically, the new generator replaces the serverpages generator in the pipeline. **Cacheable**: No
script	Executes arbitrary scripts such as JavaScript. The source for this generator is a file containing statements in the scripting language. **Cacheable**: No
velocity	Uses the Velocity Template Engine to generate XML. **Cacheable**: No
jsp	Executes Java Server Pages (JSP) producing XML. **Cacheable**: No
stream	Reads XML from a parameter sent with the request.

	Cacheable: No **Parameter:** *form-name* (mandatory) The name of the request parameter containing the XML data.

Table A.2. Transformers

Name	Description
xslt (default)	Applies style sheet transformation to an XML document. **Cacheable:** Yes **Parameters/configuration:** *use-request-parameters* (default no)—Makes all request parameters available in the XSLT style sheet as global XSLT parameters. *use-browser-capabilities-db* (default no)—Makes all properties from the browser capability database available as global XSLT parameters. *use-session-info* (default no)—Makes information about the session available to the style sheet using global XSLT parameters. For example: Boolean values that indicate if a session is available, if it is valid, and so on. In addition, the session ID is accessible. *use-cookies* (default no)—Makes the cookies from the request available in the style sheet as global XSLT parameters.
log	Logs the SAX stream to a file. **Cacheable**: No **Parameters**: *Logfile* —The name of the log file. If this is not specified, the output is written to the standard output. *append* (default no)—If this is set to yes, the output is appended to the log file.
sql	Fetches data from a database. **Cacheable**: No **Parameters/configuration**: *use-connection* – The name of the configuration from cocoon.xconf to use. *Dburl* – The database URI if no connection is used.

	Username – The username for the database if no connection is used. *Password* – The password for the user of the database. *doc-element* (default `rowset`)—The name of the element surrounding the fetched data. *row-element* (default `row`)—The name of the element used for each fetched row. *namespace-uri* (default http://apache.org/cocoon/SQL/2.0)—The namespace used for the elements. *namespace-prefix* (default `sql`)—The namespace prefix used for the elements.
i18n	Internationalization transformer. **Cacheable**: No **Configuration**: *catalogue-name* (mandatory)—Base name of the message catalog. *catalogue-location* (mandatory)—Location of the message catalogs. *untranslated-text* (default `untranslated-text`)—Default text used for untranslated keys.
xinclude	Includes external documents according to the XInclude specification. **Cacheable**: No
cinclude	Includes external documents according to the Cocoon Include specification. **Cacheable**: No
filter	The filter transformer can be used to let only a certain number of elements through in a given block. **Cacheable**: No **Parameters**: *element-name* (mandatory)—The name of the element to filter. *count* (default `10`)—Each block has this number of elements after it is filtered. *blocknr* (default `1`)—The number of blocks in the resulting XML document.
writeDOMsession	Makes a DOM object from SAX events and writes it to the session. **Cacheable**: No

	Parameters: *dom-name* (mandatory)—The unique name of the DOM in the session. *dom-root-element* (mandatory)—The name of the root element contained in the XML document. The whole tree with this root element is stored in the session.
readDOMsession	With this transformer, a DOM object that is stored in the session can be inserted at a given position. **Cacheable**: No **Parameters**: *dom-name* (mandatory)—The unique name of the DOM object in the session. *trigger-element* (mandatory)—If this element is found in the XML document, the DOM from the session is appended. *position* (default `in`)—Indicates the actual place where the DOM will be inserted[md]before, after, or in the trigger element.

Table A.3. Serializers

Name	Description
html (default)	Converts XHTML to HTML. **Cacheable:** Yes **Mime-type:** text/html **Configuration:** *doctype-public* —Sets the documentation for the document type. *doctype-system* —Specifies the public identifier to be used in the document type declaration. *encoding* —Specifies the preferred character encoding. *indent* —Specifies whether the transformer may add additional white space when outputting the result tree. *omit-xml-declaration* —Specifies whether the XSLT processor should output an XML declaration. The value must be `yes` or `no` *standalone* —Specifies whether the transformer should output a standalone document declaration. The value must be `yes` or `no`.
xml	Outputs XML.

	Cacheable: Yes **Mime-type**: text/xml **Configuration**: Same as the html serializer.
wap	Outputs WML. **Cacheable**: Yes **Mime-type**: text/vnd.wap.wml **Configuration**: Same as the html serializer.
vrml	Outputs text containing VRML. **Cacheable**: Yes **Mime-type**: model/vrml **Configuration**: Same as the html serializer.
svgxml	Outputs XML containing SVG. **Cacheable**: Yes **Mime-type**: image/svg-xml **Configuration**: Same as the html serializer.
links	Extracts all links and references and outputs them as text. **Cacheable**: No
fo2ps	Outputs postscript from XSL:FO. **Cacheable**: Yes **Mime-type**: application/postscript
fo2pdf	Outputs PDF from XSL:FO. **Cacheable**: Yes **Mime-type**: application/pdf
fo2pcl	Outputs PCL from XSL:FO.

	Cacheable: Yes **Mime-type**: application/vnd.hp-PCL
svg2jpeg	Creates a binary image from an SVG description. **Cacheable**: Yes **Mime-type**: image/jpeg
svg2png	Creates a binary image from an SVG description. **Cacheable**: Yes **Mime-type**: image/png

Table A.4. Readers

Name	Description
resource (default)	Reads data from a URI. **Cacheable:** Yes **Parameter:** `expires` (optional)—When specified, this determines how long in milliseconds the document can be cached by any proxy or browser.

Table A.5. Matchers

Name	Description
wildcard (default)	Matches the request URI against a pattern.
regexp	Matches the request URI against a regular expression.
request	Matches based on a request parameter if it exists. The source of the match directive indicates the parameter's name.
sessionstate	Matches based on session attributes. The value of a session attribute is matched against the pattern. **Configuration/parameter**: `attribute-name` (mandatory)—The name of the session attribute to match against.
next-page	Matches based on the value of a request parameter. The value of the request parameter is matched against the pattern. **Configuration/parameter**: `parameter-name` (mandatory)—The name of the request parameter to match against.
referer-match	Matches based on the value of a request header. **Configuration:**

	header-name (mandatory)—The name of the header to match against.

Table A.6. Selectors

Name	Description
browser (default)	Tests the client's user agent against a given value. **Configuration (mandatory):** For each possible user agent, an identifier can be set that can be used in the following tests: <browser name="explorer" useragent="MSIE"/>
parameter	Tests the value of a sitemap parameter. **Parameter**: *parameter-selector-test* (mandatory)—The name of the sitemap parameter to test, such as `{1}`.

Table A.7. Actions

Name	Description
locale	Makes the request's locale information (language, country, variant) available to the sitemap/pipeline.
request	Makes some request details available to the sitemap: `{context}` is the servlet's context path. `{requestURI}` is the requested URI without parameters. `{requestQuery}` is the query string, as in `"?param1=test"`. Additionally, all request parameters can be made available for use in the sitemap if the parameter *parameters* is set to `true`. A variable is created for each request parameter (only if it doesn't already exist) with the same name as the parameter itself. Default values can be set for request parameters by including a sitemap parameter named `default.`*parameter-name*.
form-validator	The action used to validate request parameters. The parameters are described via an external XML.
session-isvalid	Checks to see if a session exists and if the current session is still valid.
resource-exists	Tests if a resource exists. **Parameter:**

	url (mandatory)—The URI of the resource to test.

Common Components in cocoon.xconf

The final section of this appendix contains a brief overview of the most important components and their configuration in cocoon.xconf. The most important ones are the parser, the stream pipeline, and the event pipeline. For each component, we give a brief description and show the implementations to choose from.

In addition, we show some of the possible extensions of Cocoon, such as adding protocols, adding database connections, and XSP logic sheets. Each entry contains an example showing how the configuration in cocoon.xconf should look.

The Parser

The parser component is used to parse all XML documents. The default parser is `org.apache.cocoon.components.parser.JaxpParser`. This parser uses the JAXP 1.1 API. If the JAXP version installed does not conform to 1.1, `org.apache.cocoon.com-ponents.parser.XercesParser` should be used instead. Because the parser is also used to parse cocoon.xconf itself, the following system property should be set when the servlet engine is started:
`org.apache.cocoon.components.parser.Parser=org.apache.cocoon.components.parser.XercesParser`.

- **`org.apache.cocoon.components.parser.JaxpParser`** (default)

  A JAXP 1.1-compliant parser.

  Example:

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

- **`org.apache.cocoon.components.parser.XercesParser`**

  The Apache Xerces parser.

  Example:

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

The Stream Pipeline

The stream pipeline is a component used by the Cocoon processor to represent a processing pipeline consisting of either a reader or an event pipeline in combination with

a serializer. The stream pipeline produces the character stream that is sent back to the client.

Currently there are two alternatives: the caching and the noncaching stream pipeline.

- **`org.apache.cocoon.components.pipeline.CachingStreamPipeline`** (default)

  The result is cached for further requests (if possible).

  Example:

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

- **`org.apache.cocoon.components.pipeline.NonCachingStreamPipeline`**

  The result of the request is never cached.

  Example:

```
<stream-pipeline
     class="org.apache.cocoon.components.pipeline.NonCachingStreamPipeli
     ne"/>
```

The Event Pipeline

The event pipeline is a component used by the Cocoon processor to represent an XML processing pipeline consisting of a generator and possibly some transformers. The event pipeline produces SAX events that are consumed by the stream pipeline.

Currently there are two alternatives: the caching and the noncaching stream pipeline.

- **`org.apache.cocoon.components.pipeline.CachingEventPipeline`** (default)

  The result is cached for further requests (if possible). Example:

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

- **`org.apache.cocoon.components.pipeline.NonCachingEventPipeline`**

The result of the request is never cached.

Example:

```
<event-pipeline
    class="org.apache.cocoon.components.pipeline.NonCachingEventPipelin
    e"/>
```

The Source Handler

The source handler is the central component for resolving the custom URI protocols, such as `cocoon:` and `resource:`. It is possible to add additional protocols by adding a source factory for the protocol to the handler's configuration:

```
<source-handler>
   <protocol name="zip" class="cxa.components.source.ZipSourceFactory"/>
</source-handler>
```

Chapter 9, "Developing Components for Cocoon," contains more information about developing custom protocols.

XSP Logic Sheets

All built-in XSP logic sheets are configured in cocoon.xconf. The `markup-languages` component manages these logic sheets. A new logic sheet can be added as a configuration inside a `builtin-logicsheet` block. The following excerpt from cocoon.xconf shows the configuration for the `xsp-request` logic sheet:

```
<markup-languages>
   <xsp-language name="xsp">
     <parameter name="prefix" value="xsp"/>
     <parameter name="uri" value="http://apache.org/xsp"/>

       <builtin-logicsheet>
          <parameter name="prefix" value="xsp-request"/>
          <parameter name="uri"
    value="http://apache.org/xsp/request/2.0"/>
          <parameter name="href"
    value="resource://org/apache/cocoon/components/language/
markup/xsp/java/request.xsl"/>
     </builtin-logicsheet>
     ...

   </target-language>
  </xsp-language>

  ...
</markup-languages>
```

A logic sheet needs a namespace with a prefix. Using this prefix and namespace, an XSP document can use the tags defined in the logic sheet. The third mandatory configuration of a logic sheet is the location where Cocoon can find the logic sheet. More information about XSP and logic sheets is contained in Chapter 10, “Cocoon News Portal: Advanced Version.”

Data Sources

The connections to an external database can be configured in cocoon.xconf. A connection gets a unique name and can then be used in other components such as the sql transformer. A sample data source definition looks like this:

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

The important information for a data source includes the unique name (here it’s `portal`), the URL to access the database, and possibly a database user and password. More information is contained in Chapter 7, “Cocoon News Portal: Extended Version.”

Appendix B. Cocoon API Specifications

Together with the previous appendix, this part of the book is the “reference manual” you can use when writing applications or extending Cocoon with your own components. Chapters 8 and 9 contain the relevant information on Cocoon’s internal workings from a developer perspective. They show you how to develop new components that can then be added to the framework.

This appendix consists of the Java documentation as extracted from Release 2.0 of Cocoon, contained on the companion CD. We have reduced the number of interfaces and classes to the ones we consider to be essential for any Cocoon developer. The CD contains the complete documentation for all the classes contained in Cocoon.

Avalon Framework and LogKit

We will start with an overview of the most common classes and interfaces of the Avalon Framework and the Avalon LogKit.

Package *org.apache.log*

This package contains all the classes of the Avalon LogKit. In order to be able to use the LogKit, we only need to look at the `Logger` class.

Class *Logger*

This is the object that the client interacts with to perform any logging.

Methods

```
public final boolean isDebugEnabled()
    Determine if messages of priority DEBUG will be logged.
Returns
    true if DEBUG messages will be logged

public final void debug(String message, Throwable throwable)
```

```
    Log a debug priority event.
Parameters
    message - the message
    throwable - the throwable

public final void debug(String message)
    Log a debug priority event.
Parameters
    message - the message

public final boolean isInfoEnabled()
    Determine if messages of priority INFO will be logged.
Returns
    true if INFO messages will be logged

public final void info(String message, Throwable throwable)
    Log a info priority event.
Parameters
    message - the message

public final void info(String message)
    Log a info priority event.
Parameters
    message - the message

public final boolean isWarnEnabled()
    Determine if messages of priority WARN will be logged.
Returns
    true if WARN messages will be logged

public final void warn(String message, Throwable throwable)
    Log a warn priority event.
Parameters
    message - the message
    throwable - the throwable

public final void warn(String message)
    Log a warn priority event.
Parameters
    message - the message
public final boolean isErrorEnabled()
    Determine if messages of priority ERROR will be logged.
Returns
    true if ERROR messages will be logged

public final void error(String message, Throwable throwable)
    Log a error priority event.
Parameters
    message - the message
    throwable - the throwable

public final void error(String message)
Log a error priority event.
Parameters
    message - the message

public final boolean isFatalErrorEnabled()
```

```
    Determine if messages of priority FATAL_ERROR will be logged.
Returns
    true if FATAL_ERROR messages will be logged

public final void fatalError(String message, Throwable throwable)
    Log a fatalError priority event.
Parameters
    message - the message
    throwable - the throwable

public final void fatalError(String message)
    Log a fatalError priority event.
Parameters
    message - the message

public final void setAdditivity(boolean additivity)
    Make this logger additive. ie Send all log events to parent loggers
    LogTargets regardless of whether or not the LogTargets have been
     overidden.
    This is derived from Log4js notion of Additivity.
Parameters
    additivity - true to make logger additive, false otherwise

public final void log(Priority priority, String message,
    Throwable throwable)
    Log a event at specific priority with a certain message and throwable.
Parameters
    message - the message
    priority - the priority
    throwable - the throwable

public final void log(Priority priority, String message)
    Log a event at specific priority with a certain message.
Parameters
    message - the message
    priority - the priority
public synchronized void setPriority(Priority priority)
    Set the priority for this logger.
Parameters
    priority - the priority

public synchronized void unsetPriority()
   Unset the priority of Logger. (Thus it will use it's parent's priority
    or
   DEBUG if no parent.

public synchronized void unsetPriority(boolean recursive)
    Unset the priority of Logger. (Thus it will use it's parent's priority
    or
    DEBUG if no parent. If recursive is true unset priorities of all child
    loggers.
Parameters
    recursive - true to unset priority of all child loggers

public synchronized void setLogTargets(LogTarget[] logTargets)
    Set the log targets for this logger.
Parameters
```

```
    logTargets - the Log Targets

public synchronized void unsetLogTargets()
    Unset the logtargets for this logger. This logger (and thus all child
    loggers who don't specify logtargets) will inherit from the parents
    LogTargets.

public synchronized void unsetLogTargets(boolean recursive)
    Unset the logtargets for this logger and all child loggers if recursive
    is set. The loggers unset (and all child loggers who don't specify
    logtargets)  will inherit from the parents LogTargets.

public synchronized org.apache.log.Logger[] getChildren()
    Get all the child Loggers of current logger.
Returns
    the child loggers

public synchronized Logger getChildLogger(String subCategory)
    Create a new child logger. The category of child logger is
    [current-category].subcategory
Parameters
    subCategory - the subcategory of this logger
Returns
    the new logger
Throws
    IllegalArgumentException - if subCategory has an empty element name

public final org.apache.log.Priority getPriority()
    Retrieve priority associated with Logger.
Returns
    the loggers priority

public final java.lang.String getCategory()
    Retrieve category associated with logger.
Returns
    the Category

public org.apache.log.LogTarget[] getLogTargets()
    Get a copy of log targets for this logger.
Returns
    the child loggers
```

Fields

```
public static final CATEGORY_SEPARATOR
```

Package *org.apache.avalon.framework.activity*

This package contains some of Avalon’s life cycle interfaces. Their functionality is acti-vating and deactivating the component.

Interface *Disposable*

The `Disposable` interface is used when components need to deallocate and dispose of resources before their destruction.

Methods

```
public void dispose()
    The dispose operation is called at the end of a components lifecycle.
    This method will be called after Startable.stop() method
    (if implemented by component). Components use this method
    to release and destroy any resources that the Component owns.
```

Interface *Initializable*

The `Initializable` interface is used by components that need to allocate resources before becoming active.

Methods

```
public void initialize()
    Initialize the component. Initialization includes allocating any
     resources
    required throughout the components lifecycle.
Throws
    Exception - if an error occurs.
```

Package *org.apache.avalon.framework.component*

This is the main package for Avalon components. It contains the basic `Component` interface and all other interfaces dealing with component handling, such as `ComponentManager`.

