

# XML with Perl

Doug Treder  
UW Extension

# What is XML?

- Hype!
- Web services, e-commerce, XML, Schemas, XSL, DTDs oh my!
-

# Background--SGML

- Standard Generalized Markup Language
- SGML is an international standard for the definition of device-independent, system-independent methods of representing texts in electronic form.

ISO 8879: Information processing---Text and office systems---Standard Generalized Markup Language (SGML), ([Geneva]: ISO, 1986).

# SGML

- SGML has a record of successful application in large, complex, sophisticated, publishing applications.
- However, it had never (as of mid-1996) really taken off on the World-Wide Web, despite the fact that HTML was advertised as being an application of SGML.

# Background--HTML

- HTML is a simple subset of SGML
- HTML uses a fixed and finite set of tags, elements, and attributes that allow it to communicate to a user's browser how that browser should display the document.
- Proven technology, used everywhere on the web.
- What if, however, the current version of HTML doesn't cover the kind of data you want to markup?

# Enter XML

- XML is an attempt to package up the important virtues and most-used features of SGML in a compact, easily-implemented package that is optimized for delivery on the WWW.

# What is XML

- XML is a metalanguage - create and format your own document markups.
  - Define your own tags!
- XML isn't exciting on its own.
  - It doesn't actually do anything
  - It's just a way to wrap text-based data.
  - It's excited a lot of people because it seems great for the internet.

# XML vs HTML

- They both are text-based.
- They both consist of tags, elements, and attributes.
- Unlike HTML, however, XML allows users to structure and define the information in their documents.
- Users can create their own tags

# Why not XML?

- some people have gone too far:
  - XML is verbose to read and write
  - Difficult to author manually (type)
  - it takes more space than other formats like relational databases

# Why not XML?

- It doesn't have any built-in self-referencing
  - so it's not great for storing data and looking it up easily
- plenty of people have created whole groups of products to help with these difficulties
- Use the right tool for the job

# Tags, elements, attributes

- In this example, tags are in <>, elements are "table", "tr", and "td", attributes are "border", and "width"

```
<table border="0">  
  <tr>  
    <td width="50%">Here is the first group of text</td>  
    <td width="50%">Here is the second group of text</td>  
  </tr>  
</table>
```

# XML vs HTML

- In XML, YOU define the tags, elements and attributes

```
<?xml version="1.0"?>
<school name="UW">
  <teachers>
    <teacher id="1">Joel Grow</teacher>
  </teachers>
  <students>
    <student id="1">Fred Smith</student>
    <student id="2">Sarah Jones</student>
  </students>
</school>
```

# XML display flexibility

- After creating an XML document, you can display its contents in your format of choice.
- The same XML document could easily be displayed as HTML, a Microsoft Word document, an Adobe .pdf file, or even as text in the body of an e-mail message.

# XML display flexibility

- XML should be used only to mark up the semantic meaning of the data.
- Not how to display it.
- As long as the XML document is well formed (meaning that it follows the appropriate XML format and syntax), you can choose your method of preference (or necessity) for displaying its content.

# XML Dissection

- The XML header. Tells the document's user that this is an XML document – using version 1.0 of the XML specification in this case.

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

# XML Tags

- Just like in HTML, you use greater than (“>”) and less than (“<”) signs called tags to indicate the opening and closing of an element.



```
<teachers>  
  <teacher id="1">Joel Grow</teacher>  
</teachers>
```

# XML Elements

- The basic building blocks of XML.
- They may contain text, comments, or other elements, and consist of a start tag and an end tag.
- Typically, XML elements are like nouns in the real world. They represent people, places, or things.

