APPENDIX E

The AxKit XML Application Server

AxKit is an XML application server written using the mod_perl framework. At its core, AxKit provides the developer with many ways to set up server-side XML transformations. This allows you to rapidly develop sites that use XML, allowing delivery of the same content in different formats. It also allows you to change the layout of your site very easily, due to the forced separation of content from presentation.

This appendix gives an overview of the ways you can put AxKit to use on your mod_perl-enabled server. It is not a complete description of all the capabilities of AxKit. For more detailed information, please take a look at the documentation provided on the AxKit web site at *http://axkit.org/*. Commercial support and consultancy services for AxKit also are available at this site.

There are a number of benefits of using XML for content delivery:

- Perhaps the most obvious benefit is the longevity of your data. XML is a format that is going to be around for a very long time, and if you use XML, your data (the content of your site) can be processed using standard tools for multiple platforms and languages for years to come.
- If you use XSLT as a templating solution, you can pick from a number of different implementations. This allows you to easily switch between tools that best suit your task at hand.
- XSLT takes a fundamentally different approach to templating than almost every other Perl templating solution. Rather than focusing on "sandwiching" the data into the template at various positions, XSLT transforms a tree representation of your data into another tree. This not only makes the output (in the case of HTML) less prone to mismatched tags, but it also makes chained processing, in which the output of one transformation becomes the input of another, a lot simpler and faster.

Installing and Configuring AxKit

There are many configuration options that allow you to customize your AxKit installation, but in this section we aim to get you started as quickly and simply as possible. This appendix assumes you already have mod_perl and Apache installed and working. See Chapter 3 if this is not the case. This section does not cover installing AxKit on Win32 systems, for which there is an ActiveState package at *ftp://theoryx5.uwinnipeg.ca/pub/other/ppd/*.

First download the latest version of AxKit, which you can get either from your local CPAN archive or from the AxKit download directory at *http://axkit.org/*. Then type the following:

```
panic% gunzip -c AxKit-x.xx.tar.gz | tar xvf -
panic% cd AxKit-x.xx.tar.gz
panic% perl Makefile.PL
panic% make
panic% make test
panic% su
panic# make install
```

If Perl's *Makefile.PL* warns you about missing modules, notably `XML::XPath`, make a note of the missing modules and install them from CPAN. AxKit will run without the missing modules, but without `XML::XPath` it will be impossible to run the examples in this appendix.*

Now we need to add some simple options to the very end of our *httpd.conf* file:

```
PerlModule AxKit
SetHandler perl-script
PerlHandler AxKit
AxDebugLevel 10
PerlSetVar AxXPSInterpolate 1
```

This configuration makes it look as though AxKit will deliver all of your files, but don't worry: if it doesn't detect XML at the URL you supply, it will let *httpd* deliver the content. If you're still concerned, put all but the first configuration directive in a `<Location>` section. Note that the first line, `PerlModule AxKit`, must appear in *httpd.conf* outside of any runtime configuration blocks. Otherwise, Apache cannot see the AxKit configuration directives and you will get errors when you try to start *httpd*.

Now, assuming you have `XML::XPath` installed (try *perl -MXML::XPath -e0* on the command line to check), restart Apache. You are now ready to begin publishing transformed XML with AxKit!

* AxKit is very flexible in how it lets you transform the XML on the server, and there are many modules you can plug in to AxKit to allow you to do these transformations. For this reason, the AxKit installation does not mandate any particular modules to use. Instead, it will simply suggest modules that might help when you install AxKit.

Your First AxKit Page

Now we're going to see how AxKit works, by transforming an XML file containing data about Camelids (note the dubious Perl reference) into HTML.

First you will need a sample XML file. Open the text editor of your choice and type the code shown in Example E-1.

Example E-1. firstxml.xml

```
<?xml version="1.0"?>
<dromedaries>
  <species name="Camel">
    <humps>1 or 2</humps>
    <disposition>Cranky</disposition>
  </species>
  <species name="Llama">
    <humps>1</humps>
    <disposition>Aloof</disposition>
  </species>
  <species name="Alpaca">
    <humps>(see Llama)</humps>
    <disposition>Friendly</disposition>
  </species>
</dromedaries>
```

Save this file in your web server document root (e.g., */home/httpd/httpd_perl/htdocs/*) as *firstxml.xml*.

Now we need a stylesheet to transform the XML to HTML. For this first example we are going to use XPathScript, an XML transformation language specific to AxKit. Later we will give a brief introduction to XSLT.