Interface *Component*

This interface identifies classes that can be used as components by a `Composable` object.

The contract surrounding the component is that it can be used by other classes. A component is the basic building block of the Avalon Framework. When a class implements this interface, it allows itself to be managed by a `ComponentManager` and used by an outside element called a composable. The composable must know what type of component it is accessing, so it recasts the component into the type it needs.

In order for a component to be useful, you must either extend this interface or implement this interface in conjunction with one that has methods. The new interface is the contract with the composable that this is a particular type of component, and as such it can perform those functions on that type of component.

For example, suppose you want a component that performs a logging function. You extend the component to be a `LoggingComponent`:

```
interface LoggingComponent extends Component {log(String message); }
```

Now all composable components that want to use this type of component recast the component into a `LoggingComponent`, and the composable can use the `log` method.

Interface *ComponentManager*

A `ComponentManager` selects components based on a role. The contract is that all the components implement the differing roles, and there is one component per role. If you need to select one of several components that implement the same role, you need to use a `ComponentSelector`. A role is usually the full interface name. A role can be better understood through the analogy of a play. There are many different roles in a script. Any actor or actress can play any given part, and you get basically the same results (phrases said, movements made, and so on). However, the nuances of the performance are different. Here is a list of things that might be considered the different roles:

- `InputAdaptor` and `OutputAdaptor`
- Store and pool

The `ComponentManager` does not specify the methodology of getting the component, merely the interface used to get it. Therefore, the `ComponentManager` can be implemented with a factory pattern, an object pool, or a simple hash table. See also:

- `org.apache.avalon.framework.component.Component`
- `org.apache.avalon.framework.component.Composable`
- `org.apache.avalon.framework.component.ComponentSelector`

Methods

```
public Component lookup(String role)
    Get the Component associated with the given role. For instance, If the
    ComponentManager had a LoggerComponent stored and referenced by role,
    I would use the following call: try {MyComponent log; myComponent =
    (MyComponent)manager.lookup(MyComponent.ROLE); }catch (...) {... }
Parameters
    name - The role name of the Component to retrieve.
Throws
    ComponentException - if an error occurs

public boolean hasComponent(String role)
    Check to see if a Component exists for a role.
Parameters
    role - a string identifying the role to check.
Returns
    True if the component exists, False if it does not.
```

```
public void release(Component component)
    Return the Component when you are finished with it. This allows the
    ComponentManager to handle the End-Of-Life Lifecycle events associated
    with the Component. Please note, that no Exceptions should be thrown
    at this
point. This is to allow easy use of the ComponentManager system without
having to trap Exceptions on a release.
Parameters
    component - The Component we are releasing.
```

Interface *ComponentSelector*

A `ComponentSelector` selects components based on a hint. The contract is that all the components implement the same role. A role can be better understood through the analogy of a play. There are many different roles in a script. Any actor or actress can play any given part, and you get basically the same results (phrases said, movements made, and so on). However, the nuances of the performance are different. Here is a list of things that might be considered the same role:

- `XMLInputAdaptor` and `PropertyInputAdaptor`
- `FileGenerator` and `SQLGenerator`

The `ComponentSelector` does not specify the methodology of getting the component, merely the interface used to get it. Therefore, the `ComponentSelector` can be implemented with a factory pattern, an object pool, or a simple hash table.

See also:

- `org.apache.avalon.framework.component.Component`
- `org.apache.avalon.framework.component.Composable`
- `org.apache.avalon.framework.component.ComponentManager`

Inheritance

```
implements org.apache.avalon.framework.component.Component
```

Methods

```
public Component select(Object hint)
    Select the Component associated with the given hint. For instance, If
    the ComponentSelector has a Generator stored and referenced by a URL,
    I would
use the following call:
        try {
            Generator input;
            input = (Generator)selector.select(new URL("foo://demo/url"));
        }
        catch (...) {... }
```

```
Parameters
    name - A hint to retrieve the correct Component.
Throws
    ComponentNotFoundException - If the given role is not associated with
    a Component. ComponentNotAccessibleException - If a Component instance
    cannot be created.

public boolean hasComponent(Object hint)
    Check to see if a Component exists for a hint.
Parameters
    role - a string identifying the role to check.
Returns
    True if the component exists, False if it does not.

public void release(Component component)
    Return the Component when you are finished with it. This allows the
    ComponentManager to handle the End-Of-Life Lifecycle events associated
    with the Component. Please note, that no Exceptions should be thrown
    at this
point. This is to allow easy use of the ComponentManager system without
having to trap Exceptions on a release.
Parameters
    component - The Component we are releasing.
```

Interface *Composable*

A composer is a class that needs to connect to software components using a “role” abstraction, thus not depending on particular implementations but on behavioral interfaces.

The contract surrounding a composable is that it is a user. The composable can use components managed by the `ComponentManager` it was initialized with. As part of the contract with the system, the instantiating entity must call the `compose` method before the composable can be considered valid.

Methods

```
public void compose(ComponentManager componentManager)
    Pass the ComponentManager to the composer. The Composable implementation
    should use the specified ComponentManager to acquire the components it
    needs for execution.
Parameters
    manager - The ComponentManager which this Composable uses.
```

Class *ComponentException*

The exception is thrown to indicate a problem with components. It is usually thrown by `ComponentManager` or `ComponentSelector`.

Inheritance

```
extends org.apache.avalon.framework.CascadingException
```

Constructors

```
public ComponentException(
    String message,
    Throwable throwable)

Construct a new ComponentException instance.

Parameters
    message - the exception message
    throwable - the throwable

public ComponentException(
    String message)

Construct a new ComponentException instance.
Parameters
    message - the exception message
```

Package *org.apache.avalon.framework.configuration*

This package holds the life cycle interfaces for component configuration.

Interface *Configurable*

This interface should be implemented by classes that need to be configured with custom parameters before initialization.

The contract surrounding a configurable is that the instantiating entity must call the `configure` method before this object can be considered valid. The `configure` method must be called after the constructor and before any other method. Note that this interface is incompatible with `Parameterizable`.

Methods

```
public void configure(Configuration configuration)
    Pass the Configuration to the Configurable class. This method must always
     be called after the constructor and before any other method.
Parameters
    configuration - the class configurations.
```

Interface *Configuration*

`Configuration` is an interface encapsulating a configuration node used to retrieve configuration values. This interface prevents applications from modifying their own configurations.

The contract surrounding the configuration is that after it is created, information never changes. The configuration is built by the `SAXConfigurationBuilder` and the `ConfigurationImpl` helper classes.

Methods

```
public String getName()
    Return the name of the node.
Returns
    name of the Configuration node.

Public String getLocation()
    Return a string describing location of Configuration. Location can be
different for different mediums (ie "file:line" for normal XML files or
    "table:primary-key" for DB based configurations);
Returns
    a string describing location of Configuration

public Configuration getChild(String child)
    Return a new Configuration instance encapsulating the specified child
    node.
Parameters
    child - The name of the child node.
Returns
    Configuration

public Configuration getChild(String child,
                          boolean createNew)
    Return a new Configuration instance encapsulating the specified child
    node.
Parameters
    child - The name of the child node.
Returns
    Configuration

public Configuration[] getChildren()
    Return an Array of Configuration elements containing all node children.
Returns
    The child nodes with name
public Configuration[] getChildren(String name)
    Return an Array of Configuration elements containing all node children
    with the specified name.
Parameters
    name - The name of the children to get.
Returns
    The child nodes with name

public String[] getAttributeNames()
    Return an array of all attribute names.

public String getAttribute(String paramName)
    Return the value of specified attribute.
Parameters
    paramName - The name of the parameter you ask the value of.
```

```
Returns
    String value of attribute.
Throws
    ConfigurationException - If no attribute with that name exists.

public int getAttributeAsInteger(String paramName)
    Return the int value of the specified attribute contained in this node.
Parameters
    paramName - The name of the parameter you ask the value of.
Returns
    in value of attribute
Throws
    ConfigurationException - If no parameter with that name exists.
                        or if conversion to int fails.

public long getAttributeAsLong(String name)
    Returns the value of the attribute specified by its name as a long.
Parameters
    paramName - The name of the parameter you ask the value of.
Returns
    long value of attribute
Throws
    ConfigurationException - If no parameter with that name exists. or if
                        conversion to long fails.

public float getAttributeAsFloat(String paramName)
    Return the float value of the specified parameter contained in this node.
Parameters
    paramName - The name of the parameter you ask the value of.
Returns
    float value of attribute
Throws
    ConfigurationException - If no parameter with that name exists. or if
     conversion to float fails.
public boolean getAttributeAsBoolean(String paramName)
    Return the boolean value of the specified parameter contained in this
     node.

Parameters
    paramName - The name of the parameter you ask the value of.
Returns
    boolean value of attribute
Throws
    ConfigurationException - If no parameter with that name exists. or if
     conversion to boolean fails.

public String getValue()
    Return the String value of the node.
Returns
    the value of the node.

public int getValueAsInteger()
Return the int value of the node.
Returns
    the value of the node.
Throws
    ConfigurationException - If conversion to int fails.
```

```
public float getValueAsFloat()
Return the float value of the node.
Returns
    the value of the node.
Throws
    ConfigurationException - If conversion to float fails.

public boolean getValueAsBoolean()
Return the boolean value of the node.

Returns
    the value of the node.
Throws
    ConfigurationException - If conversion to boolean fails.

public long getValueAsLong()
Return the long value of the node.

Returns
    the value of the node.
Throws
    ConfigurationException - If conversion to long fails.

public String getValue(String defaultValue)
    Returns the value of the configuration element as a String.
    If the configuration value is not set, the default value will be used.
Parameters
    defaultValue - The default value desired.
Returns
    String value of the Configuration, or default if none specified.

public int getValueAsInteger(int defaultValue)
    Returns the value of the configuration element as an int.
    If the configuration value is not set, the default value will be used.
Parameters
    defaultValue - The default value desired.
Returns
    int value of the Configuration, or default if none specified.

public long getValueAsLong(long defaultValue)
    Returns the value of the configuration element as a long.
    If the configuration value is not set, the default value will be used.
Parameters
    defaultValue - The default value desired.
Returns
    long value of the Configuration, or default if none specified.

public float getValueAsFloat(float defaultValue)
    Returns the value of the configuration element as a float.
    If the configuration value is not set, the default value will be used.
Parameters
    defaultValue - The default value desired.
Returns
    float value of the Configuration, or default if none specified.

public boolean getValueAsBoolean(boolean defaultValue)
```

```
    Returns the value of the configuration element as a boolean.
    If the configuration value is not set, the default value will be used.
Parameters
    defaultValue - The default value desired.
Returns
    boolean value of the Configuration, or default if none specified.

public String getAttribute(String name,
                           String defaultValue)
    Returns the value of the attribute specified by its name as a String,
    or the default value if no attribute by that name exists or is empty.
Parameters
    name - The name of the attribute you ask the value of.
    defaultValue - The default value desired.
Returns
    String value of attribute. It will return the default value if the
    named attribute does not exist, or if the value is not set.

public int getAttributeAsInteger(String name, int defaultValue)
    Returns the value of the attribute specified by its name as a int,
    or the default value if no attribute by that name exists or is empty.
Parameters
    name - The name of the attribute you ask the value of.
    defaultValue - The default value desired.
Returns
    int value of attribute. It will return the default value
    if the named attribute does not exist, or if the value is not set.

public long getAttributeAsLong(String name,
                               long defaultValue)
    Returns the value of the attribute specified by its name as a long,
    or the default value if no attribute by that name exists or is empty.
Parameters
    name - The name of the attribute you ask the value of.
    defaultValue - The default value desired.
Returns
    long value of attribute. It will return the default value if the
    named attribute does not exist, or if the value is not set.

public float getAttributeAsFloat(String name,
                                 float defaultValue)
    Returns the value of the attribute specified by its name as a float,
    or the default value if no attribute by that name exists or is empty.
Parameters
    name - The name of the attribute you ask the value of.
    defaultValue - The default value desired.
Returns
    float value of attribute. It will return the default value if the
    named attribute does not exist, or if the value is not set.

public boolean getAttributeAsBoolean(String name,
                                     boolean defaultValue)
    Returns the value of the attribute specified by its name as a boolean,
    or the default value if no attribute by that name exists or is empty.
Parameters
    name - The name of the attribute you ask the value of.
    defaultValue - The default value desired.
```

```
Returns
    boolean value of attribute. It will return the default value
    if the named attribute does not exist, or if the value is not set.
```

Class *ConfigurationException*

This class is thrown when a `Configurable` component cannot be configured properly or if a value cannot be retrieved properly.

Inheritance

```
extends org.apache.avalon.framework.CascadingException
```

Constructors

```
public ConfigurationException(String message)
    Construct a new ConfigurationException instance.
Parameters
    message - The detail message for this exception.

public ConfigurationException(String message,
                              Throwable throwable)
    Construct a new ConfigurationException instance.
Parameters
    message - The detail message for this exception.
    throwable - the root cause of the exception
```

Package *org.apache.avalon.framework.context*

This Avalon package contains all the interfaces for the `Contextualizable` life cycle interface.

Interface *Context*

The context is the interface through which the component and its container communicate. Each container-component relationship also involves defining a contract between two entities. This contract specifies the services, settings, and information that are supplied by the container to the component. This relationship should be documented in a well-known place. It is sometimes convenient to derive from `Context` to provide a particular style of context for your component-container relationship. The documentation for required entries in context can then be defined there. (Examples include `Cocoon Context`, `MailetContext`, and `BlockContext`.)

Methods

```
public Object get(Object key)
    Retrieve an object from Context.
Parameters
```

```
    key - the key into context
Returns
    the object
Throws
    ContextException - if object not found. Note that this means that either
    Component is asking for invalid entry or the Container is not living
    up to contract.
```

Interface *Contextualizable*

This interface should be implemented by components that need a context to work. The context contains a runtime-generated object provided by the container to this component.

Methods

```
public void contextualize(Context context)
    Pass the Context to the component. This method is called after the
    Loggable.setLogger() (if present) method and before any other method.
Parameters
    context - the context
Throws
    ContextException - if context is invalid
```

Class *ContextException*

This exception signals a badly formed context. This can be thrown by a `Context` object when an entry is not found. It can also be thrown manually in `contextualize()` when a component detects a malformed context value.

Inheritance

```
extends org.apache.avalon.framework.CascadingException
```

Constructors

```
public ContextException(String message)
    Construct a new ContextException instance.
Parameters
    message - The detail message for this exception.

public ContextException(String message,
                        Throwable throwable)
    Construct a new ContextException instance.
Parameters
    message - The detail message for this exception.
    throwable - the root cause of the exception
```

Package *org.apache.avalon.framework.logger*

The Avalon Framework supports components that want to use the LogKit to log messages.

Interface *Loggable*

Components that need to log can implement this interface so that they will receive loggers.

Methods

```
public void setLogger(Logger logger)
    Provide component with a logger.
Parameters
    logger - the logger
```

Abstract Class *AbstractLoggable*

This is a utility class that allows the construction of simple components that perform logging.

Inheritance

```
implements org.apache.avalon.framework.logger.Loggable
```

Constructors

```
public AbstractLoggable()
```

Methods

```
public void setLogger(Logger logger)
    Set the components logger.
Parameters
    logger - the logger

protected final org.apache.log.Logger getLogger()
    Helper method to allow sub-classes to acquire logger.
Returns
    the Logger

protected void setupLogger(Object component)
    Helper method to setup other components with same logger.
Parameters
    component - the component to pass logger object to
```

```
protected void setupLogger(Object component,
                           String subCategory)
    Helper method to setup other components with logger. The logger has the
    subcategory of this components logger.
Parameters
    component - the component to pass logger object to
    subCategory - the subcategory to use (may be null)

protected void setupLogger(Object component,
                           Logger logger)
    Helper method to setup other components with logger.
Parameters
    component - the component to pass logger object to
    logger - the Logger
```

Package *org.apache.avalon.framework.parameters*

This package is another way of configuring a component.

Interface *Parameterizable*

Components should implement this interface if they want to be provided with parameters during startup. This interface is called after `Composable.compose()` and before `Initializable.initialize()`. It is incompatible with the `Configurable` interface, because a component can implement only one life cycle interface dealing with configuration.

Methods

```
public void parameterize(Parameters parameters)
    Provide component with parameters.
Parameters
    parameters - the parameters
Throws
    ParameterException - if parameters are invalid
```

Class *Parameters*

The `Parameters` class represents a set of key-value pairs. Each value stored in `Parameters` has a key. This class is similar to `java.util`. Properties with convenience methods access property values by type.

