Getting Started with XML: A Manual and Workshop

by Eric Lease Morgan

Getting Started with XML: A Manual and Workshop

by Eric Lease Morgan

Special thanks go to David Cherry, a long-time friend, who provided the drawings. Lori Bowen Ayre deserves a round of applause for providing editorial support. Infopeople are the folks who sponsored the whole thing. Roy Tennant helped with proofreading. Thank you! --ELM

For possibly more up-to-date information see the Getting Started With XML home page [http://infomotions.com/musings/getting-started/] .

This is the second release of this document, dated Monday, April 21, 2003 (Dingus Day). The first public release of this document was dated Saturday, February 22, 2003.

Table of Contents

Preface

Designed for librarians and library staff, this workshop introduces participants to the extensible markup language (XML) through numerous library examples, demonstrations, and structured hands-on exercises. Through this process you will be able to evaluate the uses of XML for making your library's data and information more accessible to people as well as computers. Examples include adding value to electronic texts, creating archival finding aids, and implementing standards compliant Web pages. By the end of the manual you will have acquired a thorough introduction to XML and be able to: 1) list the six rules governing the syntax of XML documents, 2) create your very own XML markup language, 3) write XML documents using a plain text editor and validate them using a Web browser, 4) apply page layout and typographical techniques to XML documents using cascading style sheets, 5) create simple XML documents using a number of standard XML vocabularies important to libraries such as XHTML, TEI, and EAD, and finally, 6) articulate why XML is important for libraries.

Hilights of the manual include:

- Demonstrations of the use of XML in libraries to create, store, and disseminate electronic texts, archival finding aids, and Web pages
- Teaching the six simple rules for creating valid XML documents
- Practicing with the combined use of cascading style sheets and XML documents to display data and information in a Web browser
- Practicing with the use of XHTML and learning how it can make your website more accessible to all types of people as well as Internet robots and spiders
- Demonstrating how Web pages can be programmatically created using XSLT allowing libraries to transform XML documents into other types of documents
- Enhancing electronic texts with the use of the TEI markup allowing libraries to add value to digitized documents
- Writing archival finding aids using EAD thus enabling libraries to unambiguously share special collection information with people and other institutions

The manual is divided into the following chapters/sections:

1. What is XML and why should I care?
2. A gentle introduction to XML markup
3. Creating your own markup
4. Rendering XML with cascading stylesheets
5. Transforming XML with XSL
6. Validating XML with DTDs
7. Introduction to selected XML languages: XHTML, TEI, DocBook
8. Sharing metadata with RDF and OAI
9. Selected reading list

Part I. XML in general

Chapter 1. Introduction

What is XML and why should I care?

In a sentence, the eXtensible Markup Language (XML) is an open standard providing the means to share data and information between computers and computer programs as unambiguously as possible. Once transmitted, it is up to the receiving computer program to interpret the data for some useful purpose thus turning the data into information. Sometimes the data will be rendered as HTML. Other times it might be used to update and/or query a database. Originally intended as a means for Web publishing, the advantages of XML have proven useful for things never intended to be rendered as Web pages.

Think of XML as if it represented tab-delimited text files on steroids. Tab-delimited text files are very human readable. They are easy to import into word processors, databases, and spreadsheet applications. Once imported, their simple structure make their content relative easy to manipulate. Tab-delimited text files are even cross-platform and operating system independent (as long as you can get around the carriage-return/linefeed differences between Windows, Macintosh, and Unix computers). See the following example

```
Amanda  10  dog  brown
Blake   12  dog  blue
Jack    3   cat  black
Loosey  1   cat  brown
Stop    5   pig  brown
Tilly   14  cat  silver
```

The problem with tab-delimited text files are two-fold. First, the meaning of each tab-delimited values are not explicitly articulated. In order to know what each value is suppose to represent it is necessary to be given (or be told ahead of time) some sort of map or context for the data. Second and more importantly, tab-delimited text files can only represent a very simple data structure, a data structure analogous to a simple matrix of rows and columns. Put another way, tab-delimited text files are exactly like flat file databases. There is no easy, standardized way of representing data in a hierarchial fashion.

Much like tab-delimited text files, XML files are very human readable since they are allowed to contain only Unicode characters -- a considerably extended version of the original ASCII character code set. Additionally, XML files are operating system and application independent with the added benefit of making carriage-return/linefeed sequences almost a non-issue.

Unlike tab-delimited files, XML files explicitly state the meaning of each value in the file. Very little is left up to

guesswork. Each element's value is explicitly described. XML turns data into information. The tab-delimited file from Figure 1.1 is simply an organized list of words and numbers. They have no context and therefore they only represent data. On the other hand, the words and numbers in XML files are given value and context, and therefore are transformed from data to information. Furthermore, it is very easy to create hierarchial data structures using XML. Figure 1.2 illustrates these concepts. Without very much examination, it becomes apparent the data represents a list of pets, specifically, six pets, and each pet has a name, age, type, and color. Was that as apparent in the previous example?

```
<pets>
  <pet>
   <name>Tilly</name>
   <age>14</age>
   <type>cat</type>
   <color>silver</color>
  </pet>
  <pet>
   <name>Amanda</name>
   <age>10</age>
   <type>dog</type>
   <color>brown</color>
  </pet>
  <pet>
   <name>Jack</name>
   <age>3</age>
   <type>cat</type>
   <color>black</color>
  </pet>
  <pet>
   <name>Blake</name>
   <age>12</age>
   <type>dog</type>
   <color>blue</color>
  </pet>
  <pet>
   <name>Loosey</name>
   <age>1</age>
   <type>cat</type>
   <color>brown</color>
  </pet>
  <pet>
   <name>Stop</name>
   <age>5</age>
   <type>pig</type>
   <color>brown</color>
  </pet>
</pets>
```

As the world's production economies move more and more towards service economies, the stuff of business becomes more tied to data and information. Similarly, libraries are becoming less about books and more about the ideas and concepts manifested in the books. In both of these spheres of influence there needs to be a way to move data and information around efficiently and effectively. XML data shared between computers and computer programs via the hypertext transfer protocol represents an evolving method to facilitate this sharing, a method generically called Web Services.

For example, an XML markup called RSS (Rich Site Summary) is increasingly used to syndicate lists of uniform resource locators (URL's) representing news stories found on websites. RDF (Resource Description Framework) is an XML markup used to encapsulate meta data about content found at the end of URL's. TEI (Text Encoding Initiative) and TEILite are both an SGML and well as an XML markup used to explicitly give value to things found in literary works. Similarly, another XML language called DocBook is increasingly used to markup computer-related books or articles. The Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) uses XML to gather meta data about the content found at remote Internet sites.

As information professionals, it behooves us to learn how to exploit the capabilities of XML, because XML is a tool making it easy to unambiguously and as platform independently as possible communicate information in a globally networked environment. Isn't that what librarianship and information science is all about?

Chapter 2. A gentle introduction to XML markup

XML syntax

XML documents have syntactic and semantic structures. The syntax (think spelling and punctuation) is made up of a minimum of rules such as but not limited to:

1. XML documents always have one and only one root element
2. Element names are case-sensitive
3. Elements are always closed
4. Elements must be correctly nested
5. Elements' attributes must always be quoted
6. There are only five entities defined by default (<, >, &, ", and ')

Each of these rules are described in more detail below.

XML documents always have one and only one root element

The structure of an XML document is a tree structure where there is one trunk and optionally many branches. The single trunk represents the root element of the XML document. Consider the following, overly simplified, HTML document, Figure 2.1:

```
<html>
 <head>
  <title>Hello, World</title>
 </head>
 <body>
  <p>Hello, World</p>
 </body>
</html>
```

This document structure should look familiar to you. It is a valid XML document, and it only contains a single root element, namely html. There are then two branches to the document, head and body.

Element names are case-sensitive

Element names, the basic vocabulary of XML documents, are case-sensitive. In Figure 2.1 there are five elements: html, head, title, body, and p. Since each element's name is case-sensitive, the element html does not equal HTML, nor does it equal HTmL or Html. The same is true for the other elements.

Elements are always closed

Each element is denoted by opening and closing brackets, the less than sign (<) and greater than sign (>), respectively. XML elements are rarely empty; they are usually used to provide some sort of meaning or context to some data, and consequently, XML elements usually surround data. Each of the elements is Figure 2.1 are opened and closed. For example, the title of the document is denoted with the <title> and </title> elements and the only paragraph of the document is denoted with and elements. An opened element does not contain the initial forward slash but closing elements do.

Sometimes elements can be empty such as the break tag in XHTML. In such cases the element is opened and closed at the same time, and it is encoded like this:
.