Create a new file and type the code shown in Example E-2.

Example E-2. firstxml.xps

```
<%
$t->{'humps'}{pre} = "<td>";
$t->{'humps'}{post} = "</td>";
$t->{'disposition'}{pre} = "<td>";
$t->{'disposition'}{post} = "</td>";
$t->{'species'}{pre} = "<tr><td>{\@name}</td>";
$t->{'species'}{post} = "</tr>";
%>
<html>
<head>
<title>Know Your Dromedaries</title>
</head>
<body>
  <table border="1">
    <tr><th>Species</th>
        <th>No. of Humps</th>
```

Example E-2. firstxml.xps (continued)

```
        <th>Disposition</th></tr>
    <%= apply_templates('/dromedaries/species') %>
  </table>
</body>
</html>
```

Save this file as *firstxml.xps*.

Now to get the original file, *firstxml.xml*, to be transformed on the server by *text.xps*, we need to somehow associate that file with the stylesheet. Under AxKit there are a number of ways to do that, with varying flexibility. The simplest way is to edit your *firstxml.xml* file and, immediately after the `<?xml version="1.0"?>` declaration, add the following:

```
<?xml-stylesheet href="firstxml.xps"
                 type="application/x-xpathscript"?>
```

Now assuming the files are both in the same directory under your *httpd* document root, you should be able to make a request for *text.xml* and see server-side transformed XML in your browser. Now try changing the source XML file, and watch AxKit detect the change next time you load the file in the browser.

If Something Goes Wrong

If you don't see HTML in your browser but instead get the source XML, you will need to check your error log. (In Internet Explorer you will see a tree-based representation of the XML, and in Mozilla, Netscape, or Opera you will see all the text of the document joined together.)

AxKit sends out varying amounts of debug information depending on the value of `AxDebugLevel` (which we set to the maximum value of `10`). If you can't decipher the contents of the error log, contact the AxKit user's mailing list at *axkit-users@axkit.org* with details of your problem.

How it Works?

The stylesheet above specifies how the various tags work. The ASP `<% %>` syntax delimits Perl code from HTML. You can execute any code within the stylesheet.

In this example, we use the special XPathScript `$t` hash reference, which specifies the names of tags and how they should be output to the browser. There are several options for the second level of the hash, and here we see two of those options: `pre` and `post`. `pre` and `post` specify (respectfully) what appears before the tag and what appears after it. These values in `$t` take effect only when we call the `apply_templates()` function, which iterates over the nodes in the XML, executing the matching values in `$t`.

XPath

One of the key specifications being used in XML technologies is XPath. This is a little language used within other languages for selecting nodes within an XML document (just as regular expressions is a language of its own within Perl). The initial appearance of an XPath is similar to that of a Unix directory path. In Example E-2 we can see the XPath */dromedaries/species*, which starts at the root of the document, finds the *dromedaries* root element, then finds the species *children* of the *dromedaries* element. Note that unlike Unix directory paths, XPaths can match multiple nodes; so in the case above, we select all of the *species* elements in the document.

Documenting all of XPath here would take up many pages. The grammar for XPath allows many constructs of a full programming language, such as functions, string literals, and Boolean expressions. What's important to know is that the syntax we are using to find nodes in our XML documents is not just something invented for AxKit!

Dynamic Content

AxKit has a flexible tool called *eXtensible Server Pages* (XSP) for creating XML from various data sources such as relational databases, cookies, and form parameters. This technology was originally invented by the Apache Cocoon team, and AxKit shares their syntax. This allows easier migration of projects to and from Cocoon. (Cocoon allows you to embed Java code in your XSP, similar to how AxKit allows you to embed Perl code.)

XSP is an XML-based syntax that uses namespaces to provide extensibility. In many ways, this is like the Cold Fusion model of using tags to provide dynamic functionality. One of the advantages of using XSP is that it is impossible to generate invalid XML, which makes it ideal for use in an XML framework such as AxKit. Another is that the tags can hide complex functionality, allowing the XSP tags to be added by designers and freeing programmers to perform more complex and more cost-effective tasks.

The XSP framework allows you to design new tags, or use ones provided already by others on CPAN. These extra tags are called *taglibs*. By using taglibs instead of embedding Perl code in your XSP page, you can further build on AxKit's separation of content from presentation by separating out logic too. And creating new taglibs is almost trivial using AxKit's `TagLibHelper` module, which hides all the details for you.