# XML Elements

- Note that in XML, every opening element ( `<school>` ) must also contain a closing element ( `</school>` ).
- The closing element consists of the name of the opening element, prefixed with a slash (“/”).
- XML is case-sensitive. While

```
<school></school>  <!-- is well-formed -->  
<SCH00L></school>  <!-- is not -->  
<School></sCH00L>  <!-- is not. -->
```

# XML Elements

- If the element does not contain text or other elements, you may abbreviate the closing tag by simply adding a slash (“/”) before the closing bracket in your element:

```
<school></school>
```

can be abbreviated as

```
<school />
```

# XML Elements

- In addition to the rules defining opening and closing tags, it is important to note that in order to create a well-formed XML document, you must properly nest all elements:

```
<!-- wrong -->  
<teachers>  
  <teacher id="1">Joel Grow</teacher>  
</teachers>  
  <teacher id="2">Bob Abarbanel</teacher>
```

# XML Attributes

- Modify the elements.
- Attribute values must be contained in quotes

```
<school name="UW">  
  <teachers>  
    <teacher id="1">Joel Grow</teacher>  
    <teacher id="2">Bob Abarbanel</teacher>  
  </teachers>  
</school>
```

# XHTML

- The W3C group has rewritten the HTML 4 language in XML 1.0
- All elements must be properly nested
- The document must be well-formed
- All tagnames and attributes in lowercase
- All elements must be closed
- Attributes must be quoted and have values

# XHTML is one step

- XML was designed to describe data and to focus on what data is.
- HTML has been adopted to display data and to focus on how data looks.
- HTML has become about displaying information, XML is about describing information.

# XHTML is one step

- XML is not a replacement for HTML.
- HTML should become strictly semantic markup
- Display is handled via Cascading Stylesheets (or other transformation)
- Consider `<font />` `<bold />` `<p>` and `<br>` tags used for spacing, etc.

# Attributes vs elements

- Using an attribute vs element for paytype:

```
<employee paytype="hourly">  
  <firstname>Sarah</firstname>  
  <lastname>Jones</lastname>  
</employee>
```

```
<employee>  
  <paytype>hourly</paytype>  
  <firstname>Sarah</firstname>  
  <lastname>Jones</lastname>  
</employee>
```

# Attributes vs elements

- No fixed rules about when to use attributes vs elements.
- My experience is to try to use elements whenever possible
- Some problems with using attributes:
  - attributes can not contain multiple values (elements can)

# Attributes vs elements

- More problems using attributes:
- attributes are not expandable (for future changes)
- attributes can not describe structures (like child elements can)
- attributes are more difficult to manipulate by program code
- attribute values are not easy to test against a DTD

# Attributes to the extreme

```
<?xml version="1.0"?>  
<employee firstname="Sarah"  
  lastname="Jones"  
  paytype="hourly"  
  email="sjones"  
  city="Tacoma"  
  state="WA"  
  zip="98765">  
</employee>  
  
<!-- not good -->
```

# Attributes to the extreme

```
<?xml version="1.0"?>
<employee>
  <name>
    <first>Sarah</first>
    <last>Jones </last>
  </name>
  <paytype>hourly</paytype>
  <email>sjones</email>
  <city>Tacoma</city>
  <state>WA</state>
  <zip>98765" </zip>
</employee>

<!-- better -->
```

# XML browsers

- Can we just put up XML web pages?
- Some browsers let you view XML
- Some let you apply a custom stylesheet
- Some even contain an XSL transform engine!

Try this page:

[http://www.xmlfiles.com/examples/plant\\_catalog.xml](http://www.xmlfiles.com/examples/plant_catalog.xml)

# Validation

- XML that obeys all the rules I've described so far is called "well-formed"
- XML that isn't well-formed is rejected completely by many parsers
  - so you get nothing in your program

# Validation

- XML that conforms to a Document Type Definition or XML Schema is called "valid".
- There are several open source validators that can tell you if the document "validates"

# DTD

- Document Type Definition
  - (NOT required to use XML)
- The purpose of a DTD is to define the legal building blocks of an XML document.
- It defines the document structure with a list of legal elements.
- A DTD can be declared inline in your XML document, or as an external reference.