Elements must be correctly nested

Consecutive XML elements may not be opened and then closed without closing the elements that were opened last first. Doing so is called improper nesting. Take the following incorrect encoding of an XHTML paragraph:

```
<p>This is a test. This is a test of the <em>
<strong>Emergency</em> Broadcast System.</strong></p>
```

In the example above the em and strong elements are opened, but the em element is closed before the strong element. Since the strong element was opened after the em element it must be closed before the em element. Here is correct markup:

```
<p>This is a test. This is a test of the <strong>
<em>Emergency</em> Broadcast System.</strong></p>
```

Elements' attributes must always be quoted

XML element are often qualified using attributes. For example, an integer might be marked up as a length and the length element might be qualified to denote feet as the unit of measure. For example: <length unit='feet'>5</length>. The attribute is named unit, and it's value is always quoted. It does not matter whether or not it is quoted with an apostrophe (') or a double quote (").

There are only five entities defined by default

Certain characters in XML documents have special significance, specifically, the less than (<), greater than (>), and ampersand (&) characters. The first two characters are used to delimit the existence of element names. The ampersand is used to delimit the display of special characters commonly known as entities; they ampersand character is

the "escape" character. Consequently, if you want to display any of these three characters in your XML documents, then you must express them in their entity form:

- to display the & character type &
- to display the < character type <
- to display the > character type >

XML processors, computer programs that render XML documents, should be able interpret these characters without the characters being previously defined.

There are two other characters that can be represented as entity references:

- to display the ' character optionally type '
- to display the " character optionally type "

XML semantics

The semantics of an XML document (think grammar) is an articulation of what XML elements can exist in a file, their relationship(s) to each other, and their meaning. Ironically, this is the really hard part about XML and has manifested itself as a multitude of XML "languages" such as: RSS, RDF, TEILite, DocBook, XMLMARC, EAD, XSL, etc. In the following, valid, XML file there are a number of XML elements. It is these elements that give the data value and meaning:

```
<catalog>
 <work type='prose' date='1906'>
  <title>The Gift Of The Magi</title>
  <author>O Henry</author>
 </work>
 <work type='poem' date='1845'>
  <title>The Raven</title>
  <author>Edgar Allen Poe</author>
 </work>
 <work type='play' date='1601'>
  <title>Hamlet</title>
  <author>William Shakespeare</author>
 </work>
</catalog>
```

Exercise - Checking XML syntax

In this exercise you will learn to identify syntactical errors in XML files.

1. Examine the following file. Circle all of it's syntactical errors, and write in the corrections.

```
<name>Oyster Soup</name>
<author>Eric Lease Morgan</author>
<copyright holder=Eric Lease Morgan>&copy; 2003</copyright>
<ingredients>
 <list>
```

```
  <item>1 stalk of celery
  <item>1 onion
  <item>2 tablespoons of butter
  <item>2 cups of oysters and their liquor
  <item>2 cups of half & half
 </list>
</ingredients>
<process>
 <P>Begin by sauteing the celery and onions in butter until soft.
 Add oysters, oyster liquor, and cream. Heat until the oysters float.
 Serve in warm bowls.</p>
 <p><i>Yummy!</p></i>
</process>
```

A. Check for one and only one root element. Is there a root element?

B. Check for quoted attribute values. Are the attributes quoted?

C. Check for invalid use of entities. There are two errors in the file.

D. Check for properly opened and closed element tags. Five elements are not closed.

E. Check for properly nested elements. Two elements are not nested correctly.

F. Check for case-sensitive element naming. One element is not correctly cased.

Chapter 3. Creating your own markup

Purpose and components

The "X" in XML stands for extensible. By this the creators of XML mean it should be easy to create one's own markup -- a vocabulary or language intended to describe a set of data/information. The key to creating an XML mark up language is to first articulate what the documents will be used for, and second the ability to specify the essential components of a document and assign them elements. The process of creating an XML mark up is similar to the process of designing a database application. You must ask yourself what data you will need and create places for that data to be saved.

Creating a markup for a letter serves as an excellent example:

```
December 11, 2002

Melvile Dewey
Columbia University
New York, NY

Dear Melvile,

     I have been reading your ideas concering the nature of
librarianship, and I find them very intriguing. I would love the
opportunity to discuss with you the role of the card catalog in today's
libraries considering the advent to World Wide Web. Specifically, how
are things like Google and Amazon.com changing our patrons' expectations
of library services? Mr. Cutter and I will be discussing these ideas at
the next Annual Meeting, and we are available at the follow dates/times:

        * Monday, 2-4
        * Tuesday, 3-5
        * Thursday, 1-3

     We hope you can join us.

                                          Sincerely, S. R. Ranganathan
```

As you read the letter you notice sections common to many letters. By analyzing these sections it is possible to create a list of XML elements. For example, the letter contains a date, a block of text describing the addressee, a greeting, one or more paragraphs of text, a list, and a closing statement. Upon closer examination, some of your sections

have subsections. For example, the addressee has a name, a first address line, and a second address line. Further, the body of the letter might have some sort of emphasis.

The division into smaller and smaller subsections could go all the way down to individual words. Where to stop? Only create elements for pieces of data you are going to use. If you never need to know the city or state of your addressee, then don't create an element for them. Ask yourself, what is the purpose of the document? What sort of information do you want to hilight from its content? If you wanted to create lists of all the cities you sent letters to, then you will need to demarcate the values for city. If you need to extract each and every sentence from your document, then you will have to demarcate them as well. Otherwise, save yourself the time and energy and keep it simple.

Once you have articulated the parts of the document you want to mark up you have to give them names. XML element names can contain standard English letters A - Z and a - z as well as integers 0 - 9. They can also contain non-English letters and three punctuation characters: underscore (_), hyphen (-), and period (.). Element names may not contain white space (blanks, tabs, return characters), nor other punctuation marks. Play it save. Use letters.

Now it is time to actually create a few elements. Based on the previous discussion. We could create a set of element names such as this:

- letter
- date
- addressee
- name
- address_one
- address_two
- greeting
- paragraph
- italics
- list
- item
- closing

Using these elements as a framework, it is possible to mark up the text in the following manner:

```
<letter>

 <date>December 11, 2002</date>

 <addressee>
  <name>Melvile Dewey</name>
  <address_one>Columbia University</address_one>
  <address_two>New York, NY</address_two>
 </addressee>

 <greeting>Dear Melvile,</greeting>

 <paragraph>
```

```
  I have been reading your ideas concerning nature of librarianship, and
  <italics>I find them very intriguing</italics>. I would love the
  opportunity to discuss with you the role of the card catalog in today's
  libraries considering the advent to World Wide Web. Specifically, how
  are things like Google and Amazon.com changing our patrons' expectations
  of library services? Mr. Cutter and I will be discussing these ideas at
  the next Annual Meeting, and we are available at the follow dates/times:
 </paragraph>

 <list>
  <item>Monday, 2-4</item>
  <item>Tuesday, 3-5</item>
  <item>Thursday, 1-3</item>
 </list>

 <paragraph>We hope you can join us.</paragraph>

 <closing>Sincerely, S. R. Ranganathan</closing>

</letter>
```

Exercise - Creating your own XML mark up

In this exercise you will create your own XML markup, a markup describing a simple letter.

1. Consider the following letter.

```
February 3, 2003

American Library Association
15 Huron Street
Chicago, IL 12304

To Whom It May Concern:

It has come to my attention that the Association no longer wants
to spend money on posters of famous people advocating reading.
What is wrong with you guys! Don't you know that reading is
FUNdamental? These posters really get me and my patrons going. I
thought they were great.

Please consider re-instating the posters.

Sincerely, B. Ig Reeder
```

2. As a group, decide what elements to use to mark up the letter as an XML file.

   A. What can our root element be?

   B. What sections make up the letter? What element names can we give these sections?

   C. Some of the sections, such as the address, greeting, and saluation have sub-sections. What should we call these sub-sections?

   D. Use a pen or pencil to mark up the letter above using the elements decided upon.

3. Mark up the letter as an XML document, and validate its syntax using a Web browser.

A. Use NotePad to open the file named ala.txt on the distributed CD.

B. Add the root element to the beginning and ending of the file.

C. Mark up each section and sub-section of the letter with the element names decided upon.

D. Save the file with the name ala.xml.

E. Open ala.xml in your Web browser, and fix any errors that it may report. If there are no errors, then congratulations, you have marked up your first XML document.

Chapter 4. Rendering XML with cascading style sheets

Introduction

Cascading style sheets (CSS) represent a method for rendering XML files into a more human presentation. CSS files exemplify a method for separating presentation from content.

CSS have three components: layout, typography, and color. By associating an XML file with a CSS file and processing them with a Web browser, it is possible to display the content of the XML file in an aesthetically pleasing manner.