In the examples below, we are going to show some code that embeds Perl code in the XSP pages. This is not a recommended practice, due to the ease with which you can extract functionality into tag libraries. However, it is more obvious to Perl programmers what is going on this way and provides a good introduction to the technology.

Handling Form Parameters

The `AxKit::XSP::Param` taglib allows you to easily read form and query string parameters within an XSP page. The following example shows how a page can submit back to itself. To allow this to work, add the following to your *httpd.conf* file:

```
AxAddXSPTaglib AxKit::XSP::Param
```

The XSP page is shown in Example E-3.

Example E-3. paramtaglib.xsp

```
<xsp:page
 xmlns:xsp="http://apache.org/xsp/core/v1"
 xmlns:param="http://axkit.org/NS/xsp/param/v1"
 language="Perl"
>
<page>
  <xsp:logic>
  if (<param:name/>) {
    <xsp:content>
     Your name is: <param:name/>
    </xsp:content>
  }
  else {
    <xsp:content>
      <form>
        Enter your name: <input type="text" name="name" />
        <input type="submit"/>
      </form>
    </xsp:content>
  }
  </xsp:logic>
</page>
</xsp:page>
```

The most significant thing about this example is how we freely mix XML tags with our Perl code, and the XSP processor figures out the right thing to do depending on the context. The only requirement is that the XSP page itself must be valid XML. That is, the following would generate an error:

```
<xsp:logic>
my $page = <param:page/>;
if ($page < 3) { # ERROR: less-than is a reserved character in XML
 ...
}
</xsp:logic>
```

We need to convert this to valid XML before XSP can handle it. There are a number of ways to do so. The simplest is just to reverse the expression to `if (3 > $page)`, because the greater-than sign is valid within an XML text section. Another way is to encode the less-than sign as `<`, which will be familiar to HTML authors.

The other thing to notice is the <xsp:logic> and <xsp:content> tags. The former defines a section of Perl code, while the latter allows you to go back to processing the contents as XML output. Also note that the <xsp:content> tag is not always needed. Because the XSP engine inherently understands XML, you can omit the <xsp:content> tag when the immediate child would be an element, rather than text. For example, the following example requires the <xsp:content> tag:

```
<xsp:logic>
if (<param:name/>) {
  # xsp:content needed
  <xsp:content>
  Your name is: <param:name/>
  </xsp:content>
}
</xsp:logic>
```

But if you rewrote it like this, it wouldn't, because of the surrounding non-XSP tag:

```
<xsp:logic>
if (<param:name/>) {
  # no xsp:content tag needed
  <p>Your name is: <param:name/></p>
}
</xsp:logic>
```

Note that the initial example, when processed by only the XSP engine, will output the following XML:

```
<page>
  <form>
    Enter your name: <input type="text" name="name" />
    <input type="submit"/>
  </form>
</page>
```

This needs to be processed with XSLT or XPathScript to be reasonably viewable in a browser. However, the point is that you can reuse the above page as either HTML or WML just by applying different stylesheets.

Handling Cookies

AxKit::XSP::Cookie is a taglib interface to Apache::Cookie (part of the *libapreq* package). The following example demonstrates both retrieving and setting a cookie from within XSP. In order for this to run, the following option needs to be added to your *httpd.conf* file:

```
AxAddXSPTaglib AxKit::XSP::Cookie
```

The XSP page is shown in Example E-4.

Example E-4. cookietaglib.xsp

```
<xsp:page
 xmlns:xsp="http://apache.org/xsp/core/v1"
 xmlns:cookie="http://axkit.org/NS/xsp/cookie/v1"
 language="Perl"
>
<page>
  <xsp:logic>
  my $value;
  if ($value = <cookie:fetch name="count"/>) {
    $value++;
  }
  else {
    $value = 1;
  }
  </xsp:logic>
  <cookie:create name="count">
    <cookie:value><xsp:expr>$value</xsp:expr></cookie:value>
  </cookie:create>
  <p>Cookie value: <xsp:expr>$value</xsp:expr></p>
</page>
</xsp:page>
```

This page introduces the concept of XSP *expressions*, using the `<xsp:expr>` tag. In XSP, everything that returns a value is an expression of some sort. In the last two examples, we have used a taglib tag within a Perl `if( )` statement. These tags are both expressions, even though they don't use the `<xsp:expr>` syntax. In XSP, everything understands its context and tries to do the right thing. The following three examples will all work as expected:

```
<cookie:value>3</cookie:value>

<cookie:value><xsp:expr>2 + 1</xsp:expr></cookie:value>

<cookie:value><param:cookie_value/></cookie:value>
```

We see this as an extension of how Perl works—the idea of "Do What I Mean," or DWIM.