# Internal DTD example

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note      (to,from,heading,body)>
  <!ELEMENT to        (#PCDATA)>
  <!ELEMENT from       (#PCDATA)>
  <!ELEMENT heading   (#PCDATA)>
  <!ELEMENT body       (#PCDATA)>
]>
<note>
  <to>Joel</to>
  <from>Mom</from>
  <heading>Reminder</heading>
  <body>Don't forget father's day!</body>
</note>
```

# Same example, external DTD

```
<?xml version="1.0"?>  
<!DOCTYPE note SYSTEM "note.dtd">  
<note>  
  <to>Joel</to>  
  <from>Mom</from>  
  <heading>Reminder</heading>  
  <body>Don't forget father's day!</body>  
</note>
```

# Why use a DTD?

- XML provides an application independent way of sharing data.
- With a DTD, independent groups of people can agree to use a common DTD for interchanging data.
- Your application can use a standard DTD to verify that data that you receive from the outside world is valid.
- You can also use a DTD to verify your own data.

# Why not DTDs?

- DTDs are not XML themselves
- DTDs can't fully describe everything that's legal XML.
- DTDs can't check for things that you might want to protect against (for example, restricting an attribute)
- The web has many good tutorials on building DTDs (beyond the scope of this class)

# XML Schema

- Most people have changed to using XML Schema (an XML based description language) to define their XML (if they define it at all!)
- XML Schema is a language in XML
- Comes with a series of built-in types
- You extend them to define valid XML documents for your application

# XML Schema

- Every valid schema is a valid XML document
- Schemas can therefore themselves be validated
- Many editors and tools to help write schemas
- My favorite is nxml (for recent versions of emacs)

<http://www.thaiopensource.com/nxml-mode/>

# XML Schema

- Schemas define types
- Simple types are built in but restrictions can be added
- Complex types are made up of simple types
- An import tag implies using multiple schemas
- Often a schema is split into multiple documents

# XML Namespaces

- Since so many schemas and documents could be merged together, namespaces allow the same tag name to be used by different documents
- Namespaces (and an abbreviation) are declared in the opening `<xs:schema>` tag
- Namespaces are traditionally in the form of URLs, but these *mean nothing!*

# XML Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
  targetNamespace='http://www.uvstudios.com'
  xmlns:tns='http://www.uvstudios.com'
  xmlns:pc='http://www.uvstudios.com/pc' />

<xs:import namespace='http://www.w3.org/2001/XMLSchema'
  perlclass='schemaLocation='<u>http://uvstudios.com/schemas/perlclass.xsd</u>' />

<xs:complexType name='Person'>
  <xs:sequence>
    <xs:element name='login' type='pc:LoginType' />
    <xs:element name='realname' type='xsd:string' />
  </xs:sequence>
</xs:complexType>

</xs:schema>
```

This is the only URL that means a real web location!

# XSLT

- eXtensible Stylesheet Language Transform
- XSL can be used to define how an XML file should be displayed by transforming the XML file into a format that is recognizable to a browser.
- One such format is HTML.
- Normally XSL do this by transforming each XML element into an HTML element.

# XSLT

- So, every XSL style-sheet is valid XML that conforms to the XSL specification.
- Style-sheets written in XSL define a transformation of an XML document.
- Transforms XML documents into some other form of text

# XSLT

- Transforms XML documents into some other form of text
- some style-sheets are written to result in another XML document-
- some are written to produce plain text- some are even written to produce other languages like Java- many others written to result in HTML

# XSLT

- Transforms XML documents into some other form of text
- this can result in a clean separation of data and display -
  - your data is cleanly marked in XML
  - the XSLT decides how to convert that data to HTML, which is how to display it

# XSLT

- It has been claimed that all computation is transformation - i.e. any computation can be done as a series of XSLT transformations on XML data
- It has also been claimed that XSLT is the W3C's attempt to re-solve the issue of display after Microsoft "embraced and extended" CSS and EcmaScript into IE

# XSLT Example

```
<?xml version="1.0"?>
<slide>
  <class>XML - eXtensible Markup Language</class>
  <title>Introduction</title>
  <bullets>
    <point>
      XML is a derivative of Standard Generalized Markup Language
      <bullets>
        <point>
          (as is HTML) - XHTML is legal SGML and XML and HTML.
        </point>
      </bullets>
    </point>
    <point>
      XML is a metalanguage - create your own document markups.
    </point>
    <point>
      Define your own tags!
    </point>
  </bullets>
</slide>
```

# XSLT Example

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

<xsl:template match="/">
  <xsl:apply-templates select="slide" />
</xsl:template>

<xsl:template match="slide" >
<html>
<head>
<title>
<xsl:value-of select="/slide/class/
title" />
</title>
</head>
<body>
<xsl:template match="/" />
</body>
</html>
</xsl:template>