Inheritance

```
implements java.io.Serializable
```

Constructors

```
public Parameters()
```

Methods

```
Public String setParameter(String name,
                           String value)
    Set the String value of a specified parameter.
    If the specified value is null the parameter is removed.
Returns
    The previous value of the parameter or null.

public java.util.Iterator getParameterNames()
    Return an Enumeration view of all parameter names.
Returns
    a iterator of parameter names

public String[] getNames()
    Retrieve an array of all parameter names.
Returns
    the parameters names
public boolean isParameter(String name)
    Test if the specified parameter can be retrieved.
Parameters
    name - the parameter name
Returns
    true if parameter is a name

public String getParameter(String name)
    Retrieve the String value of the specified parameter.
    If the specified parameter cannot be found, an exception is thrown.
Parameters
    name - the name of parameter
Returns
    the value of parameter
Throws
    -

public String getParameter(String name,
                           String defaultValue)
    Retrieve the String value of the specified parameter.
    If the specified parameter cannot be found, defaultValue is returned.
Parameters
    name - the name of parameter
    defaultValue - the default value, returned if parameter does not exist
Returns
    the value of parameter

public int getParameterAsInteger(String name)
    Retrieve the int value of the specified parameter.
    If the specified parameter cannot be found, an exception is thrown.
    Hexadecimal numbers begin with 0x, Octal numbers begin with 0o and
    binary numbers begin with 0b, all other values are assumed to be decimal.
Parameters
    name - the name of parameter
Returns
```

```
    the integer parameter type
Throws
    -

public int getParameterAsInteger(String name,
                                 int defaultValue)
    Retrieve the int value of the specified parameter.
    If the specified parameter cannot be found, defaultValue is returned.
    Hexadecimal numbers begin with 0x, Octal numbers begin with 0o and
    binary numbers begin with 0b, all other values are assumed to be decimal.
Parameters
    name - the name of parameter
    defaultValue - value returned if parameter does not exist or is of wrong
     type
Returns
    the integer parameter type
public long getParameterAsLong(String name)
    Retrieve the long value of the specified parameter.
    If the specified parameter cannot be found, an exception is thrown.
    Hexadecimal numbers begin with 0x, Octal numbers begin with 0o and binary
    numbers begin with 0b, all other values are assumed to be decimal.
Parameters
    name - the name of parameter
Returns
    the long parameter type
Throws
    -

public long getParameterAsLong(String name,
                               long defaultValue)
    Retrieve the long value of the specified parameter.
    If the specified parameter cannot be found, defaultValue is returned.
    Hexadecimal numbers begin with 0x, Octal numbers begin with 0o and binary
    numbers begin with 0b, all other values are assumed to be decimal.
Parameters
    name - the name of parameter
    defaultValue - value returned if parameter does not exist or is of wrong
     type
Returns
    the long parameter type

public float getParameterAsFloat(String name)
    Retrieve the float value of the specified parameter.
    If the specified parameter cannot be found, an exception is thrown.
Parameters
    name - the parameter name
Returns
    the value
Throws
    -

public float getParameterAsFloat(String name,
                                 float defaultValue)
    Retrieve the float value of the specified parameter.
    If the specified parameter cannot be found, defaultValue is returned.
Parameters
    name - the parameter name
```

```
    defaultValue - the default value if parameter does not exist or is of
    wrong type
Returns
    the value

public boolean getParameterAsBoolean(String name)
    Retrieve the boolean value of the specified parameter.
    If the specified parameter cannot be found, an exception is thrown.
Parameters
    name - the parameter name
Returns
    the value
Throws
    -

public boolean getParameterAsBoolean(String name,
                                     boolean defaultValue)
    Retrieve the boolean value of the specified parameter.
    If the specified parameter cannot be found, defaultValue is returned.
Parameters
    name - the parameter name
    defaultValue - the default value if parameter does not exist or is of
    wrong type
Returns
    the value

public Parameters merge(Parameters other)
    Merge parameters from another Parameters instance into this.
Parameters
    other - the other Parameters
Returns
    This Parameters instance.

public void makeReadOnly()

protected final void checkWriteable()

public static Parameters fromConfiguration(Configuration configuration)
    Create a Parameters object from a Configuration object.
Parameters
    configuration - the Configuration
Returns
    This Parameters instance.

public static Parameters fromProperties(Properties properties)
    Create a Parameters object from a Properties object.
Parameters
    properties - the Properties
Returns
    This Parameters instance.
```

Class *ParameterException*

Thrown when a `Parameterizable` component cannot be parameterized properly or if a value cannot be retrieved properly.

Inheritance

```
extends org.apache.avalon.framework.CascadingException
```

Constructors

```
public ParameterException(String message)
    Construct a new ParameterException instance.
Parameters
    message - The detail message for this exception.

public ParameterException(String message,
                      Throwable throwable)
    Construct a new ParameterException instance.
Parameters
    message - The detail message for this exception.
    throwable - the root cause of the exception
```

Package *org.apache.avalon.framework.thread*

The two important component lifetime interfaces are `SingleThreaded` and `ThreadSafe`.

Interface *SingleThreaded*

This interface marks a component as `SingleThreaded`. This interface is incompatible with `ThreadSafe`.

Interface *ThreadSafe*

This interface marks a component as `ThreadSafe` or reentrant. This interface is incompatible with `SingleThreaded`.

Cocoon

This section contains all the relevant classes and interfaces of the Cocoon core that are needed to develop new components.

Package *org.apache.cocoon*

Cocoon’s main package. It contains the `Processor` interface and some important constants.

Interface Constants

Fields

```
public static final NAME
public static final VERSION
public static final COMPLETE_NAME
public static final CONF_VERSION
public static final YEAR
public static final RELOAD_PARAM
public static final SHOWTIME_PARAM
public static final VIEW_PARAM
public static final ACTION_PARAM
public static final TEMPDIR_PROPERTY
public static final DEFAULT_CONTEXT_DIR
public static final DEFAULT_DEST_DIR
public static final DEFAULT_WORK_DIR
public static final DEFAULT_CONF_DIR
public static final PARSER_PROPERTY
public static final DEFAULT_PARSER
public static final XSP_PREFIX
public static final XSP_URI
public static final XSP_REQUEST PREFIX
public static final XSP_REQUEST URI
public static final XSP_RESPONSE PREFIX
public static final XSP_RESPONSE_URI
public static final XSP_COOKIE_PREFIX
public static final XSP_COOKIE_URI
public static final XSP_FORMVALIDATOR_PATH
public static final XSP_FORMVALIDATOR_PREFIX
public static final XSP_FORMVALIDATOR_URI
public static final XML_NAMESPACE_URI
public static final LINK_CONTENT_TYPE
public static final LINK_VIEW
public static final LINK_CRAWLING_ROLE
public static final REQUEST_OBJECT
public static final RESPONSE_OBJECT
public static final CONTEXT_OBJECT
public static final LINK_OBJECT
public static final INDEX_URI
public static final ERROR_NAMESPACE_URI
public static final ERROR_NAMESPACE_PREFIX
public static final CONTEXT_ENVIRONMENT_CONTEXT
public static final CONTEXT_CLASS_LOADER
public static final CONTEXT_WORK_DIR
public static final CONTEXT_UPLOAD_DIR
public static final CONTEXT_CACHE_DIR
public static final CONTEXT_CLASSPATH
public static final CONTEXT_CONFIG_URL
public static final DESCRIPTOR_RELOADABLE_DEFAULT
```

Interface *Processor*

The `Processor` is the central Cocoon object. In other words, this *is* Cocoon. A processor processes a request and generates the response for this request.

Methods

```
public boolean process(Environment environment)
    Process the given Environment producing the output.
Returns
    If the processing is successful true is returned. If not match is found
     in the sitemap false is returned.
Throws
    ResourceNotFoundException - If a sitemap component tries to access a
     resource
    which can not be found, e.g. the generator
    ConnectionResetException If the connection was reset

public boolean process(Environment environment,
                  StreamPipeline pipeline,
                  EventPipeline eventPipeline)
    Process the given Environment to assemble a StreamPipeline and an
EventPipeline.
```

Fields

```
public static final ROLE
```

Class *ProcessingException*

This exception is thrown every time there is a problem with processing a request.

Inheritance

```
extends org.apache.avalon.framework.CascadingException
```

Constructors

```
public ProcessingException(String message)
    Construct a new ProcessingException instance.

public ProcessingException(Exception ex)
    Creates a new ProcessingException instance.
Parameters
    ex - an Exception value

public ProcessingException(String message, Throwable t)
    Construct a new ProcessingException that references a parent Exception.
```

Class *ResourceNotFoundException*

This exception is thrown every time there is a problem with finding a resource.

Inheritance

```
extends org.apache.cocoon.ProcessingException
```

Constructors

```
public ResourceNotFoundException(String message)
    Construct a new ResourceNotFoundException instance.

public ResourceNotFoundException(String message, Throwable t)
    Construct a new ResourceNotFoundException that references a parent
     Exception.
```

Package *org.apache.cocoon.acting*

All the classes that belong to actions are in this package.

Interface *Action*

This interface describes the action component used in the sitemap.

Inheritance

```
implements org.apache.avalon.framework.component.Component
```

Methods

```
public java.util.Map act(Redirector redirector,
                         SourceResolver resolver,
                         Map objectModel,
                         String source,
                         Parameters par)
    Controls the processing against some values of the Dictionary objectModel
     and returns a Map object with values used in subsequent sitemap
     substitution patterns. NOTE: This interface is designed so that
     implementations can be ThreadSafe. When an action is ThreadSafe, only
     one instance serves all requests: this reduces memory usage and avoids
     pooling.
Parameters
    resolver - The SourceResolver in charge
    objectModel - The Map with object of the calling environment which can
    be used to select values this controller may need (ie Request, Response).
    source - A source String to the Action
    parameters - The Parameters for this invocation
Returns
    Map The returned Map object with sitemap substitution values which can
    be used in subsequent elements attributes like src= using a xpath like
    expression:src="mydir/myval/foo" If the return value is null the
    processing inside the <map:act> element of the sitemap will be skipped.
Throws
    Exception - Indicates something is totally wrong
```

Fields

```
public static final ROLE
```

Abstract Class *AbstractAction*

`AbstractAction` gives you the infrastructure to easily deploy more actions. In order to get at the logger, use `getLogger()`.

Inheritance

```
extends org.apache.avalon.framework.logger.AbstractLoggable
implements org.apache.cocoon.acting.Action
```

Fields

```
protected static final EMPTY_MAP
    Empty unmodifiable map.
```

Abstract Class *ComposerAction*

`ComposerAction` lets any action that extends this to access `SitemapComponents`.

Inheritance

```
extends org.apache.cocoon.acting.AbstractAction
implements org.apache.avalon.framework.component.Composable
```

Methods

```
public void compose(ComponentManager manager)
    Set the current ComponentManager instance used by this Composable.
```

Fields

```
protected ComponentManager manager       The component manager instance
```

Package *org.apache.cocoon.caching*

This package contains all the important classes and interfaces for the Cocoon caching mechanism.

Interface *Cacheable*

This marker interface declares a (sitemap) component as cacheable.

Methods

```
public long generateKey()
    Generate the unique key. This key must be unique inside the space of
    this component. This method must be invoked before the generateValidity()
    method.
Returns
    The generated key or 0 if the component is currently not cacheable.

public CacheValidity generateValidity()
    Generate the validity object. Before this method can be invoked the
    generateKey() method must be invoked.
Returns
    The generated validity object or null if the component is currently
    not cacheable.
```

Interface *CacheValidity*

A `CacheValidity` object contains all the information for one pipeline component to check if it is still valid. For example, the `FileGenerator` only stores the timestamp for the read XML file in this container.

Inheritance

```
implements java.io.Serializable
```

Methods

```
public boolean isValid(CacheValidity validity)
    Check if the component is still valid. This is only true if the incoming
    CacheValidity is of the same type and has the same values.
```

Class *AggregatedCacheValidity*

This is a validation object that uses a list to aggregate several `CacheValidity` objects. This object is useful if you have more than one piece of information that could possibly become invalid.

Inheritance

```
implements org.apache.cocoon.caching.CacheValidity
```

Constructors

```
public AggregatedCacheValidity()
```

Methods

```
public void add(CacheValidity validity)

public boolean isValid(CacheValidity validity)

public String toString()
```

Class *CompositeCacheValidity*

This is a validation object that uses a hash map to aggregate two `CacheValidity` objects. It's similar to `AggregatedCacheValidity`.

Inheritance

```
implements org.apache.cocoon.caching.CacheValidity
```

Constructors

```
public CompositeCacheValidity(CacheValidity v1, CacheValidity v2)
```

Methods

```
public boolean isValid(CacheValidity validity)

CacheValidity getValidity1()

CacheValidity getValidity2()
```

Class *NOPCacheValidity*

This is a validation object that is always valid. This might be the most-used `CacheValidity` object for serializers.

Inheritance

```
implements org.apache.cocoon.caching.CacheValidity
```

Class *ParametersCacheValidity*

This is a validation object that compares two hash maps if they contain the same keys with the same values.

Inheritance

```
implements org.apache.cocoon.caching.CacheValidity
```

Constructors

```
public ParametersCacheValidity(HashMap map)
```

Methods

```
public boolean isValid(CacheValidity validity)

public java.util.HashMap getParameters()
```

Class *TimeStampCacheValidity*

This is a validation object for timestamps. This is the most-used `CacheValidity` object.

Inheritance

```
implements org.apache.cocoon.caching.CacheValidity
```

Constructors

```
public TimeStampCacheValidity(long timeStamp)
```

Methods

```
public boolean isValid(CacheValidity validity)

public long getTimeStamp()
```

Package *org.apache.cocoon.components.parser*

This package contains all interfaces and classes having to do with the XML parser.

Interface *Parser*

This is the XML parser used by all Cocoon components.

Inheritance

```
implements org.apache.avalon.framework.component.Component
implements org.apache.cocoon.xml.XMLProducer
implements org.apache.cocoon.xml.dom.DOMFactory
```

Methods

```
public void setContentHandler(ContentHandler contentHandler)

public void setLexicalHandler(LexicalHandler lexicalHandler)

public void parse(InputSource in)

public org.w3c.dom.Document parseDocument(InputSource in)
```

Fields

```
public static final ROLE
```

Package *org.apache.cocoon.components.pipeline*

This package contains all classes and interfaces for Cocoon's processing pipelines, such as the stream and event pipeline.

Interface *EventPipeline*

This interface describes the XML processing pipeline consisting of a generator and the transformers.

Inheritance

```
implements org.apache.avalon.framework.component.Component
implements org.apache.avalon.framework.component.Composable
implements org.apache.avalon.excalibur.pool.Recyclable
```

Methods

```
public boolean process(Environment environment)
    Process the given Environment producing the output

public void setGenerator(String role, String source,
                    Parameters param, Exception e)

public void setGenerator(String role, String source, Parameters param)

public org.apache.cocoon.generation.Generator getGenerator()

public void addTransformer(String role, String source, Parameters param)
```

Fields

```
public static final ROLE
```

Interface *StreamPipeline*

A `StreamPipeline` either

- Collects a reader and lets it produce a character stream
- Connects an `EventPipeline` with a serializer and lets them produce the character stream

Inheritance

```
implements org.apache.avalon.framework.component.Component
implements org.apache.avalon.framework.component.Composable
implements org.apache.avalon.excalibur.pool.Recyclable
```

Methods

```
public boolean process(Environment environment)
    Process the given Environment producing the output

public void setEventPipeline(EventPipeline eventPipeline)
    Set the EventPipeline

public EventPipeline getEventPipeline()
    Get the EventPipeline

public void setReader(String role, String source, Parameters param)
    Set the reader for this pipeline

public void setReader(String role, String source,
                  Parameters param, String mimeType)
    Set the reader for this pipeline
public void setSerializer(String role, String source, Parameters param)
    Set the serializer for this pipeline

public void setSerializer(String role, String source,
                      Parameters param, String mimeType)
    Set the serializer for this pipeline
```

Fields

```
public static final ROLE
```

Package *org.apache.cocoon.components.source*

This package holds the interfaces and classes for source resolving.

Interface *SourceFactory*

This class describes and handles new URI protocols that can be plugged into Cocoon.

Inheritance

```
extends org.apache.avalon.framework.thread.ThreadSafe
```

Methods

```
Source getSource(Environment environment, String location)
    Get a Source object. The environment parameter is optional.

Source getSource(Environment environment, URL base, String location)
    Get a Source object. The environment parameter is optional.
```

Package *org.apache.cocoon.environment*

This package describes the interfaces for the Cocoon environment for the request and response. Everything describing the current process can be found here.

Interface *Context*

This defines an interface to provide context information.

Methods

```
public Object getAttribute(String name)

public void setAttribute(String name, Object value)

public void removeAttribute(String name)

public java.util.Enumeration getAttributeNames()
public java.net.URL getResource(String path)

public String getRealPath(String path)

public String getMimeType(String file)

public String getInitParameter(String name)
```

Interface *Cookie*

This creates a cookie, a small amount of information that is sent by a servlet to a web browser, saved by the browser, and later sent back to the server. A cookie's value can uniquely identify a client, so cookies are commonly used for session management.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number. Some web browsers have bugs in how they handle the optional attributes, so use them sparingly to improve your servlets' interoperability.

The servlet sends cookies to the browser using the response's `addCookie` method, which adds fields to HTTP response headers to send cookies to the browser one at a time. The browser is expected to support 20 cookies for each web server (300 cookies total). It may limit cookie size to 4KB each.

The browser returns cookies to the servlet by adding fields to HTTP request headers. Cookies can be retrieved from a request by using the response's `getCookies` method. Several cookies might have the same name but different path attributes.

Cookies affect the caching of the web pages that use them. HTTP 1.0 does not cache pages that use cookies created with this class. This class does not support the cache control defined with HTTP 1.1.

This class supports both the Version 0 (by Netscape) and Version 1 (by RFC 2109) cookie specifications. By default, cookies are created using Version 0 to ensure the best interoperability.