Sending Email

With the `AxKit::XSP::Sendmail` taglib, it is very simple to send email from an XSP page. This taglib combines email-address verification, using the `Email::Valid` module, with email sending, using the `Mail::Sendmail` module (which will either interface to an SMTP server or use the *sendmail* executable directly). Again, to allow usage of this taglib, the following line must be added to *httpd.conf*:

```
AxAddXSPTaglib AxKit::XSP::Sendmail
```

Then sending email from XSP is as simple as what's shown in Example E-5.

Example E-5. sendmailtaglib.xsp

```
<xsp:page
 xmlns:xsp="http://apache.org/xsp/core/v1"
 xmlns:param="http://axkit.org/NS/xsp/param/v1"
 xmlns:mail="http://axkit.org/NS/xsp/sendmail/v1"
 language="Perl"
>
<page>
  <xsp:logic>
  if (!<param:email/>) {
    <p>You forgot to supply an email address!</p>
  }
  else {
    my $to;
    if (<param:subopt/> eq "sub") {
      $to = "axkit-users-subscribe@axkit.org";
    }
    elsif (<param:subopt/> eq "unsub") {
      $to = "axkit-users-unsubscribe@axkit.org";
    }
    <mail:send-mail>
     <mail:from><param:user_email/></mail:from>
     <mail:to><xsp:expr>$to</xsp:expr></mail:to>
     <mail:body>
      Subscribe or Unsubscribe <param:user_email/>
     </mail:body>
    </mail:send-mail>
    <p>(un)subscription request sent</p>
  }
  </xsp:logic>
</page>
</xsp:page>
```

The only thing missing here is some sort of error handling. When the *sendmail* taglib detects an error (either in an email address or in sending the email), it throws an exception.

Handling Exceptions

The exception taglib, `AxKit::XSP::Exception`, is used to catch exceptions. The syntax is very simple: rather than allowing different types of exceptions, it is currently a very simple `try`/`catch` block. To use the exceptions taglib, the following has to be added to *httpd.conf*:

```
AxAddXSPTaglib AxKit::XSP::Exception
```

Then you can implement form validation using exceptions, as Example E-6 demonstrates.

Example E-6. exceptiontaglib.xsp

```
<xsp:page
 xmlns:xsp="http://apache.org/xsp/core/v1"
 xmlns:param="http://axkit.org/NS/xsp/param/v1"
 xmlns:except="http://axkit.org/NS/xsp/exception/v1"
 language="Perl"
>
<page>
 # form validation:
 <except:try>
  <xsp:logic>
  if ((<param:number/> > 10) || (0 > <param:number/>)) {
    die "Number must be between 0 and 10";
  }
  if (!<param:name/>) {
    die "You must supply a name";
  }
  # Now do something with the params
  </xsp:logic>
  <p>Values saved successfully!</p>
  <except:catch>
   <p>Sorry, the values you entered were
      incorrect: <except:message/></p>
  </except:catch>
 </except:try>
</page>
```

The exact same `try`/`catch` (and message) tags can be used for *sendmail* and for ESQL (discussed in a moment).

Utilities Taglib

The `AxKit::XSP::Util` taglib includes some utility methods for including XML from the filesystem, from a URI, or as the return value from an expression. (Normally an expression would be rendered as plain text, so a "`<`" character would be encoded as "`<`"). The AxKit utilities taglib is a direct copy of the Cocoon utilities taglib, and as such uses the same namespace as the Cocoon Util taglib, `http://apache.org/xsp/util/v1`.

Executing SQL

Perhaps the most interesting taglib of all is the ESQL taglib, which allows you to execute SQL queries against a DBI-compatible database and provides access to the column return values as strings, scalars, numbers, dates, or even as XML. (Returning XML requires the utilities taglib.) Like the *sendmail* taglib, the ESQL taglib throws exceptions when an error occurs.

