



AxKit Essentials

Matt Sergeant

Slides at <http://axkit.org/docs/presentations/tpc2002/>



Overview

- You are going to learn:
 - ◆ What AxKit *is*
 - ◆ How to write dynamic web apps using XSP
 - ◆ What XSLT is, and how to use it
 - ◆ XPathScript
 - ◆ AxKit Extensions and Plugins

AxKit Pronunciation

- AxKit is not AIX Kit ;-)





AxKit 101

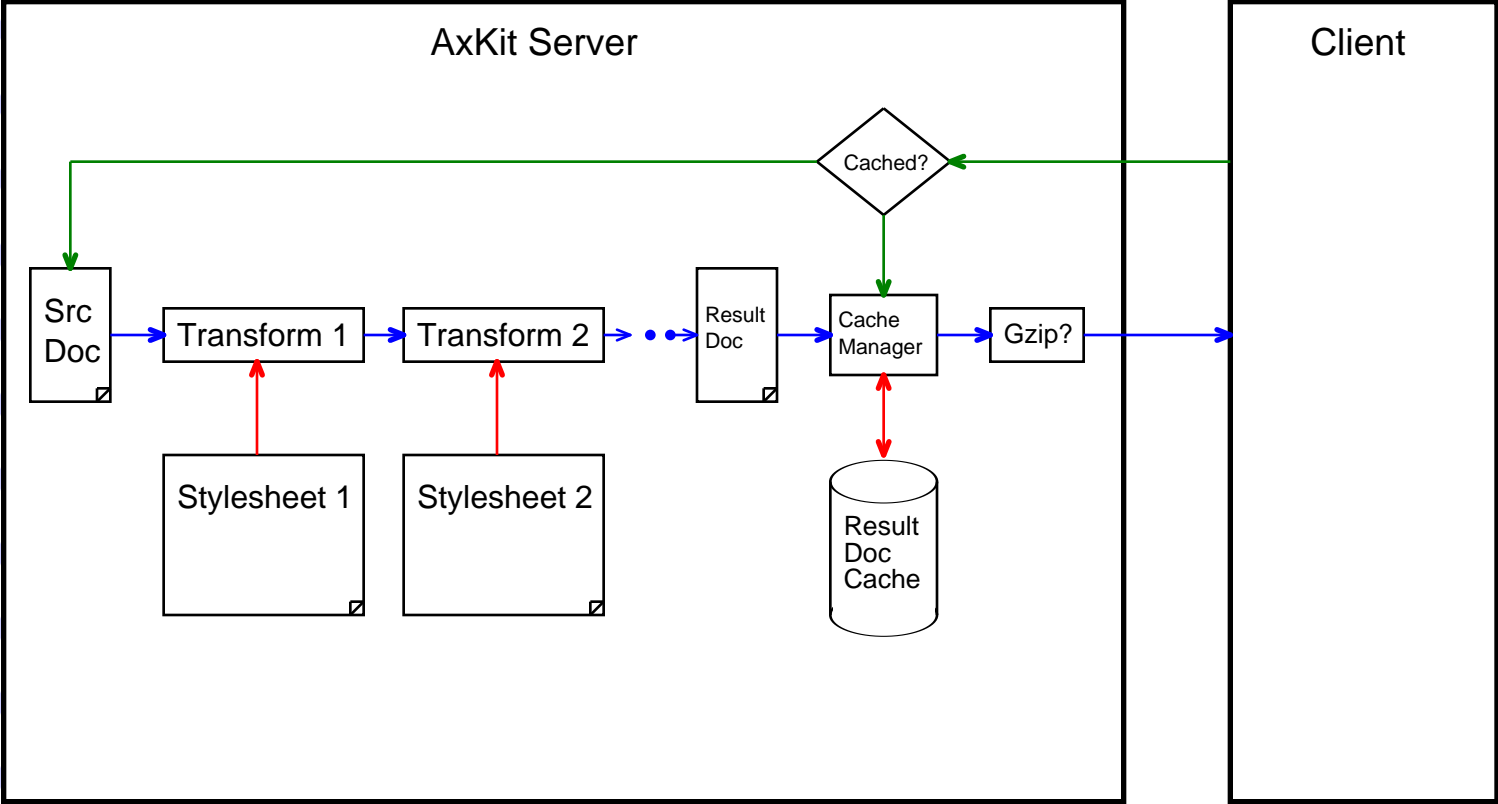
**Everything you always wanted to know about AxKit
but were afraid to ask**



30,000ft Overview

- AxKit turns Apache into an **XML Application Server**
- Meaning: Server does XML transformations
- Pipeline based - multiple transformations per request
- Transformations can be specialised based on client
- Cache management for performance

Diagrammatic Overview



What's in a transformation?