Methods

```
public void setComment(String purpose)
    Specifies a comment that describes a cookie's purpose. The comment is
    useful if the browser presents the cookie to the user. Comments are not
    supported by Netscape Version 0 cookies.
Parameters
    purpose - a String specifying the comment to display to the user
See Also
    getComment

public String getComment()
    Returns the comment describing the purpose of this cookie, or null if
    the cookie has no comment.
Returns
    a String containing the comment, or null if none
See Also
    setComment
public void setDomain(String pattern)
    Specifies the domain within which this cookie should be presented.
    The form of the domain name is specified by RFC 2109. A domain name begins
    with a dot (.foo.com) and means that the cookie is visible to servers
    in a specified Domain Name System (DNS) zone (for example, www.foo.com,
    but not a.b.foo.com). By default, cookies are only returned to the server
    that sent them.
Parameters
    pattern - String containing the domain name within which this cookie
    is
    visible; form is according to RFC 2109
See Also
   getDomain

public String getDomain()
   Returns the domain name set for this cookie. The form of the domain name
    is
```

```
   set by RFC 2109.
Returns
    a String containing the domain name
See Also
    setDomain

public void setMaxAge(int expiry)
    Sets the maximum age of the cookie in seconds.
    A positive value indicates that the cookie will expire after that many
    seconds have passed. Note that the value is the maximum age when the
    cookie will expire, not the cookie's current age.
    A negative value means that the cookie is not stored persistently and
    will be deleted when the Web browser exits. A zero value causes the cookie
    to be deleted.
Parameters
    expiry - integer specifying the maximum age of the cookie in seconds;
    if negative, means the cookie is not stored; if zero, deletes the cookie
See Also
    getMaxAge

public int getMaxAge()
    Returns the maximum age of the cookie, specified in seconds, By default,
    -1 indicating the cookie will persist until browser shutdown.
Returns
    an integer specifying the maximum age of the cookie in seconds; if
    negative, means the cookie persists until browser shutdown
See Also
    setMaxAge


public void setPath(String uri)
    Specifies a path for the cookie to which the client should return the
    cookie. The cookie is visible to all the pages in the directory you specify,
    and all the pages in that directory's subdirectories. A cookie's path
    must include the servlet that set the cookie, for example, /catalog,
    which makes the cookie visible to all directories on the server under
    /catalog. Consult RFC 2109 (available on the Internet) for more
    information on setting path names for cookies.
Parameters
    uri - String specifying a path
See Also
    getPath

public String getPath()
    Returns the path on the server to which the browser returns this cookie.
    The cookie is visible to all subpaths on the server.
Returns
    a String specifying a path that contains a servlet name, for example,
    "/catalog"
See Also
    setPath

public void setSecure(boolean flag)
    Indicates to the browser whether the cookie should only be sent using
    a secure protocol, such as HTTPS or SSL. The default value is false.
Parameters
```

Flag if - true, sends the cookie from the browser to the server using only when using a secure protocol; if false, sent on any protocol

See Also

getSecure

public boolean getSecure()

Returns true if the browser is sending cookies only over a secure protocol, or false if the browser can send cookies using any protocol.

Returns

true if the browser can use any standard protocol; otherwise, false

See Also

setSecure

public String getName()

Returns the name of the cookie. The name cannot be changed after creation.

Returns

a String specifying the cookie's name

public void setValue(String newValue)

Assigns a new value to a cookie after the cookie is created. If you use a binary value, you may want to use BASE64 encoding.

With Version 0 cookies, values should not contain white space, brackets, parentheses, equals signs, commas, double quotes, slashes, question marks, at signs, colons, and semicolons. Empty values may not behave the same way on all browsers.

Parameters

newValue - String specifying the new value

See Also

getValue

public String getValue()

Returns the value of the cookie.

Returns

a String containing the cookie's present value

See Also

setValue

public int getVersion()

Returns the version of the protocol this cookie complies with. Version 1 complies with RFC 2109, and version 0 complies with the original cookie specification drafted by Netscape. Cookies provided by a browser use and identify the browser's cookie version.

Returns

0 if the cookie complies with the original Netscape specification; 1 if the cookie complies with RFC 2109

See Also

setVersion

public void setVersion(int v)

Sets the version of the cookie protocol this cookie complies with. Version 0 complies with the original Netscape cookie specification. Version 1 complies with RFC 2109. Since RFC 2109 is still somewhat new, consider version 1 as experimental; do not use it yet on production sites.

Parameters

v 0 - if the cookie should comply with the original Netscape specification; 1 if the cookie should comply with RFC 2109

See Also

```
    getVersion
```

Interface *Environment*

This is the base interface for an environment abstraction. The environment describes the current request-response cycle. It abstracts from the environment this cycle is invoked from—servlet engine, command-line interface, and so on.

Inheritance

```
implements org.apache.cocoon.environment.SourceResolver
```

Methods

```
public org.apache.cocoon.components.source.SourceHandler
     getSourceHandler()
    Get the SourceHandler for the current request

public void setSourceHandler(SourceHandler sourceHandler)
    Set the SourceHandler for the current request

public String getURI()
    Get the URI to process. The prefix is stripped off.
public String getURIPrefix()
    Get the prefix of the URI in progress.

public java.net.URL getRootContext()
    Get the Root Context

public java.net.URL getContext()
    Get current context

public String getView()
    Get the view to process

public String getAction()
    Get the action to process

public void setContext(String prefix, String uri)
    Set the context. This is similar to changeContext() except that it is
    absolute.

public void changeContext(String uriprefix, String context)
    Change the context from uriprefix to context

public void redirect(boolean sessionmode,
                 String url)
    Redirect to the given URL

public void setContentType(String mimeType)
    Set the content type of the generated resource

public String getContentType()
```

```
    Get the content type of the resource

public void setContentLength(int length)
    Set the length of the generated content

public void setStatus(int statusCode)
    Set the response status code

public java.io.OutputStream getOutputStream()
    Get the output stream where to write the generated resource.

public java.util.Map getObjectModel()
    Get the underlying object model

public boolean isResponseModified(long lastModified)
    Check if the response has been modified since the same "resource"
    was requested. The caller has to test if it is really the same "resource"
    which is requested.

public void setResponseIsNotModified()
    Mark the response as not modified.
```

Interface *ModifiableSource*

This describes a `Source` object whose data content can change.

Inheritance

```
implements org.apache.cocoon.environment.Source
```

Methods

```
public void refresh()
    Refresh the content of this object after the underlying data content
    has changed.
```

Interface *Redirector*

This is the interface for a redirector abstraction.

Methods

```
public void redirect(boolean sessionmode, String url)
    Redirect to the given URL
```

Interface *Request*

This defines an interface to provide client request information.

Methods

```
Public Object get(String name)
    Returns the value of the named attribute as an Object, or null if no
    attribute of the given name exists.
Parameters
    name - a String specifying the name of the attribute
Returns
    an Object containing the value of the attribute, or null if the attribute
   does not exist

public Object getAttribute(String name)
    Returns the value of the named attribute as an Object, or null if no
    attribute of the given name exists.
Parameters
    name - a String specifying the name of the attribute
Returns
    an Object containing the value of the attribute, or null if the attribute
    does not exist

public java.util.Enumeration getAttributeNames()
    Returns an Enumeration containing the names of the attributes available
    to this request. This method returns an empty Enumeration if the request
    has no attributes available to it.
Returns
    an Enumeration of strings containing the names of the request's
    attributes

public String getAuthType()
    Returns the name of the authentication scheme used to protect the
    servlet, for example, "BASIC" or "SSL," or null if the servlet was not
    protected
Returns
    The name of the authentication scheme used to protect the servlet,
    or null if the servlet was not protected

public String getCharacterEncoding()
    Returns the name of the character encoding used in the body of this
    request.
    This method returns null if the request does not specify a character
    encoding
Returns
    a String containing the name of the character encoding, or null
    if the request does not specify a character encoding

public int getContentLength()
    Returns the length, in bytes, of the request body
Returns
    an integer containing the length of the request body or -1 if the
    length is not known

public String getContentType()
    Returns the MIME type of the body of the request
Returns
    a String containing the name of the MIME type of the request, or -1 if
    the type is not known

public String getParameter(String name)
```

```
    Returns the value of a request parameter as a String
Parameters
    name - a String specifying the name of the parameter
Returns
    a String representing the single value of the parameter
See Also
    getParameterValues

public java.util.Enumeration getParameterNames()
    Returns an Enumeration of String objects containing the names of the
    parameters contained in this request. If the request has no parameters,
    the method returns an empty Enumeration.
Returns
    an Enumeration of String objects, each String containing the name of
    a request parameter; or an empty Enumeration if the request has no
    parameters

public String[] getParameterValues(String name)
    Returns an array of String objects containing all of the values the given
    request parameter has, or null if the parameter does not exist.
    If the parameter has a single value, the array has a length of 1.
Parameters
    name - a String containing the name of the parameter whose value is
    requested
Returns
    an array of String objects containing the parameter's values
See Also
    getParameter

public String getProtocol()
    Returns the name and version of the protocol the request uses in the
    form protocol/majorVersion.minorVersion, for example, HTTP/1.1. For
    HTTP servlets, the value returned is the same as the value of the CGI
    variable SERVER_PROTOCOL.
Returns
    a String containing the protocol name and version number

public String getScheme()
    Returns the name of the scheme used to make this request, for example,
    http, https, or ftp. Different schemes have different rules for
    constructing URLs, as noted in RFC 1738.
Returns
    a String containing the name of the scheme used to make this request

public String getServerName()
    Returns the host name of the server that received the request. For
    HTTP servlets, same as the value of the CGI variable SERVER_NAME.
Returns
    a String containing the name of the server to which the request was sent

public int getServerPort()
    Returns the port number on which this request was received. For HTTP
    servlets, same as the value of the CGI variable SERVER_PORT.
Returns
    an integer specifying the port number

public String getRemoteAddr()
```

Returns the Internet Protocol (IP) address of the client that sent the request. For HTTP servlets, same as the value of the CGI variable REMOTE_ADDR.

Returns

a String containing the IP address of the client that sent the request

public String getRemoteHost()

Returns the fully qualified name of the client that sent the request, or the IP address of the client if the name cannot be determined. For HTTP servlets, same as the value of the CGI variable REMOTE_HOST.

Returns

a String containing the fully qualified name of the client

public void setAttribute(String name, Object o)

Stores an attribute in this request. Attributes are reset between requests. Attribute names should follow the same conventions as package names. Names beginning with java.*, javax.*, and com.sun.*, are reserved for use by Sun Microsystems.

Parameters

name - a String specifying the name of the attribute

o - the Object to be stored

public void removeAttribute(String name)

Removes an attribute from this request. This method is not generally needed as attributes only persist as long as the request is being handled. Attribute names should follow the same conventions as package names. Names beginning with java.*, javax.*, and com.sun.*, are reserved for use by Sun Microsystems.

Parameters

name - a String specifying the name of the attribute to remove

public java.util.Locale getLocale()

Returns the preferred Locale that the client will accept content in, based on the Accept-Language header. If the client request doesn't provide an Accept-Language header, this method returns the default locale for the server.

Returns

the preferred Locale for the client

public java.util.Enumeration getLocales()

Returns an Enumeration of Locale objects indicating, in decreasing order starting with the preferred locale, the locales that are acceptable to the client based on the Accept-Language header. If the client request doesn't provide an Accept-Language header, this method returns an Enumeration containing one Locale, the default locale for the server.

Returns

an Enumeration of preferred Locale objects for the client

public boolean isSecure()

Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.

Returns

a boolean indicating if the request was made using a secure channel

public Cookie[] getCookies()

Returns an array containing all of the Cookie objects the client sent with this request. This method returns null if no cookies were sent.

Returns
an array of all the Cookies included with this request, or null if the request has no cookies

public java.util.Map getCookieMap()
Returns a map of the Cookie objects the client sent with this request, indexed by name. This method returns an empty map if no cookies were sent.
Returns
a Map of Cookie objects
public long getDateHeader(String name)
Returns the value of the specified request header as a long value that represents a Date object. Use this method with headers that contain dates, such as If-Modified-Since. The date is returned as the number of milliseconds since January 1, 1970 GMT. The header name is case insensitive. If the request did not have a header of the specified name, this method returns -1. If the header can't be converted to a date, the method throws an IllegalArgumentException.
Parameters
name - a String specifying the name of the header
Returns
a long value representing the date specified in the header expressed as the number of milliseconds since January 1, 1970 GMT, or -1 if the named header was not included with the request
Throws
IllegalArgumentException - If the header value can't be converted to a date

Public String getHeader(String name)
Returns the value of the specified request header as a String. If the request did not include a header of the specified name, this method returns null. The header name is case insensitive. You can use this method with any request header.
Parameters
name - a String specifying the header name
Returns
a String containing the value of the requested header, or null if the request does not have a header of that name

public java.util.Enumeration getHeaders(String name)
Returns all the values of the specified request header as an Enumeration of String objects. Some headers, such as Accept-Language can be sent by clients as several headers each with a different value rather than sending the header as a comma separated list. If the request did not include any headers of the specified name, this method returns an empty Enumeration. The header name is case insensitive. You can use this method with any request header.
Parameters
name - a String specifying the header name
Returns
a Enumeration containing the values of the requested header, or null if
the request does not have any headers of that name

public java.util.Enumeration getHeaderNames()
Returns an enumeration of all the header names this request contains. If the request has no headers, this method returns an empty enumeration.

Some servlet containers do not allow do not allow servlets to access headers using this method, in which case this method returns null

Returns

an enumeration of all the header names sent with this request; if the request has no headers, an empty enumeration; if the servlet container does not allow servlets to use this method, null

public String getMethod()

Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT. Same as the value of the CGI variable REQUEST_METHOD.

Returns

a String specifying the name of the method with which this request was made

public String getPathInfo()

Returns any extra path information associated with the URL the client sent when it made this request. The extra path information follows the servlet path but precedes the query string. This method returns null if there was no extra path information.

Same as the value of the CGI variable PATH_INFO.

Returns

a String specifying extra path information that comes after the servlet path but before the query string in the request URL; or null if the URL does not have any extra path information

public String getPathTranslated()

Returns any extra path information after the servlet name but before the query string, and translates it to a real path. Same as the value of the CGI variable PATH_TRANSLATED. If the URL does not have any extra path information, this method returns null.

Returns

a String specifying the real path, or null if the URL does not have any extra path information

public String getContextPath()

Returns the portion of the request URI that indicates the context of the request. The context path always comes first in a request URI. The path starts with a "/" character but does not end with a "/" character. For servlets in the default (root) context, this method returns "".

Returns

a String specifying the portion of the request URI that indicates the context of the request

public String getQueryString()

Returns the query string that is contained in the request URL after the path.

This method returns null if the URL does not have a query string. Same as the value of the CGI variable QUERY_STRING.

Returns

a String containing the query string or null if the URL contains no query string

public String getRemoteUser()

Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated. Whether the

user name is sent with each subsequent request depends on the browser and type of authentication. Same as the value of the CGI variable REMOTE_USER.

Returns

a String specifying the login of the user making this request, or null if the user login is not known

public java.security.Principal getUserPrincipal()

Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated. Whether the user name is sent with each subsequent request depends on the browser and type of authentication. Same as the value of the CGI variable REMOTE_USER.

Returns

a String specifying the login of the user making this request, or null if the user login is not known

public boolean isUserInRole(String role)

Checks whether the currently logged in user is in a specified role.

Returns

true if the user is authenticated and in the role; otherwise, false

See Also

#getRemoteUser

public String getRequestedSessionId()

Returns the session ID specified by the client. This may not be the same as the ID of the actual session in use. For example, if the request specified an old (expired) session ID and the server has started a new session, this method gets a new session with a new ID. If the request did not specify a session ID, this method returns null.

Returns

a String specifying the session ID, or null if the request did not specify a session ID

See Also

isRequestedSessionIdValid

public String getRequestURI()

Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.

For example:

First line of HTTP request	Returned Value
POST /some/path.html HTTP/1.1	/some/path.html
GET http://foo.bar/a.html HTTP/1.0	http://foo.bar/a.html
HEAD /xyz?a=b HTTP/1.1	/xyz

Returns

a String containing the part of the URL from the protocol name up to the query string

public String getSitemapURI()

Returns the URI of the requested resource as interpreted by the sitemap. For example, if your webapp is mounted at "/webapp" and the HTTP request is for "/webapp/foo", this method returns "foo". Consequently, if the request is for "/webapp", this method returns an empty string.

Note that if the request is mapped to a pipeline that contains aggregated content, and if this method is called in the context of one of the aggregated

parts (e.g. a server page), this method will return the URI of the aggregated
part, not the original requested URI.
Returns
a String containing the URL as mangled by the sitemap

public String getServletPath()
Returns the part of this request's URL that calls the servlet. This includes either the servlet name or a path to the servlet, but does not include any extra path information or a query string. Same as the value of the CGI
variable SCRIPT_NAME.
Returns
a String containing the name or path of the servlet being called, as specified in the request URL

public Session getSession(boolean create)
Returns the current Session associated with this request or, if there is no current session and create is true, returns a new session.
If create is false and the request has no valid Session, this method returns null. To make sure the session is properly maintained, you must call this method before the response is committed.
Parameters
true - to create a new session for this request if necessary; false to return null if there's no current session
Returns
the Session associated with this request or null if create is false and the
request has no valid session
See Also
getSession()

public Session getSession()
Returns the current session associated with this request, or if the request
does not have a session, creates one.
Returns
the Session associated with this request
See Also
getSession(boolean)

public boolean isRequestedSessionIdValid()
Checks whether the requested session ID is still valid.
Returns
true if this request has an id for a valid session in the current session context; false otherwise
See Also
getRequestedSessionId
getSession
public boolean isRequestedSessionIdFromCookie()
Checks whether the requested session ID came in as a cookie.
Returns
true if the session ID came in as a cookie; otherwise, false
See Also
getSession

public boolean isRequestedSessionIdFromURL()

```
    Checks whether the requested session ID came in as part of the request
    URL.
Returns
    true if the session ID came in as part of a URL; otherwise, false
See Also
    getSession
```

Interface *Response*

This defines an interface to provide client response information.