One point of interest about the ESQL taglib is that it is a direct copy of the Cocoon ESQL taglib. There are only a few minor differences between the two, such as how

columns of different types are returned and how errors are trapped.* Having nearly identical taglibs helps you to port projects to or from Cocoon. As with all the other taglibs, ESQL requires the addition of the following to your *httpd.conf* file:

```
AxAddXSPTaglib AxKit::XSP::ESQL
```

Example E-7 uses ESQL to read data from an address-book table. This page demonstrates that it is possible to reuse the same code for both our list of addresses and viewing a single address in detail.

Example E-7. esqltaglib.xsp

```
<xsp:page
 language="Perl"
 xmlns:xsp="http://apache.org/xsp/core/v1"
 xmlns:esql="http://apache.org/xsp/SQL/v2"
 xmlns:except="http://axkit.org/NS/xsp/exception/v1"
 xmlns:param="http://axkit.org/NS/xsp/param/v1"
 indent-result="no"
>
<addresses>
 <esql:connection>
  <esql:driver>Pg</esql:driver>
  <esql:dburl>dbname=phonebook</esql:dburl>
  <esql:username>postgres</esql:username>
  <esql:password></esql:password>
  <except:try>
  <esql:execute-query>
   <xsp:logic>
   if (<param:address_id/>) {
    <esql:query>
     SELECT * FROM address WHERE id =
     <esql:parameter><param:address_id/></esql:parameter>
    </esql:query>
   }
   else {
    <esql:query>
     SELECT * FROM address
    </esql:query>
   }
   </xsp:logic>
   <esql:results>
    <esql:row-results>
     <address>
      <esql:get-columns/>
     </address>
    </esql:row-results>
   </esql:results>
  </esql:execute-query>
```

* In Cocoon there are ESQL tags for trapping errors, whereas AxKit uses exceptions.

Example E-7. esqltaglib.xsp (continued)

```
  <except:catch>
   Error Occured: <except:message/>
  </except:catch>
  </except:try>
 </esql:connection>
</addresses>
</xsp:page>
```

The result of running the above through the XSP processor is:

```
<addresses>
 <address>
  <id>2</id>
  <last_name>Sergeant</last_name>
  <first_name>Matt</first_name>
  <title>Mr</title>
  <company>AxKit.com Ltd</company>
  <email>matt@axkit.com</email>
  <classification_id>1</classification_id>
 </address>
</addresses>
```

More XPathScript Details

XPathScript aims to provide the power and flexibility of XSLT as an XML transformation language, without the restriction of XSLT's XML-based syntax. Unlike XSLT, which has special modes for outputting in text, XML, and HTML, XPathScript outputs only plain text. This makes it a lot easier than XSLT for people coming from a Perl background to learn. However, XPathScript is not a W3C specification, despite being based on XPath, which is a W3C recommendation.

XPathScript follows the basic ASP syntax for introducing code and outputting code to the browser: use `<% %>` to introduce Perl code, and `<%= %>` to output a value.

The XPathScript API

Along with the code delimiters, XPathScript provides stylesheet developers with a full API for accessing and transforming the source XML file. This API can be used in conjunction with the delimiters listed above to provide a stylesheet language that is as powerful as XSLT, yet supports all the features of a full programming language such as Perl. (Other implementations, such as Python or Java, also are possible.)

Extracting values

A simple example to get us started is to use the API to bring in the title from a DocBook article. A DocBook article title looks like this:

```
<article>
 <artheader>
  <title>XPathScript - A Viable Alternative to XSLT?</title>
  ...
```

The XPath expression to retrieve the text in the <title> element is:

```
/article/artheader/title/text( )
```

Putting all this together to make this text into the HTML title, we get the following XPathScript stylesheet:

```
<html>
<head>
 <title><%= findvalue("/article/artheader/title") %></title>
</head>
<body>
  This was a DocBook Article.
  We're only extracting the title for now!
<p>
The title was: <%= findvalue("/article/artheader/title") %>
</body>
</html>
```

Again, we see the XPath syntax being used to find the nodes in the document, along with the function `findvalue( )`. Similarly, a list of nodes can be extracted (and thus looped over) using the `findnodes( )` function:

```
...
<%
for my $sect1 (findnodes("/article/sect1")) {
  print $sect1->findvalue("title"), "<br>\n";
  for my $sect2 ($sect1->findnodes("sect2")) {
    print " + ", $sect2->findvalue("title"), "<br>\n";
    for my $sect3 ($sect2->findnodes("sect3")) {
      print " + + ", $sect3->findvalue("title"), "<br>\n";
    }
  }
}
%>
...
```

Here we see how we can apply the `find*` functions to individual nodes as methods, which makes the node the context node to search from. That is, `$node->findnodes("title")` finds <title> child nodes of `$node`.