- * XSLT - XML Stylesheet Language - Transformation
- * XPathScript - Combines Perl, ASP, XPath, and some whizzy magic
- XSP - eXtensible Server Pages
- AxPoint
- XSL-FO - XML Stylesheet Language - Formatting Objects
- HTMLDoc - HTML to PDF converter
- **Anything else you like**

* means "requires stylesheet"

Caching

- Smart Caching module
- Caching *must* depend on:
 - ◆ XML files involved in the request (source, external entities, included files)
 - ◆ Stylesheets involved in the request
 - ◆ The client type - web browser, handheld device, etc
 - ◆ The *style* applied to this resource
 - ◆ Custom parameters
- Caching also works smartly with gzipping of results

The AxKit Dichotomy

- AxKit offers two key things:

- **Static Publishing Engine**

- XML files stored on the server
- Transformation happens at request time, and cached
- Delivery possible in different formats - HTML, PDF, XML
- Useful as part of a content management system...
- ... or just to make building your site easier

- **Dynamic XSP Applications**

- XSP file executed at runtime
- XSP generates XML
- Transformation engine then turns that XML into something usable by the client
- XSP truly allows separation of content from presentation and logic
 - ◆ Logic: Perl code
 - ◆ Content: XSP files, external XML files, XSP results
 - ◆ Presentation: XSLT or XPathScript



XSP

Dynamic Applications



XSP

- XSP is an Apache specification for a dynamic XML embedded scripting language
- Allows for run-time creation of XML tags and content
- Also allows embedded perl

Configuring the Server

```
# Load AxKit into the server
PerlModule AxKit

#####
## Debugging Options

## Maximum debugging
AxLogDeclines On
AxDebugLevel 10

## Specify the stylesheet to be executed on errors
AxAddStyleMap text/xml Apache::AxKit::Language::LibXSLT
AxErrorStylesheet text/xml /axkit/stylesheet/error.xml
#####
## XSP Options

# Enable AxKit for .xsp files
AddHandler axkit .xsp

## Load the module mapping:
AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP

## Enable XSP processing for files starting with
## <xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">
AxAddRootProcessor application/x-xsp NULL \
    {http://apache.org/xsp/core/v1}page
```

My First XSP Page

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">
  <xsp:structure>
    <xsp:import>Time::Piece</xsp:import>
  </xsp:structure>
  <page>
    <title>My First XSP Page</title>
    The time is now:
    <xsp:expr>
      localtime->strftime("%I:%M%P on %A %e %B, %Y")
    </xsp:expr>
  </page>
</xsp:page>
```

Results in:

```
<page>
  <title>My First XSP Page</title>
  The time is now:
  11:15am on Sunday 24 March, 2002
</page>
```

My First XSP Deconstruction

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">
  <xsp:structure>
    <xsp:import>Time::Piece</xsp:import>
  </xsp:structure>
  <page>
    <title>My First XSP Page</title>
    The time is now:
    <xsp:expr>
      localtime->strftime("%I:%M%P on %A %e %B, %Y")
    </xsp:expr>
  </page>
</xsp:page>
```

- Code executed when the page is compiled
 - ◇ Module importing (equivalent to "use Module")
 - ◇ Function definition
 - ◇ Page-global variables
- Code executed at request time
- First non-xsp:foo tag indicates start of output stream

Things to note

- The source code is XML
 - ◆ Means we sometimes have to be careful with < and & signs
- This guarantees that the output will be XML
- The output is not HTML
- So how do we output HTML?

XSP -> XML -> HTML

```
#####  
## XSP Options  
  
# Enable AxKit for .xsp files  
AddHandler axkit .xsp  
  
## Load the module mapping:  
AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP  
  
## Enable XSP processing for files starting with  
## <xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">  
AxAddRootProcessor application/x-xsp NULL \  
    {http://apache.org/xsp/core/v1}page  
  
## Turn XSP output into HTML using XSLT  
AxAddRootProcessor text/xsl /stylesheets/xsp_output_2html.xsl \  
    {http://apache.org/xsp/core/v1}page
```

Mix in the XSLT...

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="/page/title"/></title>
      </head>
      <body>
        <h1><xsl:value-of select="/page/title"/></h1>
        <xsl:apply-templates match="/page/*[name() != 'title']"/>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

But...

- I promised you separation of *logic* from *content*
 - ◆ (we can see we separate *content* from *presentation*)
- But we're mixing code in on our XSP page!

The time is now:

```
<xsp:expr>  
  localtime->strftime("%I:%M%P on %A %e %B, %Y")  
</xsp:expr>
```

- The solution is taglibs

```
The time is now: <datetime:now/>
```

Taglibs

- Taglibs separate the logic from the content of your XSP
- They hide the logic behind XML tags
- Taglibs are grouped into libraries providing the taglib
- Taglib libraries are bound to XML namespaces:
 - ◆ `<xsp:page xmlns:datetime="urn:datetime-taglib" ... >`
- Taglibs are what makes XSP great for web development
- Taglibs also are then answer to the question of "How do you cope with all the typing?"

Writing Taglibs - SimpleTaglib

- Method 1: SimpleTaglib

```
package AxKit::XSP::DateTime;
use strict;

use Apache::AxKit::Language::XSP::SimpleTaglib;
use vars qw($VERSION $NS);
$VERSION = 0.90;

# The namespace associated with this taglib.
$NS = 'http://axkit.org/xsp/demo/datetime';

sub now {
    return localtime->strftime("%I:%M%P on %A %e %B, %Y");
}

package AxKit::XSP::DateTime::Handlers;

sub now : expr {
    return 'AxKit::XSP::DateTime::now()';
}

1;
```

- Note: requires Perl 5.6.0

SimpleTaglib - Gory Details

- SimpTL encapsulates the messiness of XSP internals
- Manages the mapping of tag name to sub name (via `s/[^a-zA-Z0-9]/_/g`)
- Handles child tags as parameters, or context sensitive subs