Methods

```
public String getCharacterEncoding()
    Returns the name of the charset used for the MIME body sent in this
    response.
    If no charset has been assigned, it is implicitly set to ISO-8859-1
    (Latin-1). See RFC 2047 (http://ds.internic.net/rfc/rfc2045.txt) for
    more information about character encoding and MIME.
Returns
    a String specifying the name of the charset, for example, ISO-8859-1

public void setLocale(Locale loc)
    Sets the locale of the response, setting the headers (including the
    Content-Type's charset) as appropriate. By default, the response locale
    is
    the default locale for the server.
Parameters
    loc - the locale of the response
See Also
    getLocale

public java.util.Locale getLocale()
    Returns the locale assigned to the response.
See Also
    setLocale

public Cookie createCookie(String name, String value)
    Constructs a cookie with a specified name and value.
    The name must conform to RFC 2109. That means it can contain only ASCII
    alphanumeric characters and cannot contain commas, semicolons, or white
    space or begin with a $ character. The cookie's name cannot be changed
    after creation.
    The value can be anything the server chooses to send. Its value is probably
    of interest only to the server. The cookie's value can be changed after
    creation with the setValue method.
    By default, cookies are created according to the Netscape cookie
    specification. The version can be changed with the setVersion method.
Parameters
    name - a String specifying the name of the cookie
    value - a String specifying the value of the cookie
Throws
    IllegalArgumentException - if the cookie name contains illegal
    characters
    (for example, a comma, space, or semicolon) or it is one of the tokens
```

reserved for use by the cookie protocol

public void addCookie(Cookie cookie)
Adds the specified cookie to the response. This method can be called multiple times to set more than one cookie.
Parameters
cookie - the Cookie to return to the client

public boolean containsHeader(String name)
Returns a boolean indicating whether the named response header has already been set.
Parameters
name - the header name
Returns
true if the named response header has already been set; false otherwise

public String encodeURL(String url)
Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. The implementation of this method includes the logic to determine whether the session ID needs to be encoded in the URL. For example, if the browser supports cookies, or session tracking is turned off, URL encoding is unnecessary.
For robust session tracking, all URLs emitted by a servlet should be run through this method. Otherwise, URL rewriting cannot be used with browsers which do not support cookies.
Parameters
url - the url to be encoded.
Returns
the encoded URL if encoding is needed; the unchanged URL otherwise.

public void setDateHeader(String name, long date)
Sets a response header with the given name and date-value. The date is specified in terms of milliseconds since the epoch. If the header had already been set, the new value overwrites the previous one. The containsHeader method can be used to test for the presence of a header before setting its value.
Parameters
name - the name of the header to set
value - the assigned date value
See Also
containsHeader
addDateHeader
public void addDateHeader(String name, long date)
Adds a response header with the given name and date-value. The date is specified in terms of milliseconds since the epoch. This method allows response headers to have multiple values.
Parameters
name - the name of the header to set
value - the additional date value
See Also
setDateHeader

```
public void setHeader(String name, String value)
    Sets a response header with the given name and value. If the header had
    already been set, the new value overwrites the previous one. The
    containsHeader method can be used to test for the presence of a header
    before setting its value.
Parameters
    name - the name of the header
    value - the header value
See Also
    containsHeader
    addHeader

public void addHeader(String name, String value)
    Adds a response header with the given name and value. This method allows
    response headers to have multiple values.
Parameters
    name - the name of the header
    value - the additional header value
See Also
    setHeader
```

Interface *Session*

Provides a way to identify a user across more than one page request or to visit to a web site and store information about that user.

Cocoon uses this interface to create a session between a client and Cocoon. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who might visit a site many times. The server can maintain a session in many ways, such as using cookies or rewriting URLs.

This interface allows Cocoon to

- View and manipulate information about a session, such as the session identifier, creation time, and last accessed time
- Bind objects to sessions, allowing user information to persist across multiple user connections

Session information is scoped only to the current context (`Context`), so information stored in one context is not directly visible in another.

Methods

```
public long getCreationTime()
    Returns the time when this session was created, measured in milliseconds
    since midnight January 1, 1970 GMT.
Returns
    a long specifying when this session was created, expressed in
    milliseconds
```

since 1/1/1970 GMT
Throws
IllegalStateException - if this method is called on an invalidated session

public String getId()
Returns a string containing the unique identifier assigned to this session.
The identifier is assigned by the context container and is implementation dependent.
Returns
a string specifying the identifier assigned to this session

public long getLastAccessedTime()
Returns the last time the client sent a request associated with this session,
as the number of milliseconds since midnight January 1, 1970 GMT.
Actions that your application takes, such as getting or setting a value associated with the session, do not affect the access time.
Returns
a long representing the last time the client sent a request associated with this session, expressed in milliseconds since 1/1/1970 GMT

public void setMaxInactiveInterval(int interval)
Specifies the time, in seconds, between client requests before the contextcontainer will invalidate this session. A negative time indicates the session should never timeout.
Parameters
interval - An integer specifying the number of seconds

public int getMaxInactiveInterval()
Returns the maximum time interval, in seconds, that the context container will keep this session open between client accesses. After this interval, the context container will invalidate the session. The maximum time interval
can be set with the setMaxInactiveInterval method. A negative time indicates
the session should never timeout.
Returns
an integer specifying the number of seconds this session remains open between client requests
See Also
setMaxInactiveInterval

public Object getAttribute(String name)
Returns the object bound with the specified name in this session, or null
if no object is bound under the name.
Parameters
name - a string specifying the name of the object
Returns
the object with the specified name
Throws
IllegalStateException - if this method is called on an invalidated session

public java.util.Enumeration getAttributeNames()

```
    Returns an Enumeration of String objects containing the names of all
    the
    objects bound to this session.
Returns
    an Enumeration of String objects specifying the names of all the objects
    bound to this session
Throws
    IllegalStateException - if this method is called on an invalidated
    session

public void setAttribute(String name, Object value)
    Binds an object to this session, using the name specified. If an object
    of
    the same name is already bound to the session, the object is replaced.
Parameters
    name - the name to which the object is bound; cannot be null
    value - the object to be bound; cannot be null
Throws
    IllegalStateException - if this method is called on an invalidated
    session

public void removeAttribute(String name)
    Removes the object bound with the specified name from this session. If
    the
    session does not have an object bound with the specified name, this method
    does nothing.
Parameters
    name - the name of the object to remove from this session
Throws
    IllegalStateException - if this method is called on an invalidated
    session

public void invalidate()
    Invalidates this session to it.
Throws
    IllegalStateException - if this method is called on an already
    invalidated
    session

public boolean isNew()
    Returns true if the client does not yet know about the session or if
    the
    client chooses not to join the session. For example, if the server used
    only cookie-based sessions, and the client had disabled the use of
    cookies,
    then a session would be new on each request.
Returns
    true if the server has created a session, but the client has not yet
    joined
Throws
    IllegalStateException - if this method is called on an already
                        invalidated session
```

Interface *SourceResolver*

This is a base interface for resolving a source by system identifiers. A system identifier can be any URI supported by Cocoon.

Methods

```
Public Source resolve(String systemID)
    Resolve the source.
Parameters
    systemID - This is either a system identifier (java.net.URL or a local
     file.
```

Interface *Source*

This describes a source. This interface provides a simple interface for accessing a source of data. The source of data is assumed to not change during the lifetime of the `Source` object. If you have a data source that can change its content, and you want this to be reflected in Cocoon, use a `@link ModifiableSource` object instead.

Inheritance

```
implements org.apache.avalon.excalibur.pool.Recyclable
implements org.apache.cocoon.xml.XMLizable
```

Methods

```
public long getLastModified()
    Get the last modification date of the source or 0 if it is not possible
    to determine the date.

public long getContentLength()
    Get the content length of the source or -1 if it is not possible to
    determine the length.

public java.io.InputStream getInputStream()
    Return an InputStream object to read from the source.

public org.xml.sax.InputSource getInputSource()
    Return an InputSource object to read the XML content.
Returns
    an InputSource value
Throws
    ProcessingException - if an error occurs
    IOException - if an error occurs

public String getSystemId()
    Return the unique identifier for this source
```

Class *ObjectModelHelper*

This is a set of constants and methods used to access the content of the object model.

The object model is a map used to pass information about the calling environment to the sitemap and its components (matchers, actions, transformers, and so on).

This class provides accessors for only the objects in the object model that are common to every environment and thus can be used safely. Some environments provide additional objects, but they are not described here. You should access them with care because doing so ties the application to that particular environment.

Methods

```
public static final Request getRequest(Map objectModel)

public static final Response getResponse(Map objectModel)

public static final Context getContext(Map objectModel)
```

Fields

```
public static final REQUEST_OBJECT
   Key for the environment Request in the object model.

public static final RESPONSE_OBJECT
    Key for the environment Response in the object model.

public static final CONTEXT_OBJECT
    Key for the environment Context in the object model.
```

Package *org.apache.cocoon.generation*

This is the Cocoon generator and all classes implementing this interface.

Interface *Generator*

This interface describes the generator for an XML processing pipeline in the sitemap.

Inheritance

```
implements org.apache.cocoon.xml.XMLProducer
implements org.apache.cocoon.sitemap.SitemapModelComponent
```

Methods

```
public void generate()
```

Fields

```
public static final ROLE
```

Abstract Class *AbstractGenerator*

This is an abstract class implementing the `Generator` interface. This class can be used as a base to implement additional custom generators.

Interfaces

```
extends org.apache.cocoon.xml.AbstractXMLProducer
implements org.apache.cocoon.generation.Generator
```

Methods

```
public void setup(SourceResolver resolver,
                  Map objectModel,
                  String src,
                  Parameters par)
    Set the SourceResolver, object model Map, the source and sitemap
    Parameters
    used to process the request.

public void recycle()
    Recycle the generator by removing references
```

Fields

```
protected resolver
    The current SourceResolver.

protected objectModel
    The current Map objectModel.

protected parameters
    The current Parameters.

protected source
    The source URI associated with the request or null.
```

Abstract Class *ComposerGenerator*

This is an abstract class implementing the `Generator` interface. This class can be used as a base to implement additional custom generators.

Inheritance

```
extends org.apache.cocoon.generation.AbstractGenerator
implements org.apache.avalon.framework.component.Composable
```

Methods

```
public void compose(ComponentManager manager)
    Set the current ComponentManager instance used by this Composable.
```

Fields

```
protected manager
     The component manager instance
```

Package *org.apache.cocoon.matching*

This package contains all the interfaces and classes for the sitemap's matcher component type.

Interface *Matcher*

This interface describes a matcher component used in the sitemap.

Inheritance

```
implements org.apache.avalon.framework.component.Component
```

Methods

```
public java.util.Map match(String pattern,
                      Map objectModel,
                      Parameters parameters)
    Matches the pattern against some Request values and returns a Map object
    with
    replacements for wildcards contained in the pattern.
Parameters
    pattern - The pattern to match against. Depending on the implementation
    the
            pattern can contain wildcards or regular expressions.
    objectModel - The Map with object of the calling environment which can
                be used to select values this matchers matches against.
Returns
    Map The returned Map object with replacements for wildcards/regular
    expressions contained in the pattern. If the return value is null there
    was no match.
```

Fields

```
public static final ROLE
```

Interface *PreparableMatcher*

This is a matcher that can prepare patterns during sitemap setup for a faster match at request time. This is also a regular matcher, meaning that the sitemap can decide either to prepare the pattern or to match with a request-time-evaluated pattern (for value substitution).

Inheritance

```
implements org.apache.cocoon.matching.Matcher
```

Methods

```
public Object preparePattern(String pattern)
    Prepares a pattern in a form that allows faster match. For example, a
    regular
    expression matcher can precompile the expression and return the
    corresponding
    object. This method is called once for each pattern used with a particular
    matcher class. The returned value is then passed back as the
    preparedPattern parameter of preparedMatch(Object, Map, Parameters).
    Parameters
    pattern - The pattern to prepare. Depending on the implementation the
    pattern
    can contain wildcards or regular expressions.
Returns
    an optimized representation of the pattern.
Throws
    a - PatternException if the pattern couldn't be prepared.

public java.util.Map preparedMatch(Object preparedPattern,
                                   Map objectModel,
                                   Parameters parameters)
    Matches the prepared pattern against some values in the object model
    (most often the Request) and returns a Map object with replacements for
    wildcards contained in the pattern.
Parameters
    preparedPattern - The preparedPattern to match against, as returned by
                      preparePattern(String).
    objectModel - The Map with objects of the calling environment which can
    be
                  used to select values this matchers matches against.
Returns
    a Map object with replacements for wildcards/regular-expressions
    contained
    in the pattern. If the return value is null there was no match.
```

Package *org.apache.cocoon.reading*

This contains all classes and interfaces for the `reader` sitemap component.

Interface *Reader*

This interface describes the `reader` component used in the sitemap.

Inheritance

```
implements org.apache.cocoon.sitemap.SitemapModelComponent,
org.apache.cocoon.sitemap.SitemapOutputComponent
```

Methods

```
public void generate()
    Generate the response.
public long getLastModified()
Returns
    the time the read source was last modified or 0 if it is not possible
     to
    detect
```

Fields

```
public static final ROLE
```

Abstract Class *AbstractReader*

This is an abstract class implementing the `Reader` interface. This class can be used as a base to implement additional custom readers.

Inheritance

```
extends org.apache.avalon.framework.logger.AbstractLoggable
implements org.apache.cocoon.reading.Reader,
implements org.apache.avalon.excalibur.pool.Recyclable
```

Methods

```
public void setup(SourceResolver resolver,
                  Map objectModel,
                  String src,
                  Parameters par)
    Set the SourceResolver the object model Map, the source and sitemap
    Parameters used to process the request.

public void setOutputStream(OutputStream out)
    Set the OutputStream

public String getMimeType()
    Get the mime-type of the output of this Serializer This default
    implementation returns null to indicate that the mime-type specified
     in the
```

```
    sitemap is to be used

public long getLastModified()
Returns
    the time the read source was last modified or 0 if it is not possible
     to
    detect

public void recycle()
    Recycle the component

public boolean shouldSetContentLength()
    Test if the component wants to set the content length
```

Fields

```
protected resolver
    The current SourceResolver

protected objectModel
    The current Map of the object model.

protected parameters
    The current Parameters

protected source
    The source URI associated with the request or null

protected out
    The OutputStream to write on.
```

Package *org.apache.cocoon.selection*

This package contains the interfaces and classes used for the `selector` sitemap component.

Interface *Selector*

This interface describes a selector used in the sitemap.

Inheritance

```
implements org.apache.avalon.framework.component.Component
```

Methods

```
public boolean select(String expression,
                      Map objectModel,
                      Parameters parameters)
    Selectors test pattern against some objects in a Map model and signals
```

```
    success with the returned boolean value
Parameters
    expression - The expression to test.
    objectModel - The Map containing object of the calling environment which
     may
    be used to select values to test the expression.
    parameters - The sitemap parameters, as specified by <parameter> tags.
Returns
    boolean Signals successful test.
```

Fields

```
public static final ROLE
```

Package *org.apache.cocoon.serialization*

This package contains all the classes and interfaces for the `serializer` sitemap component.

Interface *Serializer*

This interface describes a `serializer` component.

Inheritance

```
implements org.apache.cocoon.xml.XMLConsumer,
implements org.apache.cocoon.sitemap.SitemapOutputComponent
```

Fields

```
public static final ROLE
```

Abstract Class *AbstractSerializer*

This is an abstract class implementing the `Serializer` interface. This class can be used as a base to implement additional custom serializers.

Inheritance

```
extends org.apache.cocoon.xml.AbstractXMLPipe
implements org.apache.cocoon.serialization.Serializer,
implements org.apache.avalon.excalibur.pool.Recyclable
```

Methods

```
public void setOutputStream(OutputStream out)
```

```
    Set the OutputStream where the XML should be serialized.

public String getMimeType()
    Get the mime-type of the output of this Serializer This default
    implementation returns null to indicate that the mime-type specified
     in the
    sitemap is to be used

public void recycle()
    Recycle serializer by removing references

public boolean shouldSetContentLength()
    Test if the component wants to set the content length
```

Fields

```
protected output
    The OutputStream used by this serializer.
```

Abstract Class *AbstractTextSerializer*

This is an abstract class implementing the `Serializer` interface. This class can be used as a base to implement additional custom serializers that produce any text output, such as HTML or XML.