CSS files are made up of sets of things called selectors and declarations. Each selector in a CSS file corresponds to an element in an XML file. Each selector is then made of up declarations -- standardized name/value pairs -- denoting how the content of XML elements are to be displayed. They look something like this: note { display: block; }.

Here is a very simple XML document describing a note:

```
<?xml-stylesheet type="text/css" href="note.css"?>
<note>
 <para>Notes are very brief documents.</para>
 <para>They do not contain very much content.</para>
</note>
```

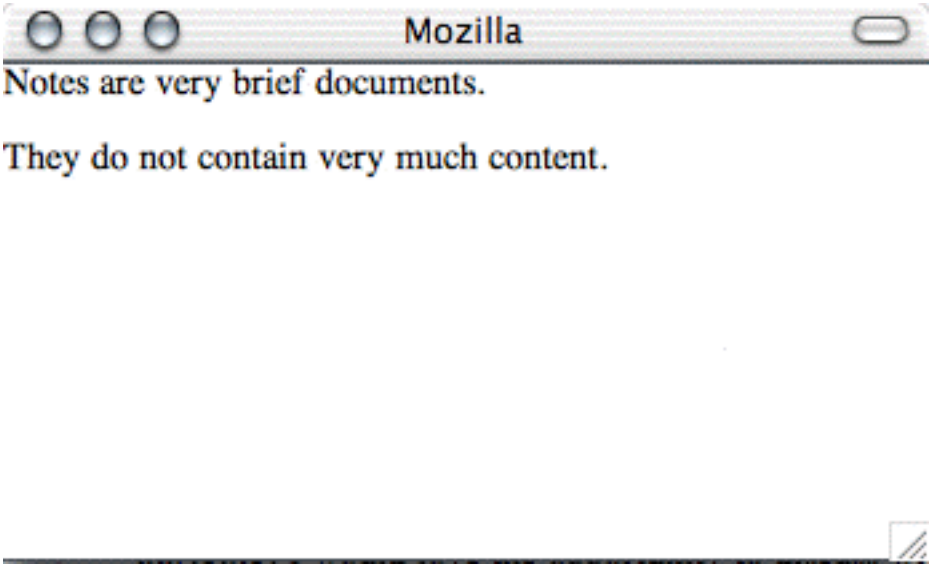
The first thing you will notice about the XML document is the addition of the very first line, an XML processing instruction. This particular instruction tells the application reading the XML file to render it using a CSS file named note.css. The balance of the XML file should be familiar to you.

If I wanted to display the contents of the note such that each paragraph were separated by a blank line, then the CSS file might look like this:

```
note { display: block; }
para { display: block; margin-bottom: 1em; }
```

In this CSS file there are two selectors corresponding to each of the elements in the XML file: note and para. Each selector is associated with one or more name/value pairs (declarations) describing how the content of the elements are to be displayed. Each name is separated from the value by a colon (:), the name/value pairs are separated from each other by a semicolon (;), and all the declarations associated with a selector are grouped together with curly braces({}).

Opening note.xml in a relatively modern Web browser should result in something looking like this:



Be forewarned. Not all web browsers support CSS similarly. (What a surprise!) In general, you will get minimal performance from Netscape Navigator 4.7 and Internet Explorer 5.0. Much better implementations of CSS are built into Mozilla 1.0 and Internet Explorer 6.0. Your milage will vary.

The key to using CSS files is knowing how to create the name/value pair declarations. For a comprehensive list of these name/value pairs see the World Wide Web Consortium's description of CSS [http://www.w3.org/TR/REC-CSS2/propidx.html] . A number of them are described below.

display

The display property is used to denote whether or not an element is to be displayed, and if so, how but only in a very general way. The most important values for display are: inline, block, list-item, or none. Inline is the default value. This means the content of the element will not include a line break after the content; the content will be displayed as a line of text. Giving display a value of block does create line breaks after the content of the element. Think of blocks as if they were paragraphs. The list-item value is like block, but it also indents the text just a bit. The use of none means the content will not be displayed; the content is hidden. Examples include:

- display: none;
- display: inline;
- display: block;
- display: list-item;

margin

The margin property is used to denote the size of white space surrounding blocks of text. Values can be denoted in terms of percentages (%), pixels (px), or traditional typographic conventions such as the em unit (em). When the simple margin property is given a value, the value is assigned to the top, bottom, left, and right margins simultaneously. It is possible to specify specific margins using the margin-top, margin-bottom, margin-left, and margin-right properties. Examples include:

- margin: 5%;
- margin: 10px;
- margin-top: 2em;
- margin-left: 85%;
- margin-right: 50px;
- margin-bottom: 1em;

text-indent

Like the margin property, the text-indent property can take percentages, pixels, or typographic units for values. This property is used to denote how particular lines in blocks of text are indented. For example:

- text-indent: 2em;
- text-indent: 3%;

text-align

Common values for text-align are right, left, center, and justify. They are used to line up the text within a block of text. These values operate in the same way your word processor aligns text. For example:

- text-align: right;
- text-align: left;
- text-align: center;
- text-align: justify;

list-style

Bulleted lists are easy to read and used frequently in today's writing styles. If you want to create a list, then you will want to use first use the selector display: list-item for the list in general, and then something like disc, circle, square, or decimal for the list-style value. For example:

- list-style: circle;
- list-style: square;
- list-style: disc;

- list-style: decimal;

font-family

Associate font-family with a selector if you want to describe what font to render the XML in. Values include the names of fonts as well as a number of generic font families such as serif or sans-serif. Font family names containing more than one word should be enclosed in quotes. Examples:

- font-family: helvetica;
- font-family: times, serif;
- font-family: 'cosmic cartoon', sans-serif;

font-size

The sizes of fonts can be denoted with exact point sized as well as relative sizes such as small, x-small, or large. For example:

- font-size: 12pt;
- font-size: small;
- font-size: x-small;
- font-size: large;
- font-size: xx-large;

font-style

Usual values for font-style are normal or italic denoting how the text is displayed as in:

- font-style: normal;
- font-style: italic;

font-weight

This is used to denote whether or not the font is displayed in bold text or not. Typical values for font-weight are normal and bold:

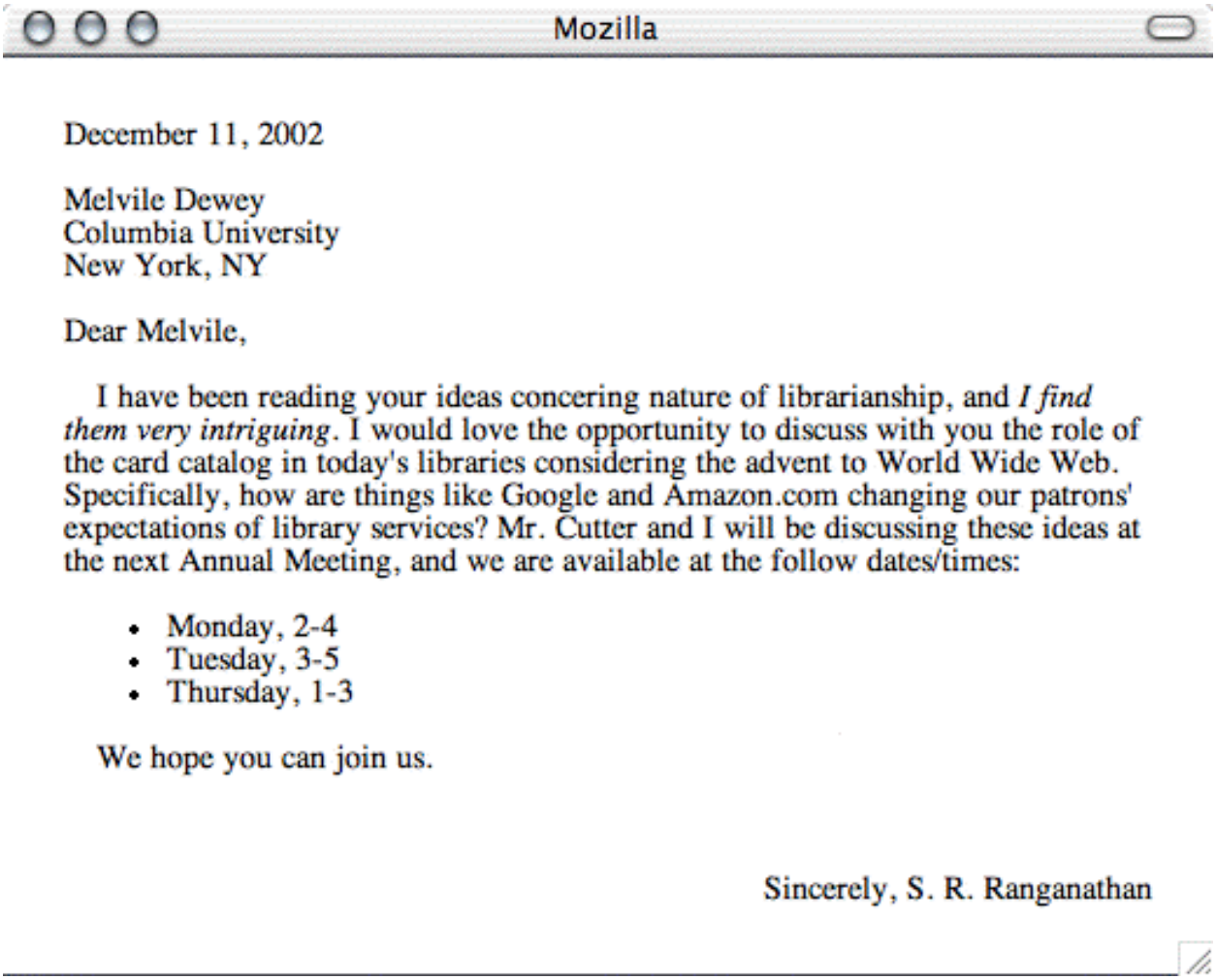
- font-weight: normal;
- font-weight: bold;