Declarative templates

We saw declarative templates earlier in this appendix, in the "Your First AxKit Page" section. The `$t` hash is the key to declarative templates. The `apply_templates( )` function iterates over the nodes of your XML file, applying the templates defined in the `$t` hash reference as it meets matching tags. This is the most important feature of

XpathScript, because it allows you to define the appearance of individual tags without having to do your own iteration logic. We call this *declarative templating*.

The keys of `$t` are the names of the elements, including namespace prefixes where appropriate. When `apply_templates()` is called, XPathScript tries to find a member of `$t` that matches the element name.

The following subkeys define the transformation:

pre
: Output to occur before the tag

post
: Output to occur after the tag

prechildren
: Output to occur before the children of this tag are written

postchildren
: Output to occur after the children of this tag are written

prechild
: Output to occur before every child element of this tag

postchild
: Output to occur after every child element of this tag

showtag
: Set to a false value (generally zero) to disable rendering of the tag itself

testcode
: Code to execute upon visiting this tag

More details about XPathScript can be found on the AxKit web site, at *http://axkit.org/*.

XSLT

One of the most important technologies to come out of the W3C is eXtensible Stylesheet Language Transformations (XSLT). XSLT provides a way to transform one type of XML document into another using a language written entirely in XML. XSLT works by allowing developers to create one or more template rules that are applied to the various elements in the source document to produce a second, transformed document.

While the basic concept behind XSLT is quite simple (apply these rules to the elements that match these conditions), the finer points of writing good XSLT stylesheets is a huge topic that we could never hope to cover here. We will instead provide a small example that illustrates the basic XSLT syntax.

First, though, we need to configure AxKit to transform XML documents using an XSLT processor. For this example, we will assume that you already have the

GNOME XSLT library (*libxml2* and *libxslt*, available at *http://xmlsoft.org/*) and its associated Perl modules (`XML::LibXML` and `XML::LibXSLT`) installed on your server.

Adding this line to your *httpd.conf* file tells AxKit to process all XML documents with a stylesheet processing instruction whose type is "text/xsl" with the `LibXSLT` language module:

```
AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT
```

Anatomy of an XSLT Stylesheet

All XSLT stylesheets contain the following:

- An XML declaration (optional)
- An `<xsl:stylesheet>` element as the document's root element
- Zero or more template rules

Consider the following bare-bones stylesheet:

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="/">
    <!-- the content for the output document contained here -->
  </xsl:template>
</xsl:stylesheet>
```

Note that the root template (defined by the `match="/"` attribute) will be called without regard for the contents of the XML document being processed. As such, this is the best place to put the top-level elements that we want to include in the output of each and every document being transformed with this stylesheet.

Template Rules and Recursion

Let's take our basic stylesheet and extend it to allow us to transform the DocBook XML document presented in Example E-8 into HTML.

Example E-8. camelhistory.xml

```
<?xml version="1.0"?>
<book>
<title>Camels: An Historical Perspective</title>
<chapter>
  <title>Chapter One</title>
  <para>
     It was a dark and <emphasis>stormy</emphasis> night...
  </para>
</chapter>
</book>
```

First we need to alter the root template of our stylesheet:

```
<xsl:template match="/">
  <html>
    <head><xsl:copy-of select="/book/title"/></head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
```

Here we have created the top-level structure of our output document and copied over the book's <title> element into the <head> element of our HTML page. The <xsl:apply-templates/> element tells the XSLT processor to pass on the entire contents of the current element (in this case the <book> element, since it is the root-level element in the source document) for further processing.

Now we need to create template rules for the other elements in the document:

```
<xsl:template match="chapter">
  <div class="chapter">
    <xsl:attribute name="id">chapter_id<xsl:number
    value="position( )" format="A"/></xsl:attribute>
    <xsl:apply-templates/>
  </div>
</xsl:template>
<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>
```

Here we see more examples of recursive processing. The <para> and <chapter> elements are transformed into and <p> elements, and the contents of those elements are passed along for further processing. Note also that the XPath expressions used within the template rules are evaluated in the context of the current element being processed. XSLT also maintains what is called the "current node list," which is the list of nodes being processed. In the example above, this is the list of all chapter elements. This is an example of XSLT using "least surprise".

While this sort of recursive processing is extremely powerful, it can also be quite a performance hit* and is necessary only for those cases where the current element contains other elements that need to be processed. If we know that a particular element will not contain any other elements, we need to return only that element's text value.