◆ e.g.

```
<datetime:now>  
  <!-- function parameter -->  
  <datetime:format>%H:%M</datetime:format>  
</datetime:now>
```

◆ May be called as:

- `now(format => "%H:%M")`
- `now__format("%H:%M")`

- Mapping from tags to sub input and output is controlled via sub attributes (new in 5.6)

```
sub now : expr { ... }
```

- Return value of your sub is some code as a string
- Return value of *that code* is converted to XML

SimpleTaglib - Available Attributes

- Attributes that affect how the result is turned into XML (mutually exclusive)
 - ◆ `expr`
 - Creates text nodes or inline text from the result
 - ◆ `node(name)`
 - Makes this tag create an XML tag called "name".
 - ◆ `nodelist(name)`
 - Turns every item in the list of results from your sub into one of a list of tags called "name"
 - e.g. `sub get_user_list : nodelist(user) { ... }` generates:

```
<user>fred</user><user>bobby</user><user>john</user>
```
 - ◆ `exprOrNode(name,attrname,attrvalue)`
 - Creates a node if the value in the attribute *attrname* is the same as *attrvalue*
 - ◆ `exprOrNodelist(name,attrname,attrvalue)`
 - Same as above, but for a nodelist
 - ◆ `struct`
 - Creates a complex XML structure based on a complex data structure



SimpleTaglib - Available Attributes (cont.)

- Attributes that affect how attributes and sub-tags get passed as parameters

Writing Taglibs - TaglibHelper

- Method 2: TaglibHelper

```
package AxKit::XSP::DateTime;
use strict;

use Apache::AxKit::Language::XSP::TaglibHelper;
use Time::Piece;
use vars qw($VERSION $NS @ISA @EXPORT_TAGLIB);
$VERSION = 0.90;

# The namespace associated with this taglib.
$NS = 'http://axkit.org/xsp/demo/datetime';

# Using TaglibHelper:
@ISA = qw(Apache::AxKit::Language::XSP::TaglibHelper);

@EXPORT_TAGLIB = (
    'now()',
);

sub now () {
    return localtime->strftime("%I:%M%P on %A %e %B, %Y");
}

1;
```

TaglibHelper - Gory Details

- Works very similarly to SimpleTaglib
- Doesn't insert code into the XSP - tag subs are executed directly
 - ◆ Which means you get compile time checking, which is good
- Uses the `@EXPORT_TAGLIB` package variable to define tag => function mappings
- `@EXPORT_TAGLIB` also contains params to define how the output gets turned into XML

TaglibHelper - @EXPORT_TAGLIB Options

- `funcname(args)[:option[:option]...]`
- Tag => funcname mapping is the same as SimpleTaglib - `s/[^A-Za-z0-9]/_/g`
- Arguments:
 - ◆ Similar to Perl's function prototypes
 - ◆ `some_func($left,$right;$go)` - works with:


```
<ns:somefunc go="1">
  <ns:left>Leftie</ns:left>
  <ns:right>Rightie</ns:right>
</ns:somefunc>
```
 - ◆ Attributes or child tags - TLH does the right thing
 - ◆ `$argument` - string
 - ◆ `@argument` - list of arguments, passed as an arrayref
 - ◆ `*argument` - XML tree is turned into a complex data structure

TaglibHelper - @EXPORT_TAGLIB Options (cont.)

- funcname(args):options
 - ◆ listtag=users
 - Specifies a tag to use to wrap lists in
 - Defaults to <funcname>-list
 - ◆ itemtag=user
 - Specifies a tag to use to wrap list items in
 - Defaults to <funcname>-item
 - ◆ forcearray=1
 - Forces the return value to be treated as an array
 - Useful in case your function returns 1 value
 - ◆ conditional=1
 - Makes the tag equivalent to an if() statement:

```
<ns:if-in-doubt> <ns:jump-ship/> </ns:if-in-doubt>
```

TaglibHelper - @EXPORT_TAGLIB Options (cont.)

- ◆ `isreally=func_name`

- A built-in aliasing system

```
person($name):isreally=get_person
```

- ◆ `as_xml=1`

- Tells TaglibHelper that your function returns a string that is already XML

- ◆ `array_uses_hash=1`

- Provides an alternative structure => XML mapping

Writing Taglibs - Raw Handler

- Method 3: The hard way (almost never needed)

```
package AxKit::XSP::DateTime;
use strict;

use Time::Piece;
use vars qw($VERSION $NS @ISA);
$VERSION = 0.90;

# The namespace associated with this taglib.
$NS = 'http://axkit.org/xsp/demo/datetime';
@ISA = qw(Apache::AxKit::Language::XSP);

sub parse_start {
    my ($e, $tag, %attrs) = @_;

    if ($tag eq 'now') {
        $e->start_expr($tag);
        $e->append_to_script(
            'Time::Piece->new->strftime("%I:%M%P on %A %e %B, %Y")'
        );
        $e->end_expr();
    }

    return '';
}
1;
```



Writing Taglibs - ObjectTaglib

- Method 4: Object Taglibs
- The "new kid on the block"
- But it's by Simon Cozens, so you know it's good ;-)
- But few docs so we'll leave it off for now
- [http://search.cpan.org/search?
dist=Apache-AxKit-Language-XSP-ObjectTaglib](http://search.cpan.org/search?dist=Apache-AxKit-Language-XSP-ObjectTaglib)

Loading/Enabling a Taglib

- Load the taglib in httpd.conf/.htaccess

```
AxAddXSPTaglib AxKit::XSP::DateTime
```

- Declare the namespace in your XSP file

```
<xsp:page
  xmlns:xsp="http://apache.org/xsp/core/v1"
  xmlns:datetime="http://axkit.org/xsp/demo/datetime"
>
```

- ◇ **Note:** xmlns declaration **must** match the one in your module!