Putting it together

Below is a CSS file intended to be applied against the letter.xml file previously illustrated. Notice how each element in the XML file has a corresponding selector in the CSS file. In order to tell your Web browser to use this CSS file, you will have to add the xml-stylesheet processing instruction (<?xml-stylesheet type="text/css" href="letter.css" ?>) to the top of letter.xml.

```
letter {
 display: block;
 margin: 5%;
}

date, addressee {
 display: block;
 margin-bottom: 1em;
}

name, address_one, address_two { display: block; }

greeting, list {
 display: block;
 margin-bottom: 1em;
}

paragraph {
 display: block;
 margin-bottom: 1em;
 text-indent: 1em;
}

italics {
 display: inline;
 font-style: italic;
}

list { display: block; }

item {
 display: list-item;
 list-style: inside;
 text-indent: 2em;
}

closing {
 display: block;
 margin-top: 3em;
 text-align: right;
}
```

Once rendered the resulting XML file should look something like this:



Tables

Tables are two-dimensional lists; they are a matrix of rows and columns. A very simple list of books (a catalog) lends itself to a tabled layout since each book (work) in the list has a number of qualities such as title, author, type, and date. Each work represents a row, and the title, author, type, and date represent columns.

Here is an XML file representing a simple, rudimentary catalog. Notice the XML processing instruction directing any XML processor to render the content of the file using the CSS file catalog.css:

```
<?xml-stylesheet href='catalog.css' type='text/css'?>
<catalog>
 <caption>This is my personal catalog.</caption>
 <structure>
  <title>Title</title>
  <author>Author</author>
  <type>Type</type>
  <date>Date</date>
 </structure>
 <work>
  <title>The Gift Of The Magi</title>
  <author>O Henry</author>
  <type>prose</type>
  <date>1906</date>
```

```
 </work>
 <work>
  <title>The Raven</title>
  <author>Edgar Allen Poe</author>
  <type>prose</type>
  <date>1845</date>
 </work>
 <work>
  <title>Hamlet</title>
  <author>William Shakespeare</author>
  <type>prose</type>
  <date>1601</date>
 </work>
</catalog>
```

CSS provides support for tables, but again, present-day browsers do not render tables equally well. To create a table you must you must learn at least three new values for an element's display value:

1. display: table;
2. display: table-row;
3. display: table-cell;

Using the catalog example above, display: table will be associated with the catalog element, display: table-row will be associated with the work element, and display: table-cell will be associated with the title, author, type, and date elements.

Additionally, you might want to use these values to make your tables more complete as well as more accessible:

1. display: table-caption;
2. display: table-header-group;

Table-caption is used to give an overall description of the table. Table-header-group is used to denote the labels for the column headings.

Exercise - Displaying XML Using CSS

In this exercise you will learn how to write a CSS file and use it to render an XML file.

1. Create a CSS file intended to render the file named ala.xml created in a previous exercise.
    A. Open ala.xml in NotePad.
    B. Add the XML processing instruction <?xml-stylesheet href="ala.css" type="text/css"?> to the top of the file. Save it.
    C. Create a new, empty file in NotePad, and save it as ala.css.
    D. In ala.css, list each XML element in ala.xml on a line by itself.
    E. Assign each element a display selector with a value of block (ex. para { display: block; }).

F. Open ala.xml in your Web browser to check your progress.

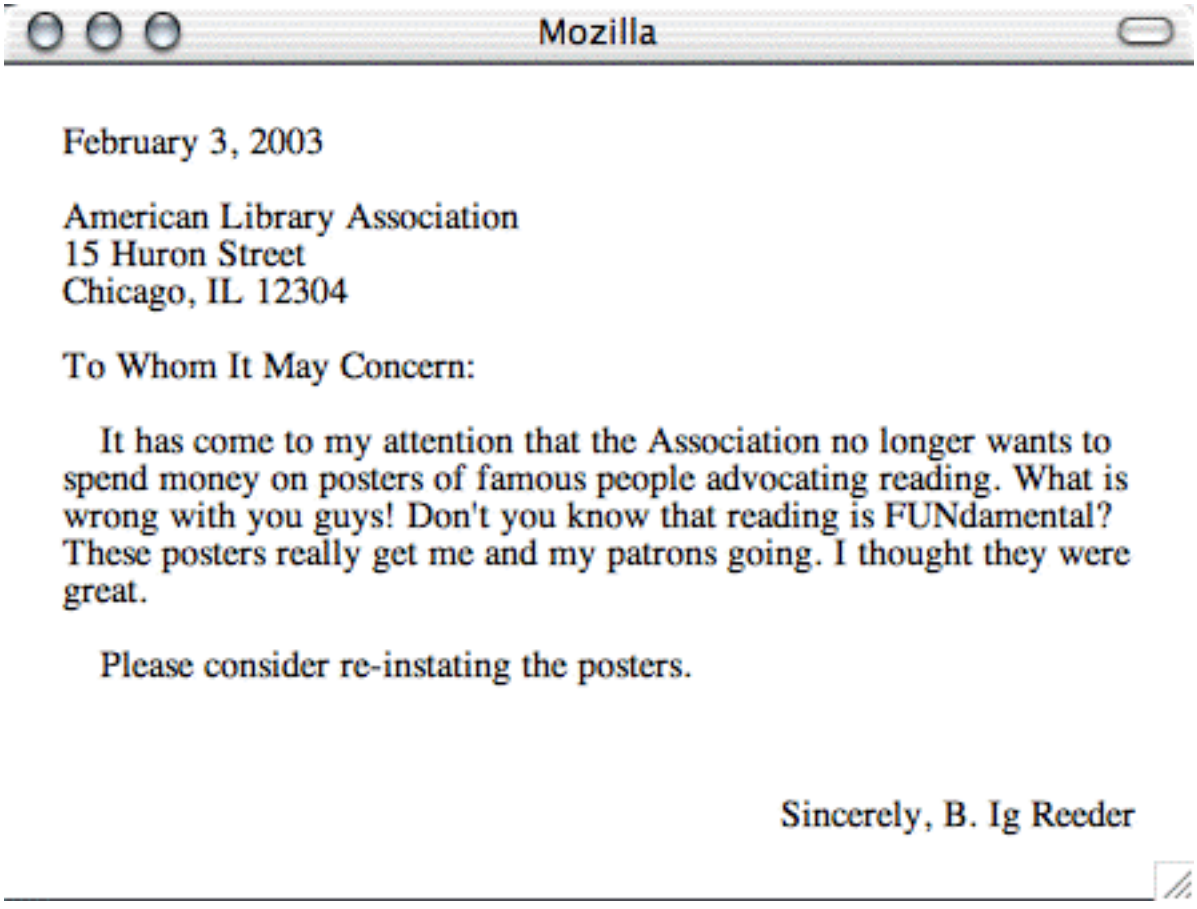
G. Add a blank line between each of the letter's sections by adding a margin-bottom: 1em to each section's selector (ex. para { display: block; margin-bottom: 1em; }).

H. Open ala.xml in your Web browser to check on your progress.

I. Change the display selector within the salutation so its sub-element is displayed as inline text, not a block (ex. salutation { display: inline; }).

J. Open ala.xml in your Web browser to check on your progress.

2. Indent the paragraphs by adding text-indent: 2em; to the para element. The final result should look something like this:



Chapter 5. Transforming XML with XSLT

Introduction

Besides CSS files, there is another method for transforming XML documents into something more human readable. Its called eXtensible Stylesheet Language: Transormation (XSLT). XSLT is a programming language implemented as an XML semantic. Like CSS, you first write/create an XML file, you then write an XSLT file and use a computer program to combine the two to make a third file. The third file can be any plain text file including another XML file, a narrative text, or even a set of sophisticated commands such as structured query language (SQL) queries intended to be applied against a relational database application.