Inheritance

```
extends org.apache.cocoon.serialization.AbstractSerializer
implements org.apache.avalon.framework.configuration.Configurable
implements org.apache.cocoon.caching.Cacheable
implements org.apache.avalon.excalibur.pool.Poolable
```

Methods

```
protected javax.xml.transform.sax.SAXTransformerFactory
    getTransformerFactory()
    Helper for TransformerFactory.

public void setOutputStream(OutputStream out)
    Set the OutputStream where the XML should be serialized.

public void configure(Configuration conf)
    Set the configurations for this serializer.

public long generateKey()
    Generate the unique key. This key must be unique inside the space of
     this
    component. This method must be invoked before the generateValidity()
     method.
Returns
    The generated key or 0 if the component is currently not cacheable.
```

```
public org.apache.cocoon.caching.CacheValidity generateValidity()
    Generate the validity object. Before this method can be invoked the
    generateKey() method must be invoked.
Returns
    The generated validity object or null if the component is currently
    not cacheable.

public void recycle()
    Recycle serializer by removing references

public void startDocument()

public void startPrefixMapping(String prefix, String uri)
    Add tracking of mappings to be able to add xmlns: attributes in
    startElement().

public void endPrefixMapping(String prefix)
    End the scope of a prefix-URI mapping: remove entry from mapping tables.
public void startElement(String eltUri, String eltLocalName,
                     String eltQName, Attributes attrs)
    Ensure all namespace declarations are present as xmlns: attributes and
     add
    those needed before calling superclass. This is a workaround for a Xalan
    bug (at least in version 2.0.1) :
     org.apache.xalan.serialize.SerializerToXML
    ignores start/endPrefixMapping().

public void endElement(String eltUri, String eltLocalName, String eltQName)
    Receive notification of the end of an element. Try to restore the element
qName.
```

Fields

```
protected format
    The Properties used by this serializer.
```

Package *org.apache.cocoon.sitemap*

This package contains some of Cocoon’s main core classes. They are responsible for managing and processing the sitemaps and their components.

Interface *SitemapModelComponent*

This interface is implemented by all sitemap components, such as generators, readers, and transformers.

Inheritance

```
implements org.apache.avalon.framework.component.Component
```

Methods

```
public void setup(SourceResolver resolver,
                  Map objectModel,
                  String src,
                  Parameters par)
    Set the SourceResolver, objectModel Map, the source and sitemap
     Parameters
    used to process the request.
```

Interface *SitemapOutputComponent*

This interface is implemented by all sitemap components producing output: a serializer and a reader.

Inheritance

```
implements org.apache.avalon.framework.component.Component
```

Methods

```
public void setOutputStream(OutputStream out)
    Set the OutputStream where the requested resource should be serialized.

public String getMimeType()
    Get the mime-type of the output of this Component

public boolean shouldSetContentLength()
    Test if the component wants to set the content length
```

Class *Handler*

This class handles the managing and stating of one sitemap.

Inheritance

```
extends org.apache.avalon.framework.logger.AbstractLoggable
implements java.lang.Runnable
implements org.apache.avalon.framework.context.Contextualizable
implements org.apache.avalon.framework.component.Composable
implements org.apache.cocoon.Processor
implements org.apache.avalon.framework.activity.Disposable
implements org.apache.cocoon.environment.SourceResolver
```

Constructors

```
protected Handler(String sourceFileName, boolean check_reload)
```

Methods

```
public void contextualize(Context context)
    Contextualizable

public void compose(ComponentManager manager)
    Composable

protected boolean available()

protected boolean hasChanged()

protected boolean isRegenerating()

protected void regenerateAsynchronously(Environment environment)

protected void regenerate(Environment environment)

public boolean process(Environment environment)

public boolean process(Environment environment,
                   StreamPipeline pipeline,
                   EventPipeline eventPipeline)
public void setBasePath(String basePath)

public void run()
    Generate the Sitemap class

public void throwEventualException()

public Exception getException()

public void dispose()
    dispose

public org.apache.cocoon.environment.Source resolve(String systemId)
    Resolve an entity. Interface SourceResolver
```

Class *Manager*

This class manages all a sitemap's subsitemaps. This instance invokes subsitemaps and checks for regeneration of the subsitemap.

Inheritance

```
extends org.apache.avalon.framework.logger.AbstractLoggable
implements org.apache.avalon.framework.component.Component
implements org.apache.avalon.framework.configuration.Configurable
implements org.apache.avalon.framework.component.Composable
implements org.apache.avalon.framework.context.Contextualizable
implements org.apache.avalon.framework.thread.ThreadSafe
implements org.apache.avalon.excalibur.logger.LogKitManageable
```

Methods

```
public void setRoleManager(RoleManager roles)
    Set the role manager

public void setLogKitManager(LogKitManager logkit)
    Set the logkit manager

public void configure(Configuration conf)
    get a configuration
Parameters
    conf - the configuration

public void contextualize(Context context)
    get a context
Parameters
    context - the context object

public void compose(ComponentManager manager)
    get a component manager
Parameters
    manager - the component manager

public boolean invoke(ComponentManager newManager,
                  Environment environment,
                  String uri_prefix,
                  String source,
                  boolean check_reload,
                  boolean reload_asynchron)
    invokes the sitemap handler to process a request
Parameters
    environment - the environment
    uri_prefix - the prefix to the URI
    source - the source of the sitemap
    check_reload - should the sitemap be automagically reloaded
    reload_asynchron - should the sitemap be reloaded asynchron
Returns
    states if the requested resource was produced
Throws
    Exception - there may be several exceptions thrown

public boolean invoke(ComponentManager newManager,
                  Environment environment,
                  String uri_prefix,
                  String source,
                  boolean check_reload,
                  boolean reload_asynchron,
                  StreamPipeline pipeline,
                  EventPipeline eventPipeline)
    invokes the sitemap handler to process a request
Parameters
    environment - the environment
    uri_prefix - the prefix to the URI
    source - the source of the sitemap
    check_reload - should the sitemap be automagically reloaded
```

```
    reload_asynchron - should the sitemap be reloaded asynchron
Returns
    states if the requested resource was produced
Throws
    Exception - there may be several exceptions thrown

public boolean hasChanged()
    has the sitemap changed
Returns
    whether the sitemap file has changed
```

Package *org.apache.cocoon.transformation*

This package contains all interfaces and classes for the `transformer` sitemap component.

Interface *Transformer*

This interface describes the `transformer` sitemap component.

Inheritance

```
implements org.apache.cocoon.xml.XMLPipe
implements org.apache.cocoon.sitemap.SitemapModelComponent
```

Fields

```
public static final ROLE
```

Abstract Class *AbstractTransformer*

This abstract class should be used as a base for additional custom transformers.

Inheritance

```
extends org.apache.cocoon.xml.AbstractXMLPipe
implements org.apache.cocoon.transformation.Transformer
```

Package *org.apache.cocoon.util*

This package contains miscellaneous utility classes for various concerns, such as hashing and file I/O.

Class *HashUtil*

This is a very efficient Java hash algorithm. It is based on the BuzHash algorithm by Robert Uzgalis (see http://www.serve.net/buz/hash.adt/java.000.html for more

information). BuzHash is copyright 1996 Robert Uzgalis. All rights reserved. Used with kind permission of the author.

Methods

```
public static long hash(String arg)
    Hash a String.
Parameters
    arg - The String to be hashed
Returns
    The hash for the input.

public static long hash(StringBuffer arg)
    Hash a String.
Parameters
    arg - The String represented by the StringBuffer to be hashed
Returns
    The hash for the input.
```

Package *org.apache.cocoon.xml*

This package contains some common interfaces describing the actors of the XML processing pipeline, such as `XMLConsumer` and `XMLProducer`.

Interface *XMLConsumer*

This interface identifies classes that consume XML data, receiving notification of SAX events. This interface unites the idea of SAX `ContentHandler` and `LexicalHandler`.

Inheritance

```
implements org.xml.sax.ContentHandler
implements org.xml.sax.ext.LexicalHandler
```

Interface *XMLizable*

This interface can be implemented by classes that are willing to provide an XML representation of their current state as SAX events.

Methods

```
public void toSAX(ContentHandler handler)
    Generates SAX events representing the object's state. NOTE : if the
    implementation can produce lexical events, care should be taken that
     handler
    can actually be a XMLConsumer that accepts such events.
```

Interface *XMLPipe*

This interface combines an XML producer and a consumer to create a SAX pipe.

Inheritance

```
implements org.apache.cocoon.xml.XMLConsumer
implements org.apache.cocoon.xml.XMLProducer
```

Interface *XMLProducer*

This interface identifies classes that produce XML data, sending SAX events to the configured `XMLConsumer`. It's beyond the scope of this interface to specify how the XML data production should be started.

Methods

```
public void setConsumer(XMLConsumer consumer)
Set the XMLConsumer that will receive XML data.
```

Abstract Class *AbstractXMLConsumer*

This abstract class provides a default implementation of the methods specified by the `XMLConsumer` interface.

Inheritance

```
extends org.apache.avalon.framework.logger.AbstractLoggable
implements org.apache.cocoon.xml.XMLConsumer
```

Methods

```
public void setDocumentLocator(Locator locator)
    Receive an object for locating the origin of SAX document events.
Parameters
    locator - An object that can return the location of any SAX document
     event.

public void startDocument()
    Receive notification of the beginning of a document.

public void endDocument()
    Receive notification of the end of a document.

public void startPrefixMapping(String prefix, String uri)
   Begin the scope of a prefix-URI Namespace mapping.
Parameters
    prefix - The Namespace prefix being declared.
    uri - The Namespace URI the prefix is mapped to.
```

```
public void endPrefixMapping(String prefix)
    End the scope of a prefix-URI mapping.
Parameters
    prefix - The prefix that was being mapping.

public void startElement(String uri, String loc, String raw, Attributes a)
    Receive notification of the beginning of an element.
Parameters
    uri - The Namespace URI, or the empty string if the element has no
    Namespace
         URI or if Namespace processing is not being performed.
    loc - The local name (without prefix), or the empty string if Namespace
         processing is not being performed.
    raw - The raw XML 1.0 name (with prefix), or the empty string if raw
    names
         are not available.
    a - The attributes attached to the element. If there are no attributes,
       it shall be an empty Attributes object.

public void endElement(String uri, String loc, String raw)
    Receive notification of the end of an element.
Parameters
    uri - The Namespace URI, or the empty string if the element has no
    Namespace
        URI or if Namespace processing is not being performed.
    loc - The local name (without prefix), or the empty string if Namespace
        processing is not being performed.
    raw - The raw XML 1.0 name (with prefix), or the empty string if raw
    names
         are not available.
public void characters(char[] ch, int start, int len)
    Receive notification of character data.
Parameters
    ch - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.

public void ignorableWhitespace(char[] ch, int start, int len)
    Receive notification of ignorable whitespace in element content.
Parameters
    ch - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.

public void processingInstruction(String target, String data)
    Receive notification of a processing instruction.
Parameters
    target - The processing instruction target.
    data - The processing instruction data, or null if none was supplied.

public void skippedEntity(String name)
    Receive notification of a skipped entity.
Parameters
    name - The name of the skipped entity. If it is a parameter entity,
    the name will begin with '%'.

public void startDTD(String name, String publicId, String systemId)
```

```
    Report the start of DTD declarations, if any.
Parameters
    name - The document type name.
    publicId - The declared public identifier for the external DTD subset,
             or null if none was declared.
    systemId - The declared system identifier for the external DTD subset,
             or null if none was declared.

public void endDTD()
    Report the end of DTD declarations.

public void startEntity(String name)
    Report the beginning of an entity.
Parameters
    name - The name of the entity. If it is a parameter entity, the name
    will
          begin with '%'.

public void endEntity(String name)
    Report the end of an entity.
Parameters
    name - The name of the entity that is ending.

public void startCDATA()
    Report the start of a CDATA section.
public void endCDATA()
    Report the end of a CDATA section.

public void comment(char[] ch, int start, int len)
    Report an XML comment anywhere in the document.
Parameters
    ch - An array holding the characters in the comment.
    start - The starting position in the array.
    len - The number of characters to use from the array.
```

Abstract Class *AbstractXMLPipe*

This class provides a bridge class to connect to existing content handlers and lexical handlers.

Inheritance

```
extends org.apache.cocoon.xml.AbstractXMLProducer
implements org.apache.cocoon.xml.XMLPipe
implements org.apache.avalon.excalibur.pool.Recyclable
```

Methods

```
public void setDocumentLocator(Locator locator)
    Receive an object for locating the origin of SAX document events.
Parameters
    locator - An object that can return the location of any SAX document
    event.
```

```
public void startDocument()
    Receive notification of the beginning of a document.

public void endDocument()
    Receive notification of the end of a document.

public void startPrefixMapping(String prefix, String uri)
    Begin the scope of a prefix-URI Namespace mapping.
Parameters
    prefix - The Namespace prefix being declared.
    uri - The Namespace URI the prefix is mapped to.

public void endPrefixMapping(String prefix)
    End the scope of a prefix-URI mapping.
Parameters
    prefix - The prefix that was being mapping.

public void startElement(String uri, String loc,
                    String raw, Attributes a)
    Receive notification of the beginning of an element.
Parameters
    uri - The Namespace URI, or the empty string if the element has no
    Namespace
         URI or if Namespace processing is not being performed.
    loc - The local name (without prefix), or the empty string if Namespace
         processing is not being performed.
    raw - The raw XML 1.0 name (with prefix), or the empty string if raw
    names
         are not available.
    a - The attributes attached to the element. If there are no attributes,
       it shall be an empty Attributes object.

public void endElement(String uri, String loc, String raw)
    Receive notification of the end of an element.
Parameters
    uri - The Namespace URI, or the empty string if the element has no
    Namespace
         URI or if Namespace processing is not being performed.
    loc - The local name (without prefix), or the empty string if Namespace
         processing is not being performed.
    raw - The raw XML 1.0 name (with prefix), or the empty string if raw
    names
        are not available.

public void characters(char[] c, int start, int len)
    Receive notification of character data.
Parameters
    c - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.

public void ignorableWhitespace(char[] c, int start, int len)
    Receive notification of ignorable whitespace in element content.
Parameters
    c - The characters from the XML document.
    start - The start position in the array.
```

```
    len - The number of characters to read from the array.

public void processingInstruction(String target, String data)
   Receive notification of a processing instruction.
Parameters
    target - The processing instruction target.
    data - The processing instruction data, or null if none was supplied.

public void skippedEntity(String name)
    Receive notification of a skipped entity.
Parameters
    name - The name of the skipped entity. If it is a parameter entity,
    the name will begin with '%'.

public void startDTD(String name, String publicId, String systemId)
    Report the start of DTD declarations, if any.
Parameters
    name - The document type name.
    publicId - The declared public identifier for the external DTD subset,
             or null if none was declared.
    systemId - The declared system identifier for the external DTD subset,
             or null if none was declared.
public void endDTD()
    Report the end of DTD declarations.

public void startEntity(String name)
    Report the beginning of an entity.
Parameters
    name - The name of the entity. If it is a parameter entity, the name
     will
    begin with '%'.

public void endEntity(String name)
    Report the end of an entity.
Parameters
    name - The name of the entity that is ending.

public void startCDATA()
    Report the start of a CDATA section.

public void endCDATA()
    Report the end of a CDATA section.

public void comment(char[] ch, int start, int len)
    Report an XML comment anywhere in the document.
Parameters
    ch - An array holding the characters in the comment.
    start - The starting position in the array.
    len - The number of characters to use from the array.
```

Abstract Class *AbstractXMLProducer*

This abstract class provides a default implementation of the methods specified by the `XMLProducer` interface.

Inheritance

```
extends org.apache.avalon.framework.logger.AbstractLoggable
implements org.apache.cocoon.xml.XMLProducer
```

Methods

```
public void setConsumer(XMLConsumer consumer)
    Set the XMLConsumer that will receive XML data.
    This method will simply call setContentHandler(consumer) and
    setLexicalHandler(consumer).

public void setContentHandler(ContentHandler handler)
    Set the ContentHandler that will receive XML data.
    Subclasses may retrieve this ContentHandler instance accessing the
     protected
    super.contentHandler field.

public void setLexicalHandler(LexicalHandler handler)
    Set the LexicalHandler that will receive XML data.
    Subclasses may retrieve this LexicalHandler instance accessing the
     protected
    super.lexicalHandler field.
Throws
    IllegalStateException - If the LexicalHandler or the XMLConsumer were
    already set.

public void recycle()
    Recycle the producer by removing references
```

Fields

```
protected xmlConsumer
    The XMLConsumer receiving SAX events.

protected contentHandler
    The ContentHandler receiving SAX events.

protected lexicalHandler
    The LexicalHandler receiving SAX events.
```

Package *org.apache.cocoon.xml.dom*

This package contains helper classes for dealing with DOM trees.

Interface *DOMBuilder.Listener*

The `Listener` interface must be implemented by objects that are willing to be notified of a successful DOM tree generation.

Methods

```
public void notify(Document doc)
    Receive notification of a successfully completed DOM tree generation.
```

Interface *DOMFactory*

This interface identifies classes producing instances of DOM `Document` objects.

Methods

```
public org.w3c.dom.Document newDocument()
    Create a new Document object.

public org.w3c.dom.Document newDocument(String name)
    Create a new Document object with a specified DOCTYPE.

public org.w3c.dom.Document newDocument(String name,
                                        String publicId,
                                        String systemId)
    Create a new Document object with a specified DOCTYPE, public ID and
     system ID.
```

Class *DOMBuilder*

`DOMBuilder` is a utility class that generates a W3C DOM document from SAX events.