- Use the taglib

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"
  xmlns:datetime="http://axkit.org/xsp/demo/datetime">
  <xsp:structure>
    <xsp:import>Time::Piece</xsp:import>
  </xsp:structure>
  <page>
    <title>My First XSP Page</title>
    The time is now: <datetime:now/>
  </page>
</xsp:page>
```



CPAN Taglibs

- Param
- Cookie
- Sendmail
- Session
- Exception
- ESQL
- CharsetConv
- And many more...
- These are generally for "one-off" scripts

Example - Param

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"
  xmlns:param="http://axkit.org/NS/xsp/param/v1"
>
  <html>
    <head>
      <title>Param Taglib Example</title>
    </head>
    <body>
      <div>
        <form method="POST">
          <input type="text" name="myparam"/>
          <input type="submit" value="Submit"/>
        </form>
      </div>

      <xsp:logic>
        if (<param:myparam/>) {
          <div>
            You entered: <param:myparam/>
          </div>
        }
      </xsp:logic>
    </body>
  </html>
</xsp:page>
```

Example - ml-sub-request

```
<xsp:page xmlns:xsp="http://apache.org/xsp/core/v1"
  xmlns:param="http://axkit.org/NS/xsp/param/v1"
  xmlns:sendmail="http://axkit.org/NS/xsp/sendmail/v1"
>
  <html>
    <head>
      <title>Subscribed to axkit-users@axkit.org</title>
    </head>
    <body>
      <xsp:logic>
        if (<param:email/>) {
          eval {
            <sendmail:send-mail>
              <sendmail:from><param:email/></sendmail:from>
              <sendmail:to>axkit-users-subscribe@axkit.org</sendmail:to>
              <sendmail:body>Subscribe to axkit-users</sendmail:body>
            </sendmail:send-mail>
            <div>
              Subscription sent.
            </div>
          }
          if (!$?) {
            <div>
              Subscription failed: <xsp:expr>${?}</xsp:expr>
            </div>
          }
        }
        else {
          <div>
            Need to enter an email address.
          </div>
        }
      </xsp:logic>
    </body>
  </html>
</xsp:page>
```

Large-scale example

- No flow control. No logic. Simply pointers to what we need to do here.

```
<?xml version="1.0"?>
<xsp:page
  xmlns:xsp="http://www.apache.org/1999/XSP/Core"
  xmlns:f="res:perl/VR/Kernel/FirstTime"
  xmlns:p="res:perl/VR/Kernel/Package"
  language="perl"
>
<response>
  <f:bootstrap-database/>
  <!--
    We install the kernel's package.xml file now because it has
    to run during the request's cleanup process. So we need to get
    everything prepared in time for the actual parameter reading.
  -->
  <p:save-temp-package-from-temp-file install-from-kernel="1"/>
</response>
</xsp:page>
```

- *"This file generates not one, but two databases on two different servers, then bootstraps a few core tables into those databases, then loads up an XML-based package description file into the database. How does it do all this? Who cares! There could be 200 or more lines of code hidden in those calls, with complicated logic and whatnot. This is the beauty of XSP and taglibs: The taglibs do all the freaky complex stuff, with no knowledge that they're in a web environment, and the XSP files just describe what an individual page should do."*



PerForm Taglib

- Callback based web forms
- Callbacks for form start/end, data load, validation, submit
- Results in an abstract form in XML
- Includes example XSLT to render to HTML

PerForm Example

```
<xsp:page
  xmlns:xsp="http://apache.org/xsp/core/v1"
  xmlns:f="http://axkit.org/NS/xsp/perform/v1">
<xsp:logic>
  use MyUserDB;
  sub validate_firstname {
    my ($ctx, $value) = @_;
    $value =~ s/^\s*//; $value =~ s/\s*$//;
    die "No value" unless $value;
    die "Invalid firstname - non word character not allowed"
      if $value =~ /\W/;
  }

  sub validate_lastname {
    my ($ctx, $value) = @_;
    validate_firstname(@_);
    if (MyUserDB->user_exists($ctx->{Form}{firstname}, $value)) {
      die "User already exists";
    }
  }

  sub submit_save {
    my ($ctx) = @_;
    MyUserDB->add_user($ctx->{Form}{firstname}, $ctx->{Form}{lastname});
  }
</xsp:logic>
<page>
  <f:form name="add_user">
    First name: <f:textfield name="firstname" width="30" maxlength="50"/>
    <br />
    Last name: <f:textfield name="lastname" width="30" maxlength="50"/>
    <br />
    <f:submit name="save" value="Save" goto="users.xsp" />
    <f:cancel name="cancel" value="Cancel" goto="home.html" />
  </f:form>
</page>
</xsp:page>
```