<xsl:template match="bullets">
<ul>
<xsl:apply-templates
match="/" />
</ul>
</xsl:template>

<xsl:template match="point">
<li><xsl:value-of select="." /
></li>
</xsl:template>

</xsl:stylesheet>
```

# XSLT

- As a technique, it can be quite advanced programming
- As a language, it really has a lot to overcome
- It's extremely verbose, heavily recursive, very difficult to debug, merges both target and program syntax in a single file, eats a lot of memory and is very slow.

# XSLT - Duplicate a string

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform" xmlns:str="http://www.ora.com/XSLTcookbook/namespaces/strings">
  <xsl:template name="str:dup">
    <xsl:param name="input"/>
    <xsl:param name="count" select="1"/>
    <xsl:choose>
      <xsl:when test="not($count) or not($input)"/>
      <xsl:when test="$count = 1">
        <xsl:value-of select="$input"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:if test="$count mod 2">
          <xsl:value-of select="$input"/>
        </xsl:if>
        <xsl:call-template name="str:dup">
          <xsl:with-param name="input" select="concat($input,$input)"/>
          <xsl:with-param name="count" select="floor($count div 2)"/>
        </xsl:call-template>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

# XSLT - HTML tables

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/
XSL/Transform">
<xsl:output method="html"/>
<xsl:param name="URL"/>

<xsl:template match="/">
  <xsl:apply-templates select="*" mode="index"/>
  <xsl:apply-templates select="*" mode="content"/>
</xsl:template>

<xsl:template match="salesBySalesperson" mode="index">
  <!-- Non-standard xsl: saxon:document! -->
  <xsl:document href="index.html">
    <html>
      <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1"/>
      </head>

      <body bgcolor="#FFFFFF" text="#000000">
        <h1>Sales By Salesperson</h1>
        <xsl:apply-templates mode="index"/>
      </body>
    </html>
  </xsl:document>
</xsl:template>

<xsl:template match="salesperson" mode="index">
  <h2>
    <a href="{concat($URL,@name,'.html')}">
      <xsl:value-of select="@name"/>
    </a>
  </h2>
</xsl:template>

<xsl:template match="salesperson" mode="content">
  <xsl:document href="{concat(@name,'.html')}">
    <html>
      <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1"/>
      </head>
      <body bgcolor="#FFFFFF" text="#000000">
        <h1><xsl:value-of select="@name"/> Sales</h1>
        <table border="1" cellpadding="3">
          <tbody >
            <tr>
              <th>SKU</th>
              <th>Sales (in US $)</th>
            </tr>
            <xsl:apply-templates mode="content"/>
          </tbody>
        </table>
        <h2><a href="{concat($URL,'index.html')}">Home</a></h2>
      </body>
    </html>
  </xsl:document>

  <xsl:template match="product" mode="content">
    <tr>
      <td><xsl:value-of select="@sku"/></td>
      <td><xsl:value-of select="@totalSales"/></td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

# XSLT - HTML tables

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/
XSL/Transform">
<xsl:output method="html"/>
<xsl:param name="URL"/>

<xsl:template match="/">
  <xsl:apply-templates select="*" mode="index"/>
  <xsl:apply-templates select="*" mode="content"/>
</xsl:template>

<xsl:template match="salesBySalesperson" mode="index">
  <!-- Non-standard xsl: saxon:document! -->
  <xsl:document href="index.html">
    <html>
      <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1"/>
      </head>

      <body bgcolor="#FFFFFF" text="#000000">
        <h1>Sales By Salesperson</h1>
        <xsl:apply-templates mode="index"/>
      </body>
    </html>
  </xsl:document>
</xsl:template>

<xsl:template match="salesperson" mode="index">
  <h2>
    <a href="{concat($URL,@name,'.html')}">
      <xsl:value-of select="@name"/>
    </a>
  </h2>
</xsl:template>
```

```
<xsl:template match="salesperson" mode="content">
  <xsl:document href="{concat(@name,'.html')}">
    <html>
      <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1"/>
      </head>
      <body bgcolor="#FFFFFF" text="#000000">
        <h1><xsl:value-of select="@name"/> Sales</h1>
        <table border="1" cellpadding="3">
          <tbody >
            <tr>
              <th>SKU</th>
              <th>Sales (in US $)</th>
            </tr>
            <xsl:apply-templates mode="content"/>
          </tbody>
        </table>
        <h2><a href="{concat($URL,'index.html')}">Home</a></h2>
      </body>
    </html>
  </xsl:document>

<xsl:template match="product" mode="content">
  <tr>
    <td><xsl:value-of select="@sku"/></td>
    <td><xsl:value-of select="@totalSales"/></td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

# XPath

# Perl and XML

- Many many XML modules on CPAN
- They all have different functions:
- translating data between an XML instance and Perl data structures
- implementing one of the standard XML APIs
- special-purpose modules to simplify the execution of some specific XML-related task.

# Perl and XML

- Choosing the right XML module for your project depends largely upon the nature of the project and your past experience
- Usually you want to either create an XML document from another source, or extract data from an XML document

# Examples

- See website for examples of parsing and outputting XML using XML::Simple.
- XML and DOM
- XML and SAX

# XML::Simple

- Pretty simple...