Inheritance

```
implements org.apache.cocoon.xml.XMLConsumer
implements org.apache.avalon.framework.logger.Loggable
```

Constructors

```
protected DOMBuilder()
    Construct a new instance of this TreeGenerator.

public DOMBuilder(DOMFactory factory)
    Construct a new instance of this TreeGenerator.

public DOMBuilder(DOMFactory factory, DOMBuilder.Listener listener)
    Construct a new instance of this TreeGenerator.

public DOMBuilder(Node parentNode)
    Constructs a new instance that appends nodes to the given parent node.
    Note : you cannot use a Listener when appending to a Node, because the
    notification occurs at endDocument() which does not happen here.
```

Methods

```
public void setLogger(Logger logger)

public org.w3c.dom.Document getDocument()
    Return the newly built Document.

public void setDocumentLocator(Locator loc)
    Set the SAX Document Locator.
Parameters
    loc - The SAX Locator.

public void startDocument()
    Receive notification of the beginning of a document.
Throws
    SAXException - If this method was not called appropriately.

public void endDocument()
    Receive notification of the beginning of a document.
Throws
    SAXException - If this method was not called appropriately.

public void startDTD(String name, String publicId, String systemId)
    Report the start of DTD declarations, if any.
Parameters
    name - The document type name.
    publicId - The declared public identifier for the external DTD subset,
             or null if none was declared.
    systemId - The declared system identifier for the external DTD subset,
             or null if none was declared.
Throws
    SAXException - If this method was not called appropriately.

public void endDTD()
    Report the end of DTD declarations.
Parameters
    chars - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.
Throws
    SAXException - If this method was not called appropriately.

public void startElement(String uri, String loc,
                         String raw, Attributes a)
    Receive notification of the beginning of an element.
Throws
    SAXException - If this method was not called appropriately.

public void endElement(String uri, String loc, String raw)
    Receive notification of the end of an element.
Throws
    SAXException - If this method was not called appropriately.

public void startPrefixMapping(String prefix, String uri)
    Begin the scope of a prefix-URI Namespace mapping.
Parameters
    pre - The Namespace prefix being declared.
    uri - The Namespace URI the prefix is mapped to.
```

```
Throws
    SAXException - If this method was not called appropriately.

public void endPrefixMapping(String prefix)
    End the scope of a prefix-URI mapping.
Parameters
    prefix - The Namespace prefix that was being mapped.

public void startCDATA()
    Report the start of a CDATA section.
Throws
    SAXException - If this method was not called appropriately.

public void endCDATA()
    Report the end of a CDATA section.
Throws
    SAXException - If this method was not called appropriately.
public void startEntity(String name)
    Report the beginning of an entity.
Parameters
    chars - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.
Throws
    SAXException - If this method was not called appropriately.

public void endEntity(String name)
    Report the end of an entity.
Parameters
    chars - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.
Throws
    SAXException - If this method was not called appropriately.

public void characters(char[] chars, int start, int len)
    Receive notification of character data.
Parameters
    chars - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.
Throws
    SAXException - If this method was not called appropriately.

public void ignorableWhitespace(char[] chars, int start, int len)
    Receive notification of ignorable whitespace data.
Parameters
    chars - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.
Throws
    SAXException - If this method was not called appropriately.

public void processingInstruction(String target, String data)
    Receive notification of a processing instruction.
Parameters
    target - The processing instruction target.
```

```
    data - The processing instruction data.
Throws
    SAXException - If this method was not called appropriately.

public void comment(char[] chars, int start, int len)
    Report an XML comment anywhere in the document.
Parameters
    chars - The characters from the XML document.
    start - The start position in the array.
    len - The number of characters to read from the array.
Throws
   SAXException - If this method was not called appropriately.

public void skippedEntity(String name)
    Receive notification of a skipped entity.
Parameters
    name - The name of the skipped entity. If it is a parameter entity,
           the name will begin with '%'.

protected void notify(Document doc)
    Receive notification of a successfully completed DOM tree generation.
```

Fields

```
protected log

protected listener
    The listener

protected factory
    The document factory
```

Class *DOMStreamer*

`DOMStreamer` is a utility class that generates SAX events from a W3C DOM document.

Inheritance

```
extends org.apache.cocoon.xml.AbstractXMLProducer
```

Constructors

```
public DOMStreamer()
    Create a new DOMStreamer instance.

public DOMStreamer(XMLConsumer consumer)
    Create a new DOMStreamer instance.

public DOMStreamer(ContentHandler content)
    Create a new DOMStreamer instance.

public DOMStreamer(ContentHandler content, LexicalHandler lexical)
```

```
    Create a new DOMStreamer instance.
```

Methods

```
public void stream(Node node)
    Start the production of SAX events.
```

Fields

```
protected static factory
    The transformer factory shared by all instances

protected transformer
    The private transformer for this instance
```

SAX

The SAX (Simple API for XML) model is an event-based approach. The SAX model is a recommendation not hosted by the W3C, but it has reached the same acceptance. It is programming language-independent and defines a set of interfaces dealing with the various events occurring during XML parsing.

Package *org.xml.sax*

This is the core of the SAX model, containing all interfaces.

Interface *Attributes*

This is the interface for a list of XML attributes. This module, both source code and documentation, is in the public domain, and it comes with no warranty.

This interface allows access to a list of attributes in three different ways:

- By attribute index
- By namespace-qualified name
- By qualified (prefixed) name

The list does not contain attributes that were declared `#IMPLIED` but were not specified in the `start` tag. It also does not contain attributes used as namespace declarations (xmlns*) unless the http://xml.org/sax/features/namespace-prefixes feature is set to `true` (it is `false` by default).

If the `namespace-prefixes` feature is `false`, access by qualified name might not be available. If the http://xml.org/sax/features/namespaces feature is false, access by namespace-qualified names might not be available.

This interface replaces the now-deprecated SAX1 `@link org.xml.sax. AttributeList AttributeList` interface, which does not contain namespace support. In addition to namespace support, it adds the `getIndex` methods.

The order of attributes in the list is unspecified. It varies from implementation to implementation.

See also:

- `org.xml.sax.helpers.AttributeListImpl`

Methods

```
public int getLength()
    Return the number of attributes in the list.
    Once you know the number of attributes, you can iterate through the list.
Returns
    The number of attributes in the list.
See Also
    getURI(int)
    getLocalName(int)
    getQName(int)
    getType(int)
    getValue(int)

public String getURI(int index)
    Look up an attribute's Namespace URI by index.
Parameters
    index - The attribute index (zero-based).
Returns
    The Namespace URI, or the empty string if none is available, or null
     if
    the index is out of range.
See Also
    getLength

public String getLocalName(int index)
    Look up an attribute's local name by index.
Parameters
    index - The attribute index (zero-based).
Returns
    The local name, or the empty string if Namespace processing is not being
    performed, or null if the index is out of range.
See Also
    getLength

public String getQName(int index)
    Look up an attribute's XML 1.0 qualified name by index.
Parameters
    index - The attribute index (zero-based).
Returns
    The XML 1.0 qualified name, or the empty string if none is available,
    or null if the index is out of range.
```

```
See Also
    getLength

public String getType(int index)
    Look up an attribute's type by index.
    The attribute type is one of the strings "CDATA", "ID", "IDREF", "IDREFS",
    "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES", or "NOTATION"
    (always in upper case).
    If the parser has not read a declaration for the attribute, or if the
     parser
    does not report attribute types, then it must return the value "CDATA"
     as
    stated in the XML 1.0 Recommendation (clause 3.3.3,).
    For an enumerated attribute that is not a notation, the parser will report
    the type as "NMTOKEN".
Parameters
    index - The attribute index (zero-based).
Returns
    The attribute's type as a string, or null if the index is out of range.
See Also
    getLength

public String getValue(int index)
    Look up an attribute's value by index.
    If the attribute value is a list of tokens (IDREFS, ENTITIES, or
     NMTOKENS),
    the tokens will be concatenated into a single string with each token
    separated by a single space.
Parameters
    index - The attribute index (zero-based).
Returns
    The attribute's value as a string, or null if the index is out of range.
See Also
    getLength

public int getIndex(String uri, String localName)
    Look up the index of an attribute by Namespace name.
Parameters
    uri - The Namespace URI, or the empty string if the name has no
         Namespace URI.
    localName - The attribute's local name.
Returns
    The index of the attribute, or -1 if it does not appear in the list.

public int getIndex(String qName)
    Look up the index of an attribute by XML 1.0 qualified name.
Parameters
    qName - The qualified (prefixed) name.
Returns
    The index of the attribute, or -1 if it does not appear in the list.

public String getType(String uri, String localName)
    Look up an attribute's type by Namespace name.
    See getType(int) for a description of the possible types.
Parameters
    uri - The Namespace URI, or the empty String if the name has no
         Namespace URI.
```

```
    localName - The local name of the attribute.
Returns
    The attribute type as a string, or null if the attribute is not in the
    list or if Namespace processing is not being performed.

public String getType(String qName)
    Look up an attribute's type by XML 1.0 qualified name.
    See getType(int) for a description of the possible types.
Parameters
    qName - The XML 1.0 qualified name.
Returns
    The attribute type as a string, or null if the attribute is not in the
    list or if qualified names are not available.

public String getValue(String uri, String localName)
    Look up an attribute's value by Namespace name.
    See getValue(int) for a description of the possible values.
Parameters
    uri - The Namespace URI, or the empty String if the name has no
         Namespace URI.
    localName - The local name of the attribute.
Returns
    The attribute value as a string, or null if the attribute is not in the
     list.

public String getValue(String qName)
    Look up an attribute's value by XML 1.0 qualified name.
    See getValue(int) for a description of the possible values.
Parameters
    qName - The XML 1.0 qualified name.
Returns
    The attribute value as a string, or null if the attribute is not in the
    list or if qualified names are not available.
```

Interface *ContentHandler*

This interface receives notification of a document's logical content. This module, both source code and documentation, is in the public domain and comes with no warranty.

This is the main interface that most SAX applications implement. If the application needs to be informed of basic parsing events, it implements this interface and registers an instance with the SAX parser using the `setContentHandler` method. The parser uses the instance to report basic document-related events such as the start and end of elements and character data.

The order of events in this interface is very important. It mirrors the order of information in the document itself. For example, all an element's content (character data, processing instructions, and/or subelements) appears, in order, between the `startElement` event and the corresponding `endElement` event.

This interface is similar to the now-deprecated SAX 1.0 `DocumentHandler` interface, but it adds support for namespaces and for reporting skipped entities (in nonvalidating XML processors).

Implementers should note that there is also a Java class `ContentHandler` in the `java.net` package. This means that it's probably a bad idea to do this:

```
import java.net.*;
import org.xml.sax.*;
```

In fact, this is usually a sign of sloppy programming anyway, so you should consider this a feature rather than a bug.

See also:

- `org.xml.sax.XMLReader`
- `org.xml.sax.DTDHandler`
- `org.xml.sax.ErrorHandler`

Methods

```
public void setDocumentLocator(Locator locator)
    Receive an object for locating the origin of SAX document events.
    SAX parsers are strongly encouraged (though not absolutely required)
     to
    supply a locator: if it does so, it must supply the locator to the
    application by invoking this method before invoking any of the other
    methods in the ContentHandler interface.
    The locator allows the application to determine the end position of any
    document-related event, even if the parser is not reporting an error.
    Typically, the application will use this information for reporting its
     own
    errors (such as character content that does not match an application's
    business rules). The information returned by the locator is probably
     not
    sufficient for use with a search engine.
    Note that the locator will return correct information only during the
    invocation of the events in this interface. The application should not
    attempt to use it at any other time.
Parameters
    locator - An object that can return the location of any SAX document
     event.
See Also
    org.xml.sax.Locator

public void startDocument()
    Receive notification of the beginning of a document.
    The SAX parser will invoke this method only once, before any other methods
    in this interface or in DTDHandler (setDocumentLocator).
Throws
    org.xml.sax.SAXException - Any SAX exception, possibly wrapping another
```

exception.
See Also
endDocument

public void endDocument()
Receive notification of the end of a document.
The SAX parser will invoke this method only once, and it will be the last
method invoked during the parse. The parser shall not invoke this method
until it has either abandoned parsing (because of an unrecoverable error)
or reached the end of input.
Throws
org.xml.sax.SAXException - Any SAX exception, possibly wrapping another
exception.
See Also
startDocument
public void startPrefixMapping(String prefix, String uri)
Begin the scope of a prefix-URI Namespace mapping.
The information from this event is not necessary for normal Namespace
processing: the SAX XML reader will automatically replace prefixes for
element and attribute names when the
http://xml.org/sax/features/namespaces
feature is true (the default).
There are cases, however, when applications need to use prefixes in
character
data or in attribute values, where they cannot safely be expanded
automatically; the start/endPrefixMapping event supplies the
information to
the application to expand prefixes in those contexts itself, if
necessary. Note that start/endPrefixMapping events are not guaranteed
to be
properly nested relative to each-other: all startPrefixMapping events
will
occur before the corresponding startElement event, and all
endPrefixMapping events will occur after the corresponding endElement
event, but their order is not otherwise guaranteed. There should never
be
start/endPrefixMapping events for the "xml" prefix, since it is
predeclared
and immutable.
Parameters
prefix - The Namespace prefix being declared.
uri - The Namespace URI the prefix is mapped to.
Throws
org.xml.sax.SAXException - The client may throw an exception during
processing.
See Also
endPrefixMapping
startElement

public void endPrefixMapping(String prefix)
End the scope of a prefix-URI mapping.
See startPrefixMapping for details. This event will always occur after
the
corresponding endElement event, but the order of endPrefixMapping events
is
not otherwise guaranteed.

```
Parameters
    prefix - The prefix that was being mapping.
Throws
    org.xml.sax.SAXException - The client may throw an exception during
    processing.
See Also
    startPrefixMapping
    endElement

public void startElement(String namespaceURI,
                         String localName,
                         String qName,
                         Attributes atts)
    Receive notification of the beginning of an element.
    The Parser will invoke this method at the beginning of every element
    in the
    XML document; there will be a corresponding endElement event for every
    startElement event (even when the element is empty). All of the element's
    content will be reported, in order, before the corresponding endElement
    event. This event allows up to three name components for each element:
    the Namespace URI, the local name, and the qualified (prefixed) name.
    Any or all of these may be provided, depending on the values of the
    http://xml.org/sax/features/namespaces and the
    http://xml.org/sax/features/namespace-prefixes properties:
    - the Namespace URI and local name are required when the namespaces
    property
      is true (the default), and are optional when the namespaces property
    is
      false (if one is specified, both must be);
    - the qualified name is required when the namespace-prefixes property
    is
      true, and is optional when the namespace-prefixes property is false
      (the default).
    Note that the attribute list provided will contain only attributes with
    explicit values (specified or defaulted): #IMPLIED attributes will be
    omitted. The attribute list will contain attributes used for Namespace
    declarations (xmlns* attributes) only if the
    http://xml.org/sax/features/namespace-prefixes property is true (it is
    false
    by default, and support for a true value is optional).
Parameters
    uri - The Namespace URI, or the empty string if the element has no
         Namespace URI or if Namespace processing is not being performed.
    localName - The local name (without prefix), or the empty string if
              Namespace processing is not being performed.
    qName - The qualified name (with prefix), or the empty string if qualified
           names are not available.
    atts - The attributes attached to the element. If there are no attributes,
          it shall be an empty Attributes object.
Throws
    org.xml.sax.SAXException - Any SAX exception, possibly wrapping another
    exception.
See Also
    endElement
    org.xml.sax.Attributes

public void endElement(String namespaceURI,
```

```
                String localName,
                String qName)
    Receive notification of the end of an element.
    The SAX parser will invoke this method at the end of every element in
     the
    XML document; there will be a corresponding startElement event for every
    endElement event (even when the element is empty).
    For information on the names, see startElement.
Parameters
    uri - The Namespace URI, or the empty string if the element has no
         Namespace URI or if Namespace processing is not being performed.
    localName - The local name (without prefix), or the empty string if
              Namespace processing is not being performed.
    qName - The qualified XML 1.0 name (with prefix), or the empty string
           if qualified names are not available.
Throws
    org.xml.sax.SAXException - Any SAX exception, possibly wrapping another
    exception.
public void characters(char[] ch, int start, int length)
    Receive notification of character data.
    The Parser will call this method to report each chunk of character data.
    SAX parsers may return all contiguous character data in a single chunk,
    or they may split it into several chunks; however, all of the characters
    in any single event must come from the same external entity so that the
    Locator provides useful information.
    The application must not attempt to read from the array outside of the
    specified range.
    Note that some parsers will report whitespace in element content using
     the
    ignorableWhitespace method rather than this one (validating parsers must
    do so).
Parameters
    ch - The characters from the XML document.
    start - The start position in the array.
    length - The number of characters to read from the array.
Throws
    org.xml.sax.SAXException - Any SAX exception, possibly wrapping another
    exception.
See Also
    ignorableWhitespace
    org.xml.sax.Locator

public void ignorableWhitespace(char[] ch, int start, int length)
    Receive notification of ignorable whitespace in element content.
    Validating Parsers must use this method to report each chunk of whitespace
    in element content (see the W3C XML 1.0 recommendation, section 2.10):
    non-validating parsers may also use this method if they are capable of
    parsing and using content models.
    SAX parsers may return all contiguous whitespace in a single chunk, or
     they
    may split it into several chunks; however, all of the characters in any
    single event must come from the same external entity, so that the Locator
    provides useful information.
    The application must not attempt to read from the array outside of the
    specified range.
Parameters
    ch - The characters from the XML document.
```

```
    start - The start position in the array.
    length - The number of characters to read from the array.
Throws
    org.xml.sax.SAXException - Any SAX exception, possibly wrapping another
    exception.
See Also
    characters

public void processingInstruction(String target, String data)
    Receive notification of a processing instruction.
    The Parser will invoke this method once for each processing instruction
    found: note that processing instructions may occur before or after the
     main
    document element.
    A SAX parser must never report an XML declaration (XML 1.0, section 2.8)
     or a
    text declaration (XML 1.0, section 4.3.1) using this method.
Parameters
    target - The processing instruction target.
    data - The processing instruction data, or null if none was supplied.
          The data does not include any whitespace separating it from the
          target.
Throws
    org.xml.sax.SAXException - Any SAX exception, possibly wrapping another
    exception.

public void skippedEntity(String name)
    Receive notification of a skipped entity.
    The Parser will invoke this method once for each entity skipped.
    Non-validating processors may skip entities if they have not seen the
    declarations (because, for example, the entity was declared in an
    external DTD subset). All processors may skip external entities,
    depending on the values of the
    http://xml.org/sax/features/external-general-entities and the
    http://xml.org/sax/features/external-parameter-entities properties.
Parameters
    name - The name of the skipped entity. If it is a parameter entity, the
          name will begin with '%', and if it is the external DTD subset,
          it will be the string "[dtd]".
Throws
    org.xml.sax.SAXException - Any SAX exception, possibly wrapping another
exception.
```

Class *InputSource*

This is a single input source for an XML entity. This module, both source code and documentation, is in the public domain and comes with no warranty.