PerForm Details

- Form posts back to itself (otherwise the subs wouldn't be there)
- PerForm logic figures out the functions to call at compile time
- Calls `load_<name>` at load time, and `validate_<name>` at save time
- Finally calls `submit_<name>` after validating
- If any `validate_<name>` functions die("some text"), the "some text" gets added as an error in the output XML
- XSLT styles the resulting form into however you want it to look
 - ◆ You can put errors next to their widgets, or all errors at the top

XSP as an MVC Framework

- MVC is:
 - ◆ Model - The data model - database, abstraction layer, classes, etc
 - ◆ View - Our view onto the data - how we present it to the user
 - ◆ Controller - Code controlling the interaction between the view and the model
- With XSP, MVC is:
 - ◆ Model - the XSP taglib module
 - ◆ View - The XSLT stylesheet applied to the output of our XSP page
 - ◆ Controller - the combination of the XSP and the .htaccess/httpd.conf files

XSP+MVC Example

- AxKit Wiki
- XSP Page:
 - ◆ Decodes the URI into db, page, etc.
 - ◆ Decides what the "command" is depending on the querystring
 - ◆ Dispatches to the appropriate taglib function
- Taglib Module:
 - ◆ Knows **nothing** about the web
 - ◆ Simply returns XML representation of a Wiki page, either raw (for edit) or converted to XML
- XSLT:
 - ◆ Turns <edit> chunk into HTML <textarea>.
 - ◆ Turns POD XML representation (generated by Pod::SAX) into HTML
 - ◆ Turns other formats (Wiki2XML, sdocbook) into HTML
- Available from CPAN

Wiki Code

```
<?xml version="1.0"?>
<xspwiki xmlns:wiki="http://axkit.org/NS/xsp/wiki/1"
  xmlns:xsp="http://apache.org/xsp/core/v1">
  <xsp:logic>
    my ($path_info,$dbroot,$db,$page) = ($r->path_info,'/tmp','AxKit','DefaultPage');
    if ($path_info) {
      ($db, $page) = AxKit::XSP::Wiki::path_info_split($path_info);
    }
    else {
      $r->header_out(Location => "view/$db/$page");
      return 302;
    }

    my $action = $cgi->param('action') || 'view';
    if ($action eq 'save') {
      AxKit::XSP::Wiki::save_page( $dbroot, $db, $page,
        $cgi->param('text'),
        $cgi->param('texttype'),
      );
      $r->header_out(Location => ".$page");
      return 302;
    }

    my $title = join(' ', split(/(?=[A-Z])/, $page));

    <xsp:content>
      <title><xsp:expr>$title</xsp:expr></title>
      <page><xsp:expr>$page</xsp:expr></page>
      <wiki:display-page>
        <wiki:dbpath><xsp:expr>$dbroot</xsp:expr></wiki:dbpath>
        <wiki:db><xsp:expr>$db</xsp:expr></wiki:db>
        <wiki:page><xsp:expr>$page</xsp:expr></wiki:page>
        <wiki:action><xsp:expr>$action</xsp:expr></wiki:action>
      </wiki:display-page>
    </xsp:content>

  </xsp:logic>
</xspwiki>
```

Wiki Taglib

- 4 main functions
 - ◆ display_page
 - ◆ view_page
 - ◆ edit_page
 - ◆ save_page
- display_page calls view_page or edit_page depending on \$action
- Both of those return XML
- The taglib puts the XML into the result document

Why is this a better Wiki?

- Most wiki's have the presentation code directly in their WikiText parsers
- They rely on CSS to change the style
- And "sandwich" techniques for more complex layouts
- The AxKit wiki is fully styled by XSLT
- Every aspect of its appearance can be changed without touching the perl code



XSP Internals

- What happens when we compile an XSP page?
- What happens when we run an XSP page?
- How does XSP work?

XSP Internals - Compilation

- First processes XInclude tags
- Then pass through a Namespace Dispatch SAX parser
- Each module registers interest in 1 or more namespaces
- Each tag (and child text) in that namespace is passed to the relevant SAX module
- The core tags (e.g. <xsp:expr>) are passed to the AxKit::XSP::Core module
- Each callback function returns a string to append to the generated code
- Non-taglib tags get turned into code to re-generate those tags

XSP Internals - Example Code

```
<xsp:page
  xmlns:xsp="...">

  <xsp:structure>
    <xsp:import>
      Time::Piece
    </xsp:import>
  </xsp:structure>

  <page>
    The time is now:
    <xsp:expr>
      localtime->strftime
    </xsp:expr>
  </page>
</xsp:page>
```

```
package _some_obfuscated_package;
# import XSP modules...
# ...
use Time::Piece;

sub handler {
  my ($r, $cgi, $document) = @_;
  my $parent;
  {
    my $elem = $document->createElement(q(page));
    $document->setDocumentElement($elem);
    $parent = $elem;
  }
  my $node = $document->createTextNode(
    q(The time is now: )
  );
  $parent->appendChild($node);
}
{
  my $node = $document->createTextNode(''.
    do { localtime->strftime }
  );
  $parent->appendChild($node);
}
return OK;
}
```

- Important: prior to <page> is package global - compiled once



XSP Conclusions

- XSP Taglibs allow true separation of content from logic and presentation
- Taglibs are **easy!**
- XSP can also be for throwaway scripts
- XSP -> XSLT (or XPathScript) means we can redirect our output to different formats