```
<xsl:template match="emphasis">
  <em><xsl:value-of select="."/></em>
</xsl:template>
<xsl:template match="chapter/title">
  <h2><xsl:value-of select="."/></h2>
```

* Although, since XSLT engines tend to be written in C, they are still very fast (often faster than most compiled Perl templating solutions).

```
    </xsl:template>
    <xsl:template match="book/title">
      <h1><xsl:value-of select="."/></h1>
    </xsl:template>
    </xsl:stylesheet>
```

Look closely at the last two template elements. Both match a `<title>` element, but one defines the rule for handling titles whose parent is a `book` element, while the other handles the `chapter` titles. In fact, any valid XPath expression, XSLT function call, or combination of the two can be used to define the match rule for a template element.

Finally, we need only save our stylesheet as *docbook-snippet.xsl*. Once our source document is associated with this stylesheet (see the section titled "Putting Everything Together" later in this appendix), we can point our browser to *camelhistory.xml*, and we'll see the output generated by the code in Example E-9.

Example E-9. camelhistory.html

```
<?xml version="1.0"?>
<html>
  <head>
    <title>Camels: An Historical Perspective</title>
  </head>
  <body>
    <h1>Camels: An Historical Perspective</h1>
    <div class="chapter" id="Chapter One">
      <h2>Chapter One</h2>
      <p>
         It was a dark and <em>stormy</em> night...
      </p>
    </div>
  </body>
</html>
```

The entire stylesheet is rendered in Example E-10.

Example E-10. docbook-snippet.xsl

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

 <xsl:template match="/">
  <html>
    <head><xsl:copy-of select="/book/title"/></head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
 </xsl:template>
```

Example E-10. docbook-snippet.xsl (continued)

```
 <xsl:template match="chapter">
  <div class="chapter">
    <xsl:attribute name="id">chapter_id<xsl:number
    value="position( )" format="A"/></xsl:attribute>
    <xsl:apply-templates/>
  </div>
 </xsl:template>
 <xsl:template match="para">
  <p><xsl:apply-templates/></p>
 </xsl:template>

 <xsl:template match="emphasis">
  <em><xsl:value-of select="."/></em>
 </xsl:template>
 <xsl:template match="chapter/title">
  <h2><xsl:value-of select="."/></h2>
 </xsl:template>
 <xsl:template match="book/title">
  <h1><xsl:value-of select="."/></h1>
 </xsl:template>
</xsl:stylesheet>
```

Learning More

We have only scratched the surface of how XSLT can be used to transform XML documents. For more information, see the following resources:

- The XSLT specification: *http://www.w3.org/TR/xslt/*
- Miloslav Nic's XSLT reference: *http://www.zvon.org/xxl/XSLTreference/Output/index.html*
- Jeni Tennison's XSLT FAQ: *http://www.jenitennison.com/xslt/index.html*

Putting Everything Together

The last key piece to AxKit is how everything is tied together. We have a clean separation of logic, presentation, and content, but we've only briefly introduced using processing instructions for setting up the way a file gets processed through the AxKit engine. A generally better and more scalable way to work is to use the AxKit configuration directives to specify how to process files through the system.

Before introducing the configuration directives in detail, it is worth looking at how the W3C sees the evolving web of new media types. The HTML 4.0 specification defines eight media types:

screen
: The default media type, for normal web browsers.

tty
: For *tty*-based devices (e.g., the Lynx web browser).

printer
: For printers requesting content directly (rather than for printable versions of a HTML page). Also for PDF or other paginated content.

handheld
: For handheld devices. You need to distinguish between WAP, cHTML, and other handheld formats using styles, because the W3C did not make this distinction when it defined the media types.

braille
: For braille interpreters.

tv
: For devices with a TV-based browser, such as Microsoft's WebTV and Sega's Dreamcast.

projection
: For projectors or presentations.

aural
: For devices that can convert the output to spoken words, such as VoiceXML.

AxKit allows you to plug in modules that can detect these different media types, so you can deliver the same content in different ways. For finer control, you can use named stylesheets. In named stylesheets, you might have a printable page output to the *screen* media type. Named stylesheets are seen on many magazine sites (e.g., *http://take23.org/*) for displaying multi-page articles.