Unlike CSS or XHTML, XSLT is a programming language. It is complete with input parameters, conditional processing, and function calls. Unlike most programming languages, XSLT is declarative and not procedural. This means parts of the computer program are executed as particular characteristics of the data are met and less in a linear top to bottom fashion. This also means it is not possible to change the value of variables once they have been defined.

There are a number of XSLT processors available for various Java, Perl, and operating-system specific platforms:

- Xerces and Xalan [http://xml.apache.org/] - Java-based implementations
- xsltproc [http://xmlsoft.org/XSLT/xsltproc2.html] - A binary application built using a number of C libraries, and also comes with a program named xmllint used to validate XML documents
- Sablotron [http://www.gingerall.com/charlie/ga/xml/p_sab.xml] - Another binary distribution built using C++ libraries and has both a Perl and a Python API
- Saxon [http://saxon.sourceforge.net/] - another Java implementation

Displaying narrative text

In this section we will transform a letter created previously into an XHTML file. To do so we will first create an XSLT file taking advantage of a number of XML commands and then combine the XSLT file with our letter using an XSLT processor. Here is a list of the various XSLT commands we will be using. Remember XSLT is a programming language in the form of an XML file. Therefore, each of the commands is an XML element, and commands are qualified using XML attributes.

- stylesheet - This is the root of all XSLT files. It requires attributes defining the XSLT namespace and version number. This is pretty much the standard XSLT stylesheet definition: <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">.

- output - This is used to denote what type of text file will be created as output and whether or not it requires indentation and/or a DTD specification. For example, this use of output tells the XSLT processor to indent the output to make the resulting text easier to read: <xsl:output indent="yes" />.

- template - This command is used to match/search for a particular part of an XML file. It requires an attribute named match and is used to denote what branch of the XML tree to process. For example, this use of template identifies all the things in the root element of the XML input file: <xsl:template match="/">.

- value-of - Used to output the result of the required attribute named select which defines exactly what to output. In this example, the XSLT processor will output the value of a letter's date element: <xsl:value-of select="/letter/date/" />.

- apply-templates - Searches the current XSLT file for a template named in the command's select statement or outputs the content of the current node of the XML file if there is no corresponding template. Here the apply-templates command tells the processor to find templates in the current XSLT file matching paragraph or list elements: <xsl:apply-templates select="paragraph | list" />.

- Besides XSLT commands (elements), XSLT files can contain plain text and/or XML markup. When this plain text or markup is encountered, the XSLT processor is expected to simply output these values. This is what allows us to create XHTML output. The processor reads an XML file as well as the XSLT file. As it reads the XSLT file it processes the XSLT commands or outputs the values of the non-XSLT commands resulting in another XML file or some other plain text file.

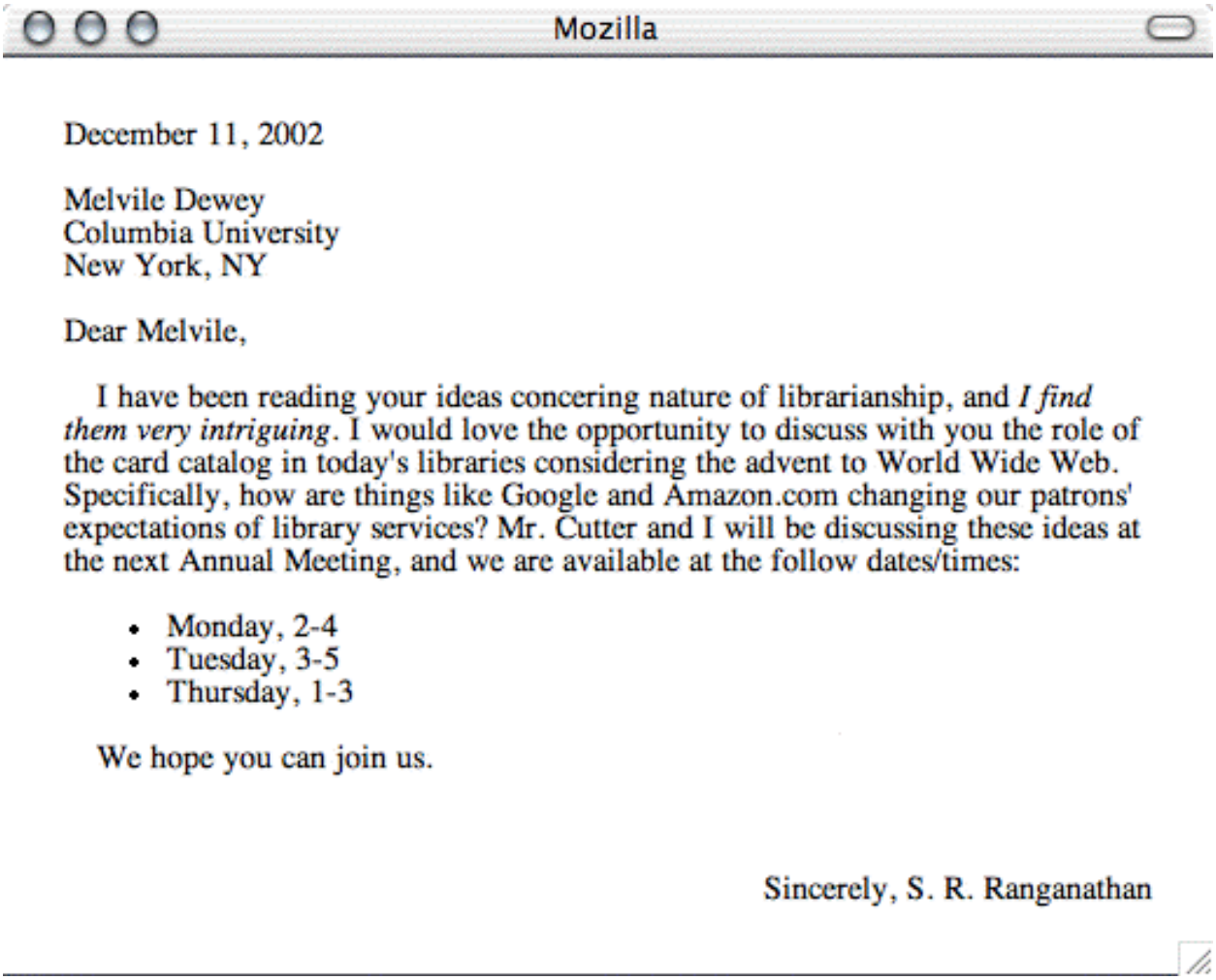
Below is our first XSLT example. Designed to be applied against the file named letter.xml, it will output a valid XHTML file. You can see this in action by using an XSLT processor named xsltproc. Assuming all the necessary files exist in the same directory, the xstlproc command is **xsltproc -o letter.html letter2html.xsl letter.xml** .

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

 <!-- letter2html.xsl; an XSL file -->

 <!-- define the output as an XML file, specficially, an XHTML file -->
 <xsl:output
  method="xml"
  omit-xml-declaration="no"
  indent="yes"
  doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
  doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" />

 <!-- start at the root of the file, letter -->
 <xsl:template match="letter">

  <!-- output an XHTML root element -->
  <html>

   <!-- open the XHTML's head element -->
   <head>

    <!-- output a title element with the addressee's name -->
    <title><xsl:value-of select="addressee/name"/></title>

   <!-- close the head element -->
   </head>

   <!-- open the body tag and give it some style -->
   <body style="margin: 5%">
```

```
   <!-- find various templates in the XSLT file with their
        associated values -->
   <xsl:apply-templates select="date"/>
   <xsl:apply-templates select="addressee"/>
   <xsl:apply-templates select="greeting"/>
   <xsl:apply-templates select="paragraph | list" />
   <xsl:apply-templates select="closing"/>

  <!-- close the body tag -->
  </body>

 <!-- close the XHTML file -->
 </html>

</xsl:template>

<!-- date -->
<xsl:template match="date">

 <!-- output a paragraph tag and the content of the current
      node, date -->
 <p><xsl:apply-templates/></p>

</xsl:template>

<!-- addressee -->
<xsl:template match="addressee">

 <!-- open a paragraph -->
 <p>

  <!-- output the content of letter.xml's name, address_one,
       and address_two elements, as well a couple br tags -->
  <xsl:value-of select="name"/><br />
  <xsl:value-of select="address_one"/><br />
  <xsl:value-of select="address_two"/>