Static Publishing

Server side transformations

Configuring the Server

```
# Load AxKit into the server
PerlModule AxKit

## Load the module mapping:
AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT

#####
## Debugging Options

## Maximum debugging
AxLogDeclines On
AxDebugLevel 10

## Specify the stylesheet to be executed on errors
AxErrorStylesheet text/xsl /axkit/stylesheets/error.xsl
#####
## XSLT Options

# Enable AxKit for .xml files
AddHandler axkit .xml

## Enable XSLT processing for files starting with
## <article> with stylesheet /stylesheets/article_2html.xsl
AxAddRootProcessor text/xsl /stylesheets/article_2html.xsl article

#####
## Caching Options
AxCacheDir /tmp/axkit-cache
```



XSLT Introduction

- XSLT provides XML -> XML transformation based on a stylesheet
- This is **different** to Perl template tools
 - ◆ And by that we mean Template Toolkit, Text::Template, HTML::Template, etc
- - those modules provide *data structure* to *text* transformations
- XSLT is a key to the pipeline approach
- Like XSP, XSLT uses XML Namespaces to define "functions"
- XSLT is a recursive declarative functional language!!!

XSLT Example

- First, our source 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>
```

XSLT Example

- Next, our XSLT:

```
<?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>
  <xsl:template match="chapter">
    <div class="chapter">
      <xsl:attribute name="id"><xsl:value-of
        select="title"/></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>
```

XSLT Example

- Breakdown: Root Template

```
<?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>
  <xsl:template match="chapter">
    <div class="chapter">
      <xsl:attribute name="id"><xsl:value-of
        select="title"/></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>
```

```
<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>
```

XSLT Example

- Breakdown: Recurse to <chapter>

```
<?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>
  <xsl:template match="chapter">
    <div class="chapter">
      <xsl:attribute name="id"><xsl:value-of
        select="title"/></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>
```

```
<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>
```

XSLT Example

- Breakdown: Remaining Templates

```
<?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>
  <xsl:template match="chapter">
    <div class="chapter">
      <xsl:attribute name="id"><xsl:value-of
        select="title"/></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>
```

```
<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>
```

Results

```
<?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>
```



Things to note

- Result is cached
- Cache invalidated if XSLT or XML changes
- There may be further XSLT transformations made after this one
 - ◆ For example: add a table of contents
- Could use XPathScript instead of XSLT
 - ◆ (but that's another talk :-)

XPathScript

- An alternate XML stylesheet language, invented by me!
- Combination of:
 - ◆ Perl
 - ◆ ASP <% %> delimiters
 - ◆ XML::XPath for locating nodes in the source XML
 - ◆ Declarative (Rules based) processing
- Not a functional language
- Side effects allowed
- Full access to Apache request API
- Stylesheet is compiled into a perl function (like XSP, Apache::ASP, etc)

XPathScript example (xml)

```
<employees>
  <employee>
    <name>
      <firstname>Roger</firstname>
      <lastname>Rabbit</lastname>
    </name>
    <department>Humour</department>
  </employee>
  <employee>
    <name>
      <firstname>Jessica</firstname>
      <lastname>Rabbit</lastname>
    </name>
    <department>Modelling</department>
  </employee>
</employees>
```

XPathScript example (stylesheet)

```
<html>
<head><title>Employee List</title></head>
<body>
<h1>Employees at Acme Corp.</h1>

<% foreach my $employee (findnodes('/employees/employee')) { %>

<b><%= $employee->findvalue("name/lastname") %>,
<%= $employee->findvalue("name/firstname") %></b>
works in the <%= $employee->findvalue('department') %>
department.

<hr>

<% } %>

</body>
</html>
```

XPathScript example (results)

```
<html>
<head><title>Employee List</title></head>
<body>
<h1>Employees at Acme Corp.</h1>

<b>Rabbit, Roger</b> works in the Humour department
<hr>
<b>Rabbit, Jessica</b> works in the Modelling department
<hr>
</body>
</html>
```



XPathScript - Processing Documents

- Transforming Data is very different to transforming Documents
- Documents have mixed content
- Needs rules based processing
- XPathScript implements a feature rich declarative processing system

XPathScript - Declarative Templates

- The \$t hash reference

```
$t->{'para'}{pre} = '<p>';  
$t->{'para'}{post} = '</p>';
```

- Matches element names (unlike XSLT)
- XPathScript applies the rules as it traverses the document tree

XPathScript - \$t subkeys

- Sub-keys of \$t specify what to do
- Keys whose value is something to add to the output:
 - ◆ pre
 - ◆ post
 - ◆ prechildren
 - ◆ postchildren
 - ◆ prechild
 - ◆ postchild
- Other keys:
 - ◆ showtag
 - ◆ testcode

XPathScript - testcode

- testcode value is a code ref
- Executed every time an element with that name is visited
- Gives access to XML::XPath API on that node, and a local copy of \$t

```
$t->{'a'}{testcode} = sub {  
  my ($node, $t2) = @_;  
  if ($node->findvalue('@name')) {  
    # process as anchor  
  }  
  else {  
    # process as link  
  }  
  return DO_SELF_AND_KIDS;  
};
```



Mapping XML to Processing Pipelines



Two Ways

- `<?xml-stylesheet?>`
 - ◆ W3C Standard
 - ◆ <http://www.w3.org/TR/xml-stylesheet>
- Configuration Directives
 - ◆ AxAddProcessor and co.

Using `<?xml-stylesheet?>`

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="/stylesheets/render2html.xsl"?>
<article>
  <title>How to skin a cat</title>
  ...
```