For example, to map all files with the extension *.dkb* to a DocBook stylesheet, you would use the following directives:

```
<Files *.dkb>
AxAddProcessor text/xsl /stylesheet/docbook.xsl
</Files>
```

Now if you wanted to display those DocBook files on WebTV as well as ordinary web browsers, but you wanted to use a different stylesheet for WebTV, you would use:

```
<Files *.dkb>
  <AxMediaType tv>
    AxAddProcessor text/xsl /stylesheets/docbook_tv.xsl
  </AxMediaType>
  <AxMediaType screen>
    AxAddProcessor text/xsl /stylesheets/docbook_screen.xsl
  </AxMediaType>
</Files>
```

Now let's extend that to chained transformations. Let's say you want to build up a table of contents the same way in both views. One way you can do it is to modularize the stylesheet. However, it's also possible to chain transformations in AxKit, simply by defining more than one processor for a particular resource:

```
<Files *.dkb>
  AxAddProcessor text/xsl /stylesheets/docbook_toc.xsl
  <AxMediaType tv>
    AxAddProcessor text/xsl /stylesheets/docbook_tv.xsl
  </AxMediaType>
  <AxMediaType screen>
    AxAddProcessor text/xsl /stylesheets/docbook_screen.xsl
  </AxMediaType>
</Files>
```

Now the TV-based browsers will see the DocBook files transformed by *docbook_toc.xsl*, with the output of that transformation processed by *docbook_tv.xsl*.

This is exactly how we would build up an application using XSP:

```
<Files *.xsp>
  AxAddProcessor application/x-xsp .
  <AxMediaType tv>
    AxAddProcessor text/xsl /stylesheets/page2tv.xsl
  </AxMediaType>
  <AxMediaType screen>
    AxAddProcessor text/xsl /stylesheets/page2html.xsl
  </AxMediaType>
</Files>
```

This resolves the earlier issue we had where the XSP did not output HTML, but something entirely different. Now we can see why—because this way we can build dynamic web applications that work easily on different devices!

There are four other configuration directives similar to `AxAddProcessor`. They take two additional parameters: one that specifies a particular way to examine the file being processed and one to facilitate the match. The directives are:

`AxAddRootProcessor`
: Takes a root element name to match the first (root) element in the XML document. For example:

  ```
  AxAddRootProcessor text/xsl article.xsl article
  ```

  processes all XML files with a root element of `<article>` with the *article.xsl* stylesheet.

`AxAddDocTypeProcessor`
: Processes XML documents with the given XML public identifier.

`AxAddDTDProcessor`
: Processes all XML documents that use the DTD given as the third option.

`AxAddURIProcessor`
: Processes all resources at the matching URI (which is a Perl regex).

  This option was added for two reasons: because the `<LocationMatch>` directive is not allowed in an *.htaccess* file, and because the built-in Apache regular expressions are not powerful enough—for example, they cannot do negative matches.

Finally, the <AxStyleName> block allows you to specify named stylesheets. An example that implements printable/default views of a document might be:

```
<AxMediaType screen>
  <AxStyleName #default>
    AxAddProcessor text/xsl /styles/article_html.xsl
  </AxStyleName>
  <AxStyleName printable>
    AxAddProcessor text/xsl /styles/article_html_print.xsl
  </AxStyleName>
</AxMediaType>
```

By mixing the various embedded tags, it is possible to build up a very feature-rich site map of how your files get processed.

More Reasons to Use AxKit

Hopefully this will have whetted your appetite to play with AxKit. If you still need convincing, here are some extra things AxKit can do:

- AxKit can work with filter-aware modules and, instead of XSP, use other templating systems (such as Mason) to produce XML structures that will be styled on the fly after being passed to AxKit.
- XSLT, XSP, and XPathScript aren't the only possible processors. You can fairly easily create a new type of processor (such as a graph-outputting processor that would transform XML into charts, or rasterize some SVG).
- Apache configuration isn't the only way to control AxKit. You can create a `ConfigReader` that reads the configuration from another system, such as an XML file on disk.
- There are ways to choose stylesheets on the fly—for instance, to allow people to see the site with the design they prefer, based on cookies or a query string.
- AxKit has an intelligent and powerful caching system that can be controlled in various ways or replaced by a custom cache if needed.
- You don't need to fetch the initial content from the filesystem. The Provider interface allows you to return data from wherever Perl can get it (e.g., a content-management system).

For more information, help, support, and community chat, please visit the web site at *http://axkit.org/* and join in the discussions on the mailing lists, where you will find like minded people building a range of solutions.