 <!-- close the paragraph -->
 </p>

</xsl:template>

<!-- each of the following templates operate exactly like the
     date template -->

<!-- greeting -->
<xsl:template match="greeting">
 <p>
  <xsl:apply-templates/>
 </p>
</xsl:template>

<!-- paragraph -->
<xsl:template match="paragraph">
 <p style="text-indent: 1em">
  <xsl:apply-templates/>
 </p>
</xsl:template>

<!-- closing -->
<xsl:template match="closing">
 <p style="margin-top: 3em; text-align: right">
  <xsl:apply-templates/>
 </p>
</xsl:template>

<!-- italics -->
<xsl:template match='italics'>
 <i>
  <xsl:apply-templates/>
```

```
  </i>
 </xsl:template>

 <!-- list -->
 <xsl:template match='list'>
  <ul>
   <xsl:apply-templates/>
  </ul>
 </xsl:template>

 <!-- item -->
 <xsl:template match='item'>
  <li>
   <xsl:apply-templates/>
  </li>
 </xsl:template>

</xsl:stylesheet>
```

The end result should look something like this:



Admittedly, the example above looks rather complicated and truthfully functions exactly like our CSS files. At the same time, displaying the letter.xml file with CSS requires a modern browser. If the letter2html.xsl file were incorporated into a Web server, then Web browser's would not need to understand CSS. Given the example above, there

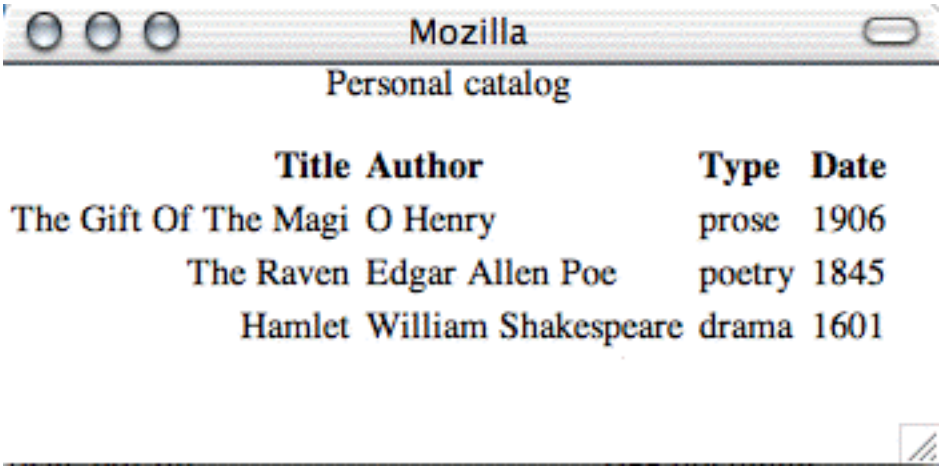
is not a compelling reason to use XSLT, yet.

Displaying tabular data

Here is an other example of an XSLT file used to render an XML file. This example renders our catalog.xml file. It too functions very much like a plain o' CSS file. You can transform it using xsltproc like this: **xsltproc -o catalog.html catalog2html.xsl catalog.xml** .

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

 <!-- catalog2html.xsl -->

 <xsl:output
  method="xml"
  omit-xml-declaration="no"
  indent="yes"
  doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
  doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" />

 <!-- catalog -->
 <xsl:template match="catalog">
  <html>
   <head>
    <title><xsl:value-of select="caption"/></title>
   </head>
   <body>
    <table>
     <xsl:apply-templates select="caption"/>
     <xsl:apply-templates select="structure"/>
     <xsl:apply-templates select="work"/>
    </table>
   </body>
  </html>
 </xsl:template>

 <!-- caption -->
 <xsl:template match="caption">
  <caption style="text-align: center; margin-bottom: 1em">
   <xsl:value-of select="."/>
  </caption>
 </xsl:template>

 <!-- structure -->
 <xsl:template match="structure">
  <thead style="font-weight: bold">
   <tr><xsl:apply-templates/></tr>
  </thead>
 </xsl:template>

 <!-- work -->
 <xsl:template match="work">
  <tr><xsl:apply-templates/></tr>
 </xsl:template>

 <!-- title -->
 <xsl:template match="title">
  <td style="text-align: right; padding: 3px"><xsl:value-of select="."/></td>
 </xsl:template>

 <!-- author, type, or date -->
 <xsl:template match="author | type | date">
  <td><xsl:value-of select="."/></td>
 </xsl:template>

</xsl:stylesheet>
```

Again, the end result should look something like this:



Manipulating XML data

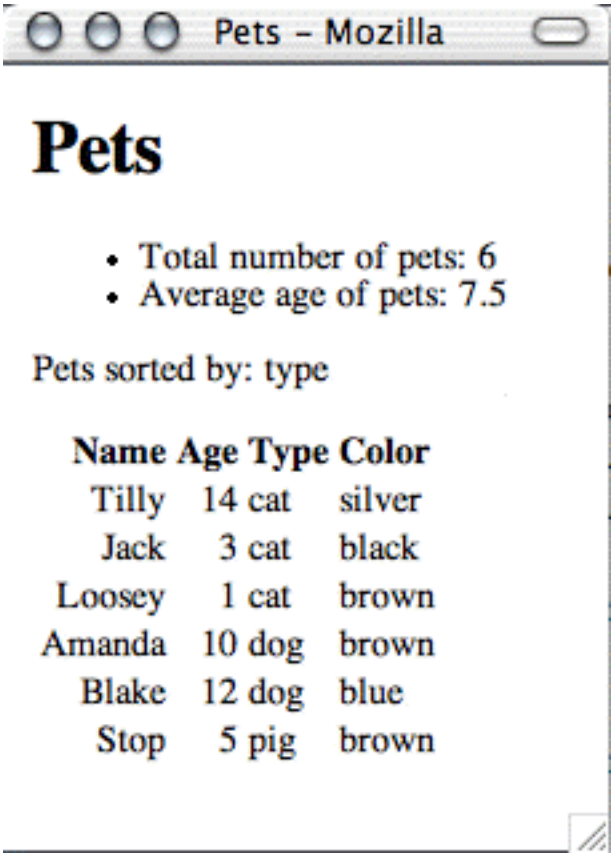
CSS files, just like the XSLT files above, process the XML input from top to bottom. This technique does not take advantage of the programmatic characteristics of XSLT. The next example does. First of all, the next example takes input, namely a value to sort by. Second, this XSLT file takes advantage of a few function calls such as count, sum and sort. Herein lies an important distinction between CSS and XSLT. CSS is intended for display, only. XSLT can be used to display XML content. It can be used to manipulate content as well.

In this example, calculations are done on our list of pets. First of all, a count of the number of pets is displayed as well as their average age. Second, the list of pets can be sorted by their name, age, type, or color. To see this in action, try the following command: **xsltproc -o pets.html --stringparam sortby age pets2html.xsl pets.xml** . Different output can be gotten by changing the sortby value to name, color, or type. What happens if an invalid sortby value is passed to the XSLT file? What happens to the output if no --stringparam values are passed? Why?

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

 <!-- pets2html.xsl -->

 <xsl:output
  method="xml"
  omit-xml-declaration="no"
  indent="yes"
  doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
  doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" />

 <!-- get an input parameter and save it to the variable named
      sortby; use name by default -->
 <xsl:param name="sortby" select="'name'"/>

 <!-- pets -->
 <xsl:template match="pets">

  <html>
   <head>
    <title>Pets</title>
   </head>
   <body style="margin: 5%">
```

```
    <h1>Pets</h1>
    <ul>

     <!-- use the count function to determine the number of pets -->
     <li>Total number of pets: <xsl:value-of select="count(pet)"/></li>

     <!-- calculate the average age of the pets by using the sum
          and count functions, as well as the div operator -->
     <li>Average age of pets: <xsl:value-of select="sum(pet/age) div count(pet)"/></li>
    </ul>
    <p>Pets sorted by: <xsl:value-of select="$sortby"/></p>
    <table>
     <thead>
      <tr>
       <td style="text-align: right; font-weight: bold">Name</td>
       <td style="text-align: right; font-weight: bold">Age</td>
       <td style="font-weight: bold">Type</td>
       <td style="font-weight: bold">Color</td>
      </tr>
     </thead>
     <xsl:apply-templates select="pet">

      <!-- sort the pets by a particular sub element ($sortby); tricky! -->
      <xsl:sort select="*[name()=$sortby]"/>
     </xsl:apply-templates>
    </table>
   </body>
  </html>
 </xsl:template>

 <!-- pet -->
 <xsl:template match="pet">
  <tr><xsl:apply-templates/></tr>
 </xsl:template>

 <!-- name -->
 <xsl:template match="name">
  <td style="text-align: right"><xsl:value-of select="."/></td>
 </xsl:template>