- Causes the file to be processed by XSLT at </stylesheets/render2html.xsl> relative to web root

Two Stylesheets

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
    href="/stylesheets/create_toc.xsl"?>
<?xml-stylesheet type="text/xsl"
    href="/stylesheets/render2html.xsl"?>
<article>
  <title>How to skin a cat</title>
  ...
```

- Sends first through XSLT `create_toc.xsl`, then through `render2html.xsl`
- Note any `<?xml-stylesheet?>` generated by `create_toc.xsl` is ignored

Title attribute

```
<?xml version="1.0"?>
<?xml-stylesheet href="/stylesheets/docbook_screen.xps"
  type="application/x-xpathsript" title="default"?>
<?xml-stylesheet href="/stylesheets/docbook_print.xps"
  type="application/x-xpathsript" title="print"
  alternate="yes"?>
<article>

<arthead>
  <title>Getting Started with AxKit</title>
...
```

- By default, goes through /stylesheets/docbook_screen.xps
- If user chooses the "print" style somehow, goes through /stylesheets/docbook_print.xps
- e.g. http://server/getting_started.xml?style=print

Media Types

```
<?xml version="1.0"?>
<?xml-stylesheet href="/stylesheets/docbook_screen.xps"
  type="application/x-xpathsript" media="screen"?>
<?xml-stylesheet href="/stylesheets/docbook_print.xps"
  type="application/x-xpathsript" media="printer"?>
<article>

<artheader>
  <title>Getting Started with AxKit</title>
  ...
```

- Media type chosen by a plugin

AxAddProvider and co.

```
AxAddProvider text/xsl /stylesheets/render2html.xsl
```

- Apache config directives
- Go in either httpd.conf or .htaccess

```
<Files *.xml>  
  AxAddProcessor text/xsl /stylesheets/render2html.xsl  
</Files>
```

- AxAdd*Provider give more power

AxAddDocTypeProcessor & AxAddDTDProcessor

- AxAddDocTypeProcessor:

```
AxAddDocTypeProcessor text/xsl /stylesheets/docbook.xsl \  
    "-//OASIS//DTD DocBook XML V4.1.2//EN"
```

- Uses the DOCTYPE PUBLIC identifier:

```
<?xml version="1.0">  
<!DOCTYPE article PUBLIC  
    "-//OASIS//DTD DocBook XML V4.1.2//EN">  
<article>  
    ...
```

- AxAddDTDProcessor:

```
AxAddDTDProcessor text/xsl /stylesheets/rss2html.xsl \  
    "/dtds/rss.dtd"
```

AxAddRootProcessor

```
AxAddRootProcessor text/xsl /stylesheets/book2html.xsl book
```

- Defines pipeline based on root element:

```
<?xml version="1.0"?>
<book>

  <title>Making Money with Open Source</title>
  ...
</book>
```

- Also works with XML Namespaces:

```
# Map to <xsp:page xmlns:xsp="http://apache.org/xsp/core/v1">
AxAddRootProcessor application/x-xsp NULL \
  {http://apache.org/xsp/core/v1"}page
```

AxAddURIProcessor

- Gives perl5 regexp power to URI mapping

```
AxAddURIProcessor text/xsl book.xsl "book.*?\.(xml|dxb)$"
```

Building Pipelines

- Plain pipelines

```
AxAddProcessor application/x-xsp NULL
AxAddProcessor text/xsl /stylesheets/results2html.xsl
```

- Pipelines with different styles:

```
<AxStyleName "#default">
  AxAddProcessor application/x-xpathscript \
    /stylesheets/docbook_screen.xps
</AxStyleName>
<AxStyleName "print">
  AxAddProcessor application/x-xpathscript \
    /stylesheets/docbook_print.xps
</AxStyleName>
```





Debugging AxKit



AxKit Debugging Tools

- The AxKit Error Log
- AxTraceIntermediate
- XSLT Interactive Debugging

Error Log

AxDebugLevel 10
AxLogDeclines On

- Debug Level goes from 0..10
- AxKit can decline to process the resource if it finds an empty pipeline

AxTraceIntermediate

- Often we need to see what's happening mid-pipeline

```
AxTraceIntermediate /tmp/axkit-trace
```

- Puts files in that directory representing each stage of the transformation

```
# ll /tmp/axkit-trace/  
total 12k  
-rw-r--r--      1 nobody   nobody           1.6k Jul 15 21:04 | wiki | v  
-rw-r--r--      1 nobody   nobody           1.6k Jul 15 21:04 | wiki | v  
-rw-r--r--      1 nobody   nobody           3.6k Jul 15 21:04 | wiki | v
```

XSLT Debugging

- Many tools available
- Komodo - Allows visual XSLT debugging
- Visual XSLT - Same, but from within Visual Studio
- xsldbg - free software, console debugger





Providers

Overriding Where AxKit Gets Data From



Providers? What's that?

- Providers are a filesystem/url abstraction
- They allow us to generate XML from non-XML sources
- Or bring in data from non-file sources

AxKit Renders POD