This class allows a SAX application to encapsulate information about an input source in a single object, which may include a public identifier, a system identifier, a byte stream (possibly with a specified encoding), and/or a character stream.

There are two places that the application delivers this input source to the parser: as the argument to the `Parser.parse` method, or as the return value of the `EntityResolver.resolveEntity` method.

The SAX parser uses the `InputSource` object to determine how to read XML input. If a character stream is available, the parser reads that stream directly. If not, the parser uses a byte stream, if available. If neither a character stream nor a byte stream is available, the parser attempts to open a URI connection to the resource identified by the system identifier.

An `InputSource` object belongs to the application. The SAX parser never modifies it in any way. (It may modify a copy if necessary.)

See also:

- `org.xml.sax.Parser parse`
- `org.xml.sax.EntityResolver resolveEntity`
- `java.io.InputStream`
- `java.io.Reader`

Constructors

```
public InputSource()
    Zero-argument default constructor.
See Also
    setPublicId
    setSystemId
    setByteStream
    setCharacterStream
    setEncoding

public InputSource(String systemId)
    Create a new input source with a system identifier.
    Applications may use setPublicId to include a public identifier as well,
    or setEncoding to specify the character encoding, if known.
    If the system identifier is a URL, it must be full resolved.
Parameters
    systemId - The system identifier (URI).
See Also
    setPublicId
    setSystemId
    setByteStream
    setEncoding
    setCharacterStream

public InputSource(InputStream byteStream)
    Create a new input source with a byte stream.
    Application writers may use setSystemId to provide a base for resolving
    relative URIs, setPublicId to include a public identifier, and/or
     setEncoding
    to specify the object's character encoding.
```

```
Parameters
    byteStream - The raw byte stream containing the document.
See Also
    setPublicId
    setSystemId
    setEncoding
    setByteStream
    setCharacterStream

public InputSource(Reader characterStream)
    Create a new input source with a character stream.
    Application writers may use setSystemId() to provide a base for resolving
    relative URIs, and setPublicId to include a public identifier.
    The character stream shall not include a byte order mark.
See Also
    setPublicId
    setSystemId
    setByteStream
    setCharacterStream
    setCharacterStream
```

Methods

```
public void setPublicId(String publicId)
    Set the public identifier for this input source.
    The public identifier is always optional: if the application writer
    includes
    one, it will be provided as part of the location information.
Parameters
    publicId - The public identifier as a string.
See Also
    getPublicId
    org.xml.sax.Locator#getPublicId
    org.xml.sax.SAXParseException#getPublicId

public String getPublicId()
    Get the public identifier for this input source.
Returns
    The public identifier, or null if none was supplied.
See Also
    setPublicId

public void setSystemId(String systemId)
    Set the system identifier for this input source.
    The system identifier is optional if there is a byte stream or a character
    stream, but it is still useful to provide one, since the application
    can use
    it to resolve relative URIs and can include it in error messages and
    warnings
    (the parser will attempt to open a connection to the URI only if there
    is no
    byte stream or character stream specified).
    If the application knows the character encoding of the object pointed
    to by
    the system identifier, it can register the encoding using the setEncoding
```

method. If the system ID is a URL, it must be fully resolved.
Parameters
systemId - The system identifier as a string.
See Also
setEncoding
getSystemId
org.xml.sax.Locator#getSystemId
org.xml.sax.SAXParseException#getSystemId

public String getSystemId()
Get the system identifier for this input source.
The getEncoding method will return the character encoding of the object pointed to, or null if unknown.
If the system ID is a URL, it will be fully resolved.
Returns
The system identifier, or null if none was supplied.
See Also
setSystemId
getEncoding

public void setByteStream(InputStream byteStream)
Set the byte stream for this input source.
The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself. If the application knows the character encoding of the
byte stream, it should set it with the setEncoding method.
Parameters
byteStream - A byte stream containing an XML document or other entity.
See Also
setEncoding
getByteStream
getEncoding
java.io.InputStream

public java.io.InputStream getByteStream()
Get the byte stream for this input source.
The getEncoding method will return the character encoding for this byte stream, or null if unknown.
Returns
The byte stream, or null if none was supplied.
See Also
getEncoding
setByteStream

public void setEncoding(String encoding)
Set the character encoding, if known.
The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML 1.0 recommendation).
This method has no effect when the application provides a character stream.
Parameters
encoding - A string describing the character encoding.
See Also
setSystemId
setByteStream
getEncoding

```
public String getEncoding()
    Get the character encoding for a byte stream or URI.
Returns
    The encoding, or null if none was supplied.
See Also
    setByteStream
    getSystemId
    getByteStream

public void setCharacterStream(Reader characterStream)
    Set the character stream for this input source.
    If there is a character stream specified, the SAX parser will ignore
     any byte
    stream and will not attempt to open a URI connection to the system
    identifier.
Parameters
    characterStream - The character stream containing the XML document
    or other entity.
See Also
    getCharacterStream
    java.io.Reader

public java.io.Reader getCharacterStream()
    Get the character stream for this input source.
Returns
    The character stream, or null if none was supplied.
See Also
    setCharacterStream
```

Class *SAXException*

Encapsulate a general SAX error or warning. This module, both source code and documentation, is in the public domain and comes with no warranty.

This class can contain basic error or warning information from either the XML parser or the application. A parser writer or application writer can subclass it to provide additional functionality. SAX handlers may throw this exception or any exception subclassed from it.

If the application needs to pass through other types of exceptions, it must wrap those exceptions in a `SAXException` or an exception derived from a `SAXException`. If the parser or application needs to include information about a specific location in an XML document, it should use the `@link org.xml.sax.SAXParseException SAXParseException` subclass.

See also:

- `org.xml.sax.SAXParseException`

Inheritance

```
extends Exception
```

Constructors

```
public SAXException()
    Create a new SAXException.

public SAXException(String message)
    Create a new SAXException.
Parameters
    message - The error or warning message.
See Also
    org.xml.sax.Parser#setLocale
  public SAXException(Exception e)
    Create a new SAXException wrapping an existing exception.
    The existing exception will be embedded in the new one, and its message
     will
    become the default message for the SAXException.
Parameters
    e - The exception to be wrapped in a SAXException.

public SAXException(String message, Exception e)
    Create a new SAXException from an existing exception.
    The existing exception will be embedded in the new one, but the new
     exception
    will have its own message.
Parameters
    message - The detail message.
    e - The exception to be wrapped in a SAXException.
See Also
    org.xml.sax.Parser#setLocale
```

Methods

```
public String getMessage()
    Return a detail message for this exception.
    If there is an embedded exception, and if the SAXException has no detail
    message of its own, this method will return the detail message from the
    embedded exception.
Returns
    The error or warning message.
See Also
    org.xml.sax.Parser#setLocale

public Exception getException()
    Return the embedded exception, if any.
Returns
    The embedded exception, or null if there is none.

public String toString()
    Override toString to pick up any embedded exception.
Returns
    A string representation of this exception.
```

Package *org.xml.sax.ext*

This package contains some optional extensions to the SAX core.

Interface *LexicalHandler*

This is a SAX2 extension handler for lexical events. This module, both source code and documentation, is in the public domain and comes with no warranty.

This is an optional extension handler for SAX2 to provide lexical information about an XML document, such as comments and CDATA section boundaries. XML readers are not required to support this handler, and it is not part of the core SAX2 distribution.

The events in the lexical handler apply to the entire document, not just to the document element, and all lexical handler events must appear between the content handler's `startDocument` and `endDocument` events.

To set the `LexicalHandler` for an XML reader, use the `@link org.xml.sax.XMLReader#setProperty setProperty` method with the `propertyId` "http://xml.org/sax/properties/lexical-handler". If the reader does not support lexical events, it throws a `@link org.xml.sax.SAXNotRecognizedException SAXNotRecognizedException` or a `@link org.xml.sax.SAXNotSupportedException SAXNotSupportedException` when you attempt to register the handler.

See also:

- `org.xml.sax.XMLReader#setProperty`
- `org.xml.sax.SAXNotRecognizedException`
- `org.xml.sax.SAXNotSupportedException`

Methods

```
public void startDTD(String name, String publicId, String systemId)
    Report the start of DTD declarations, if any.
    This method is intended to report the beginning of the DOCTYPE
    declaration;
    if the document has no DOCTYPE declaration, this method will not be
    invoked.
    All declarations reported through DTDHandler or DeclHandler events must
    appear between the startDTD and endDTD events. Declarations are assumed
    to belong to the internal DTD subset unless they appear between
    startEntity and endEntity events. Comments and processing instructions
    from the DTD should also be reported between the startDTD and endDTD
    events, in their original order of (logical) occurrence; they are not
    required to appear in their correct locations relative to DTDHandler
    or DeclHandler events, however. Note that the start/endDTD events will
    appear within the start/endDocument events from ContentHandler and
    before the first startElement event.
```

```
Parameters
    name - The document type name.
    publicId - The declared public identifier for the external DTD subset,
             or null if none was declared.
    systemId - The declared system identifier for the external DTD subset,
             or null if none was declared.
Throws
    SAXException - The application may raise an exception.
See Also
    endDTD
    startEntity

public void endDTD()
    Report the end of DTD declarations.
    This method is intended to report the end of the DOCTYPE declaration;
    if the document has no DOCTYPE declaration, this method will not be
     invoked.
Throws
    SAXException - The application may raise an exception.
See Also
    startDTD

public void startEntity(String name)
    Report the beginning of some internal and external XML entities.
    The reporting of parameter entities (including the external DTD subset)
     is optional, and SAX2 drivers that support LexicalHandler may not support
     it; you can use the http://xml.org/sax/features/lexical-handler/
    parameter-entities feature to query or control the reporting of
    parameter entities. General entities are reported with their regular
    names, parameter entities have '%' prepended to their names, and the
    external DTD subset has the pseudo-entity name "[dtd]".
    When a SAX2 driver is providing these events, all other events must be
    properly nested within start/end entity events. There is no additional
    requirement that events from DeclHandler or DTDHandler be properly
     ordered. Note that skipped entities will be reported through the
     skippedEntity event, which is part of the ContentHandler interface.
    Because of the streaming event model that SAX uses, some entity boundaries
    cannot be reported under any circumstances:
    - general entities within attribute values
    - parameter entities within declarations
    These will be silently expanded, with no indication of where the original
    entity boundaries were.
    Note also that the boundaries of character references (which are not
     really entities anyway) are not reported.
    All start/endEntity events must be properly nested.
Parameters
    name - The name of the entity. If it is a parameter entity, the name
     will begin with '%', and if it is the external DTD subset, it will be
     "[dtd]".
Throws
    SAXException - The application may raise an exception.
See Also
    endEntity
    org.xml.sax.ext.DeclHandler#internalEntityDecl
    org.xml.sax.ext.DeclHandler#externalEntityDecl

public void endEntity(String name)
```

```
    Report the end of an entity.
Parameters
    name - The name of the entity that is ending.
Throws
    SAXException - The application may raise an exception.
See Also
    #startEntity

public void startCDATA()
    Report the start of a CDATA section.
    The contents of the CDATA section will be reported through the regular
    characters event; this event is intended only to report the boundary.
Throws
    SAXException - The application may raise an exception.
See Also
    #endCDATA

public void endCDATA()
    Report the end of a CDATA section.
Throws
    SAXException - The application may raise an exception.
See Also
    #startCDATA

public void comment(char[] ch, int start, int length)
    Report an XML comment anywhere in the document.
    This callback will be used for comments inside or outside the document
    element, including comments in the external DTD subset (if read).
    Comments in the DTD must be properly nested inside start/endDTD and
    start/endEntity events (if used).
Parameters
    ch - An array holding the characters in the comment.
    start - The starting position in the array.
    length - The number of characters to use from the array.
Throws
    SAXException - The application may raise an exception.
```

Appendix C. Links on the Web

There is a lot of additional information about Cocoon and related subjects available on the web. For the most part, we kept web links out of the text because we felt that it would make more sense to collect them in an appendix at the end.

Links are listed according to the first chapter they appear in. Some chapters are not covered because they have no relevant links that need to be listed.

Chapter 1, "An Introduction to Internet Applications"

History of the Web

http://www.w3.org/History.html

Internet Timeline

http://www.zakon.org/robert/internet/timeline/

Introduction to CGI

http://hoohoo.ncsa.uiuc.edu/cgi/intro.html

History of the Apache Server

http://httpd.apache.org/ABOUT_APACHE.html

History of Microsoft's IIS

http://www.webdeveloper.com/servers/servers_dynamic_iis.html

Chapter 2, “Building the Machine Web with XML”

Introducing Tim Berners Lee

http://www.w3.org/People/Berners-Lee/ShortHistory.html

W3C Home Page

http://www.w3.org/

XML Specifications and Documentation

http://www.w3.org/XML/

XSLT Specification

http://www.w3.org/TR/xslt

Xpath Specification

http://www.w3.org/TR/xpath

Apache Xalan Project: Java Version

http://xml.apache.org/xalan-j/index.html

Apache Xerces Project: Java Version

http://xml.apache.org/xerces-j/index.html

LotusXSL and IBM XML4J

http://www.alphaworks.ibm.com/aw.nsf/FAQs/lotusxsl

Introduction to Native XML Databases

http://www.xml.com/lpt/a/2001/10/31/nativexmldb.html

List of XML Database Products

http://www.rpbourret.com/xml/XMLDatabaseProds.htm

Apache XML Database: Xindice (Formerly dbXML)

http://xml.apache.org/xindice/

Software AG Tamino XML Database

http://www.softwareag.com/taminoplatform/

Oracle and XML

http://www.oracle.com/xml/

Moreover News Provider

http://www.moreover.com

Apache Cocoon Project

http://xml.apache.org/cocoon/

Chapter 3, "Getting Started with Cocoon"

Cocoon Distribution Directory

http://xml.apache.org/cocoon/dist/

Sun Java SDK

http://java.sun.com/j2se/

Apache Tomcat Servlet Engine Home Page

http://jakarta.apache.org/tomcat/index.html

Cocoon Mailing Lists

http://xml.apache.org/cocoon/mail-lists.html

Mailing Lists Archive

http://marc.theaimsgroup.com/

Cocoon Frequently Asked Questions (FAQs)

http://xml.apache.org/cocoon/faq.html

Information on Installing Cocoon

http://xml.apache.org/cocoon/installing/index.html

Cocoon CVS

http://cvs.apache.org/viewcvs.cgi/xml-cocoon2/

Chapter 4, "Putting Cocoon to Work"

Apache FOP Project Home Page

http://xml.apache.org/fop/index.html

Scalable Vector Graphics (SVG) Format

http://www.w3.org/Graphics/SVG/

Apache SVG Project

http://xml.apache.org/batik/index.html

XSL Formatting Objects

http://www.w3.org/TR/xslt/

JTidy Project

http://sourceforge.net/projects/jtidy/

Chapter 5, "Cocoon News Portal: Entry Version"

Resource Description Framework (RDF) Specification

http://www.w3.org/TR/REC-rdf-syntax/

RDF Site Summary Specification

http://groups.yahoo.com/group/rss-dev/files/specification.html

LinuxToday

http://www.linuxtoday.com/

Chapter 7, "Cocoon News Portal: Extended Version"

HSQL Database Project

http://sourceforge.net/projects/hsqldb/

Chapter 8, "A Developer's Look at the Cocoon Architecture"

Apache Avalon Project

http://jakarta.apache.org/avalon/

Developing with Apache Avalon

http://jakarta.apache.org/avalon/developing/index.html

Software Design Patterns

http://www.patterndepot.com

SAX Home Page

http://www.saxproject.org/

Chapter 9, "Developing Components for Cocoon"

JavaMail API

http://java.sun.com/products/javamail/index.html

JavaBeans Activation Framework

http://java.sun.com/products/javabeans/glasgow/jaf.html

Chapter 11, “Designing Cocoon Applications”

XMLSpy

http://www.xmlspy.com/

XMetal

http://www.xmetal.com

CookTop

http://www.xmlcooktop.com