```
use XML::Simple;

my $ref = XMLin([<xml file or string>] [, <options>]);

my $xml = XMLout($hashref [, <options>]);

# online plant catalog example
```

# XML::RSS

- RSS is Real Simple Syndication
- XML format for news headlines
- A website updates this xml document and serves it like any other page.
- This feed/channel/xml document can be read by programs

# XML::RSS

- First version (0.9) with wide acceptance released by Dave Winer of Userland Software
- Later versions (1.0 and 2.0) greatly expanded the format. Not completely compatible (unfortunately).
- Mostly, all versions have title, link, and description tags, and more.

# XML::RSS

- The channel itself can have a title, link, and description - this is the website itself.
- Each story (item) has a title, link, and description.
- The description may be empty, or simply repeat the title, a short summary, or the full text of the article.

# A sample feed

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
<channel>
<title>Finnegan Picture Of The Day</title>
<link>http://www.trederfamily.org/finnpotd/</link>
<description></description>
<copyright>Copyright 2005</copyright>
<lastBuildDate>Thu, 05 May 2005 22:09:37 -0800</
lastBuildDate>
<generator>http://www.movabletype.org/?v=3.121</
generator>
<docs>http://blogs.law.harvard.edu/tech/rss</docs>

<!-- items -->
```

# A sample feed

```
<item>
<title>Koala Bear</title>
<description><![CDATA[<p><p>On his five month birthday, he was clinging
to his daddy like a little koala bear. He now laughs out
loud and enjoys singing. He also started to say, "oo,"
instead of "oh," or "ah."<br />
Besides getting eight teeth he continued to progress
rapidly.</p>]]></description>
<link>http://www.trederfamily.org/finnpotd/archives/
2005/05/koala_bear.html</link>
<guid>http://www.trederfamily.org/finnpotd/archives/
2005/05/koala_bear.html</guid>
<category></category>
<pubDate>Thu, 05 May 2005 22:09:37 -0800</pubDate>
</item>
```

# RSS in Perl

- Using RSS from perl is easy. First, you need to get the XML feed itself using LWP::Simple.

```
my $xml = LWP::Simple::get( $url );  
  
# Then use XML::RSS::Parser::Lite to parse the file.  
  
my $rss = XML::RSS::Parser::Lite->new();  
$rss->parse($xml);
```

# RSS in Perl

- You can then use the RSS functions to get the individual pieces out:

```
my $count = $rss->count;    # returns number of items
my $item  = $rss->get($i);  # gets the $i'th story.
```

```
my $title = $item->get('title');
# gets the title of the story
```

```
my $link  = $item->get('url');
# gets the url directly to the full story text
```

```
my $description = $item->get('description');
```

# RSS in Perl

```
my $count = $rss->count;

for my $i (0..($count-1)) {
    my $item = $rss->get($i);
    my $title = $item->get('title');
    my $link = $item->get('url');
    my $description = $item->get('description')
        || ' No description ';

    my $story = News::Story->new
        (title => $title,
         link => $link,
         description => $description );
}
```