```
<FilesMatch "\.pod">  
  AddHandler axkit .pod  
  AxContentProvider Apache::AxKit::Provider::PodSAX  
  AxAddProcessor text/xsl /stylesheets/pod2html.xsl  
</FilesMatch>
```

- Now POD is rendered directly to HTML
- POD -> Pod::SAX -> XML -> XSLT -> HTML

AxKit as an Apache::Filter

```
PerlModule Apache::Filter
<FilesMatch "\.asp">
  SetHandler perl-script
  PerlHandler Apache::ASP AxKit
  PerlSetVar Filter On
  AxContentProvider Apache::AxKit::Provider::Filter
</FilesMatch>
```

- Now write an ASP that outputs XML, and AxKit will transform it



OpenOffice Delivery

- Apache::AxKit::Provider::OpenOffice
- Commercial (\$100) product - delivers OpenOffice Writer as XML
- Includes (large) XSLT stylesheets to convert directly to HTML

Providers Dissected

- `init()` - called for you from `new()` - used to convert relative URIs into files, do `stat()`, etc
- `get_strref()` and `get_fh()`
 - ◆ Throw an `Apache::AxKit::Exception::IO` if you can't `get_fh()`
- `process()` - returns boolean whether we should process this resource
- `has_changed($time)` - returns true if this resource has changed since `$time`
- `key()` - return a unique key for this resource (usually the filename)
- `exists()` - returns true if this resource really exists
- `mtime()` - returns the resource's last modification time

Example Provider - PodSAX

```
package Apache::AxKit::Provider::PodSAX;
use strict;
use vars qw/@ISA $VERSION/;
@ISA = ('Apache::AxKit::Provider::File');
$VERSION = '1.00';

use Apache::AxKit::Provider::File;
use XML::SAX::Writer;
use Pod::SAX;

sub get_strref {
    my $self = shift;
    if ($self->_is_dir()) {
        throw Apache::AxKit::Exception::IO(
            -text => "$self->{file} is a directory - please overload File provider and use
        )
    }

    my $outie;
    my $w = XML::SAX::Writer->new( Output => \$outie );
    my $generator = Pod::SAX->new( Handler => $w );

    eval {
        $generator->parse_uri( $self->{file} );
    };

    if (my $error = $@) {
        throw Apache::AxKit::Exception::IO(
            -text => "PodSAX Generator Error: $error");
    }
    #warn "OUTIE $outie \n";
    return \$outie
}

1;
```

● Providers can be complex, but they can be very simple



AxKit Plugins

Modifying AxKit in scary but good ways

StyleChooser Plugins

- Choose the style (which specifies the processing chain)
- Choose by:
- Cookie
- File Suffix
- Path Info
- Query String
- User Agent

```
AxAddPlugin Apache::AxKit::StyleChooser::Cookie
```



Cookie Stylechooser

- Picks up the style from the "axkit_preferred_style" cookie
- Use XSP or some other dynamic engine to create the cookie
- Example at <http://hampton.ws/>

File Suffix Stylechooser

- Allows you to request /foo.xml.html, or /foo.xml.pdf
- Needs to be a PerlTypeHandler - prior to URI => filename translation is performed

```
PerlTypeHandler Apache::AxKit::StyleChooser::FileSuffix
```

- This is useful because some browsers rely on the suffix (esp. for PDF)

User Agent Stylechooser

- Allows you to customise the site to different browsers
- Map multiple browser to the same style

```
AxAddPlugin Apache::AxKit::StyleChooser::UserAgent
```

```
PerlSetVar AxUAStyleMap "lynx      => Lynx,\  
                          explorer => MSIE,\  
                          opera    => Opera,\  
                          netscape => Mozilla"
```

Augmenting the Cache

- Cache index is based on:
 - ◆ "filename" of the resource
 - ◆ any path_info on the URI
 - ◆ Preferred style
 - ◆ Requesting media type (e.g. web browser, PDA, etc)
 - ◆ Any "extras"
- Extras stored in `$r->notes('axkit_cache_extra')`
- Example: Paginated document based on querystring
 - ◆ `http://server/docs/document.xml?section=2`
 - ◆ QueryString not used in cache by default, so same page will always be delivered
 - ◆ Need a plugin to augment the cache with the values in the querystring

Apache::AxKit::Plugin::QueryStringCache

```
package Apache::AxKit::Plugin::QueryStringCache;
use strict;

use Apache::Constants qw(OK);

sub handler {
    my $r = shift;
    $r->notes('axkit_cache_extra', $r->notes('axkit_cache_extra'));
    return OK;
}

1;
```

Passthru

- Sometimes we want a way to get at the untransformed XML
- e.g. these presentations - people want the raw .axp file
- Install the Passthru plugin:

```
AxAddPlugin Apache::AxKit::Plugin::Passthru
```

- Now *<http://server/docs/axkit.axp?passthru=1>* gets you the source

Further Resources

- The AxKit Web site: <http://axkit.org/>
- W3C XSLT Spec: <http://www.w3.org/TR/xslt>
- Zvon - XML Technologies Tutorials: <http://www.zvon.org/>
- The XSLT FAQ: <http://www.jenitennison.com/xslt/index.html>
- XML Projects @ Apache.org: <http://xml.apache.org/>
- libxml2 and libxslt: <http://www.xmlsoft.org/>
- Sablotron XSLT: <http://www.gingerall.com/>
- AxKit XSP Taglibs:
<http://search.cpan.org/search?mode=module;query=AxKit%3A%3AXSP>