 <!-- age -->
 <xsl:template match="age">
  <td style="text-align: right"><xsl:value-of select="."/></td>
 </xsl:template>

 <!-- type or color -->
 <xsl:template match="type | color">
  <td><xsl:value-of select="."/></td>
 </xsl:template>

</xsl:stylesheet>
```

Here some same output:



Using XSLT to create other types of text files

This final example uses the pets.xml file, again. This time the XSLT file is used to create another type of output, namely a very simple set of SQL statements. The point of this example is to illustrate how the pets.xml file can be repurposed. Once for display, and once for storage. Use this command to see the result: **xsltproc -o pets.sql pets2sql.xsl pets.xml** . What could you do to make the output prettier?

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

 <!-- pets2sql.xsl -->

 <!-- create plain o' text output -->
 <xsl:output method="text" />

 <!-- find each each pet -->
 <xsl:template match="pets">

  <!-- loop through each pet -->
  <xsl:for-each select="pet">

   <!-- output an SQL INSERT statement for the pet -->
   INSERT INTO pets (name, age, type, color)
   WITH VALUES ('<xsl:value-of select="name" />',
   '<xsl:value-of select="age" />',
   '<xsl:value-of select="type" />',
   '<xsl:value-of select="color" />');

  </xsl:for-each>
```

```
 </xsl:template>
</xsl:stylesheet>
```

SQL created by the XSLT file above looks like this:

```
INSERT INTO pets (name, age, type, color) WITH VALUES ('Tilly', '14', 'cat', 'silver');
INSERT INTO pets (name, age, type, color) WITH VALUES ('Amanda', '10', 'dog', 'brown');
INSERT INTO pets (name, age, type, color) WITH VALUES ('Jack', '3', 'cat', 'black');
INSERT INTO pets (name, age, type, color) WITH VALUES ('Blake', '12', 'dog', 'blue');
INSERT INTO pets (name, age, type, color) WITH VALUES ('Loosey', '1', 'cat', 'brown');
INSERT INTO pets (name, age, type, color) WITH VALUES ('Stop', '5', 'pig', 'brown');
```

This file could then be feed to a relational database program that understands SQL and populate a table with data.

This section barely scratched the surface of XSLT. It is an entire programming language unto itself and much of the promise of XML lies in the exploitation of XSLT to generate various types of output be it output for Web browsers, databases, or input for other computer programs.

Chapter 6. Document type definitions

Defining XML vocabularies with DTDs

Creating your own XML mark up is all well and good, but if you want to share your documents with other people you will need to communicate to these other people the vocabulary your XML documents understand. This is the semantic part of XML documents -- what elements do your XML files contain and how are the elements related to each other? These semantic relationships are created using Document Type Definitions (DTD) and/or XML Schemas. DTDs are legacy implementations from the SGML world. They are more commonly used than the newer, XML-based, XML Schemas. This section provides an overview for creating DTDs.

DTDs can exist inside an XML document or outside an XML document. If they reside in an XML document, then they begin with a DOCTYPE declaration followed by the name of the XML document's root element and finally a list of all the elements and how they are related to each other. Here is a simple DTD for embedded in the pets.xml file itself:

```
<!DOCTYPE pets [
 <!ELEMENT pets  (pet+)>
 <!ELEMENT pet   (name, age, type, color)>
 <!ELEMENT name  (#PCDATA)>
 <!ELEMENT age   (#PCDATA)>
 <!ELEMENT type  (#PCDATA)>
 <!ELEMENT color (#PCDATA)>
]>

<pets>
 <pet>
  <name>Tilly</name>
  <age>14</age>
  <type>cat</type>
  <color>silver</color>
 </pet>
 <pet>
  <name>Amanda</name>
  <age>10</age>
  <type>dog</type>
  <color>brown</color>
 </pet>
 <pet>
  <name>Jack</name>
  <age>3</age>
  <type>cat</type>
  <color>black</color>
 </pet>
 <pet>
```

```
  <name>Blake</name>
  <age>12</age>
  <type>dog</type>
  <color>blue</color>
 </pet>
 <pet>
  <name>Loosey</name>
  <age>1</age>
  <type>cat</type>
  <color>brown</color>
 </pet>
 <pet>
  <name>Stop</name>
  <age>5</age>
  <type>pig</type>
  <color>brown</color>
 </pet>
</pets>
```

More commonly, DTDs reside outside an XML document since they are intended to be used by many XML files. In this case, the DOCTYPE declaration includes a pointer to a file where the XML elements are described.

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
 "http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
```

Whether or not the DTD is internal or external, a list of XML elements needs to be articulated. Each item on the list will look something like !ELEMENT pets (pet+) where "!ELEMENT" denotes an element, "pets" is the element being defined, and "(pet+)" is the definition. The definitions are the difficult part. There are many different types of values the definitions can include, and only a few of them are described here.

Names and numbers of elements

First of all, the definitions can include the names of other elements. In our example above, the first declaration defines and element called pets and it is allowed to include just on other element, pet. Similarly, the element defined as pet is allowed to contain four other elements: name, age, type, and color. Each element is qualified by how many times it can occur in the XML document. This is done with the asterisk (*), question mark (?), and plus sign (+) symbols. Each of these symbols have a specific meaning:

- asterisk (*) - The element may appear zero or more times
- question mark (?) - The element may appear zero or one time, only
- plus sign (+) - The element appears at least once if not more times

If an element is not qualified with one of these symbols, then the element can appear once and only once. Consequently, in the example above, since pets is defined to contain the element pet, and the pet element is qualified with a plus sign, there must be at least one pet element within the pets element.

PCDATA

There is another value for element definitions you need to know, #PCDATA. This stands for parsed character data, and it is used to denote content that contains only text, text without markup.

Sequences

Finally, it is entirely possible that an element will contain multiple, sub elements. When strung together, this list of multiple elements is called a sequence, and they can be grouped together in the following ways:

- comma (,) is used to denote the expected order of the elements in the XML file
- parentheses (()) are used to group elements together
- vertical bar (|) is used to denote a Boolean union relationship between the elements.

Putting it all together

Walking through the DTD for pets.xml we see that:

1. The root element of the document should is pets.
2. The root element, pets, contains at least one pet element.
3. Each pet element can contain one and only one name, age, type, and color element, in that order.
4. The elements name, age, type, and color are to contain plain text, no mark up.

Below is a DTD for the letter in a previous example.

```
<!ELEMENT letter       (date, addressee, greeting, (paragraph+ | list+)*, closing)>
<!ELEMENT date         (#PCDATA)>
<!ELEMENT addressee    (name, address_one, address_two)>
<!ELEMENT name         (#PCDATA)>
<!ELEMENT address_one  (#PCDATA)>
<!ELEMENT address_two  (#PCDATA)>
<!ELEMENT greeting     (#PCDATA)>
<!ELEMENT paragraph    (#PCDATA | italics)*>
<!ELEMENT italics      (#PCDATA)>
<!ELEMENT list         (item+)>
<!ELEMENT item         (#PCDATA)>
<!ELEMENT closing      (#PCDATA)>
```

This example is a bit more complicated. Walking through it we see that:

1. The letter element contains one date element, one adressee element, one greeting element, at least one paragraph or at least one list element, and one closing element.
2. The date element contains plain o' text, no markup.
3. The addressee element contains one and only one name, address_one, and address_two element, in that order.
4. The name, address_one, address_two, and greeting elements contain text, no markup.
5. The paragraph element can contain plain text or the italics element.
6. The italics element contains plain, non-marked up, text.
7. The list element contains at least one item element.

8. The item and closing elements contain plain text.

To include this DTD in our XML file, we must create pointer to the DTD, and since the DTD is local to our environment, and not a standard, the pointer should be included in the XML document looking like this:

```
<!DOCTYPE letter SYSTEM "letter.dtd">
<letter>
  <date>
   December 11, 2002
  </date>
  <addressee>
   <name>
    Melvile Dewey
   </name>
   <address_one>
    Columbia University
   </address_one>
   <address_two>
    New York, NY
   </address_two>
  </addressee>
  <greeting>
    Dear Melvile,
  </greeting>
  <paragraph>
   I have been reading your ideas concerning nature of librarianship,
   and <italics>I find them very intriguing</italics>. I would love
   the opportunity to discuss with you the role of the card catalog
   in today's libraries considering the advent to World Wide Web.
   Specifically, how are things like Google and Amazon.com changing
   our patrons' expectations of library services? Mr. Cutter and I
   will be discussing these ideas at the next Annual Meeting, and we
   are available at the follow dates/times:
  </paragraph>
  <list>
 <item>
  Monday, 2-4
 </item>
 <item>
  Tuesday, 3-5
 </item>
 <item>
  Thursday, 1-3
 </item>
  </list>
  <paragraph>
    We hope you can join us.
  </paragraph>
  <closing>
    Sincerely, S. R. Ranganathan
  </closing>
</letter>
```

By feeding this XML to an XML processor, the XML processor should know that the element named letter is the root of the XML file, and the XML file can be validated using a local, non-standardized DTD file named letter.dtd.

Exercise - Writing a simple DTD

In this exercise your knowledge of DTDs will be sharpened by examining an existing DTD, and then you will write your own DTD.

1. Consider the DTD describing the content of the catalog.xml file, below, and on the back of this paper write the answers the following questions:

```
<!ELEMENT catalog    (caption, structure, work+)>
<!ELEMENT caption    (#PCDATA)>
<!ELEMENT structure  (title, author, type, date)>
<!ELEMENT work       (title, author, type, date)>
<!ELEMENT title      (#PCDATA)>
<!ELEMENT author     (#PCDATA)>
<!ELEMENT type       (#PCDATA)>
<!ELEMENT date       (#PCDATA)>
```

   A. How many elements can the catalog element contain, and what are they?

   B. How many works can any one catalog.xml file contain?

   C. Can marked up text be included in the title element? Explain why or why not.

   D. If this DTD is intended to be a locally developed DTD, and intended to be accessed from outside the XML document, how would you write the DTD declaration appearing in the XML file?

2. Create an internal DTD for the file ala.xml, and validate the resulting XML file.

   A. Open ala.xml in NotePad.

   B. Add an internal document type declaration to the top of the file, <!DOCTYPE letter [ ]>.

   C. Between the square brackets ([]), enter the beginings of an element declaration for each element needed to be defined (i.e. letter, date, address, greeting, etc.). For example, type <!ELEMENT para ()> for the paragraph element between the square brackets.

   D. For each element define its content by listing either other element names or #PCDATA, depending on how the XML file is structured. Don't forget to append either a plus sign (+), an asterisk (*), or a question mark (?) to denote the number of times an element or list of elements may appear in the XML file.

   E. Save ala.xml.

   F. Select and copy the entire contents of ala.xml to the clipboard.

   G. Open your Web browser, and validate your XML file by using a validation form [http://www.stg.brown.edu/service/xmlvalid/] .

Part II. Introductions to specific XML vocabularies

Chapter 7. XHTML

Introduction

XHTML a pure XML implementation of HTML. Therefore the six rules of XML syntax apply: there is one root element, element names are case-sensitive (lower-case), elements must be closed, elements must be correctly nested, attributes must be quoted, and the special characters must be encoded as entities. There are a few XHTML DTDs ranging from a very strict version allowing no stylistic tags or tables to a much more lenient version where such things are simply not encouraged. XHTML documents require a DOCTYPE declaration at the beginning of the file in order to specify what version of XHTML follows. So, the following things apply:

- Your document must have one and only one html element.
- All elements are to be lower-case
- Empty elements such as hr and br must be closed as in <hr /> and
- Attributes must be quoted as in
- You can not improperly next elements
- The <, >, and & characters must be encoded as entities

XHTML has four of "common" attributes, attributes common to any XHTML element. These attributes are:

1. id - used to identify a unique location in a file
2. title - used to give a human-readable label to an element
3. style - a place older for CSS style information
4. class - used to give an element label usually used for CSS purposes

By liberally using these common attributes and assigning them meaningful values it is possible to completely separate content from presentation and at the same time create accessible documents, documents that should be readable

by all types of people as well as computers.

Stylistic elements are discouraged in an effort to further separate content from presentation. When stylizing is necessary you are encouraged to make liberal use of CSS. Your CSS specification can reside in either an external file, embedded in the head of the XHTML document, or specified within each XHTML element using the style attribute.

Tables are a part of XHTML, and they are intended to be used to display tabular data. Using tables for layout is discouraged. Instead, by using the div and span element, in combination with CSS file, blocks of text within XHTML documents can be positioned on the screen.

Below is a simple XHTML file and CSS file representing a home page. Graphic design is handled by the CSS file, and even when the CSS file is not used the display is not really that bad.

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
 <title>A simple home page</title>
 <link rel="stylesheet" href="home.css" type="text/css" />
</head>
<body>
<div class="menu">
 <h3 class="label">Links</h3>
 <a href="http://infomotions.com/travel/" title="Travel
 logs">Travel logs</a>
 <br />
 <a href="http://infomotions.com/musings/" title="Musings on
 Information">Musings on Information</a>
 <br />
 <a href="http://infomotions.com/" title="Infomotions home
 page">About us</a>
 <br />
</div>
<div class="content">
 <h1>A simple home page</h1>
 <p>
         This is a simple home page illustrating the use of
         XHTMLversion 1.0.
 </p>
 <p>
         <img src="water.jpg" alt="Atlantic Ocean" width="120"
         height="160" align="right" hspace="5" vspace="3" />XHTML is
         not a whole lot different from HTML. It includes all of the
         usual tags such as the anchor tag for hypertext references
         (links) and images. Tables are still a part of the
         specification, but they are not necessarily intended for
         formatting purposes.
 </p>
 <p>
         The transition from HTML to XHTML is simple as long as you
         keep in mind a number of things. First, make sure you take
         into account the six rules for XML syntax. Second, shy away
         from using stylistic tags (elements) such as bold, italics,
         and especially font. Third, make liberal use of the div and
         span elements. When used in conjunction with CSS files, you
         will be able to easily position and format entire blocks of
         text on the screen.
 </p>
 <hr />
 <p class="footer">
         Author: Eric Lease Morgan &lt;<a
         href="mailto:eric_morgan@infomotions.com">eric_morgan@
         infomotions.com</a>&gt;
         <br />
         Date: 2003-01-19
```

```
         <br />
         URL: <a href="./home.html">./home.html</a>
 </p>

</div>
</body>
</html>
```

```
h1, h2, h3, h4, h5, h6 {
  font-family: helvetica, sans-serif;
}

p {
 font-family: times, serif;
 font-size: large;
}

p.footer {
 font-size: small;
}

div.menu {
 position: absolute;
 margin-right: 82%;
 text-align: right;
 font-size: small;
}

div.content {
 position: absolute;
 margin-left: 22%;
 margin-right: 3%;;
}

a:hover {
 color: red;
 background-color: yellow;
}
```

Rendering these files in your CSS-aware Web browers should display something ike this:

Exercise - Writing an XHTML document

In this exercise you will experiment marking up a document using a standard DTD, specifically, XHTML.

1. Mark up ala.txt as an XHTML document.

    A. Open ala.txt in NotePad.

    B. Save the file as ala.html.

    C. Add the XML declaration to the top of file file, <?xml version="1.0"?>.

    D. Add the document type declaration, <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

    E. Add the opening and closing html elements (i.e. <html>).

    F. Add an opening and closing head and body sections (i.e. <head>, <body>).

    G. Add a title element to the head section (i.e. <title>).

    H. Add a link element pointing to the location where your CSS file will be (i.e. <link rel="stylesheet" href="ala.css" type="text/css" />).

    I. Using only the p and br elements, mark up the body of the letter making sure to properly open and close each element (i.e. <p> ,
).

    J. Save your file, again, and view it in your Web browser.

K. Create a new empty file in NotePad, and save it as ala.css.

L. Add only two selectors to the CSS file, each for a different implementation of the p element (i.e. p { font-family: times, serif; font-size: large } and p.salutation { text-align: right }).

M. Add the common attribute "class" to the last paragraph of your letter giving it a value of salutation (i.e. <p class="salutation">).

N. Save your file, again, and view it in your Web browser.

Chapter 8. TEI

Introduction

TEI, the Text Encoding Initiative, is a grand daddy of markup languages. Starting its life as SGML and relatively recently becoming XML compliant, TEI is most often used by the humanities community to mark up literary works such as poems and prose. Many times these communities digitally scan original documents, convert the documents into text using optical character recognition techniques, correct the errors, and mark up the resulting text in TEI. Ideally a scholar would have on hand an original copy of a book or manuscript along side a digital version in TEI. Using these two things in combination the scholar would be able to very thoroughly analyse the text and create new knowledge.

The TEI DTD is very rich and verbose. It contains elements for every literary figure (paragraphs, stanza, chapters, footnotes, etc.). Since TEI documents are, for the most part, intended to replicate as closely as possible original documents, the DTD contains markup to denote the location of things like page breaks and line numbers in the original text. There is markup for cross references and hyperlinks. There is even markup for editorial commentary, interpretation, and analysis. The DTD so verbose that some TEI experts suggest using only parts of the DTD. In practice, many institutions using the TEI DTD use what is commonly called TEILite, a pared down version of the DTD containing the definitions of elements of use to most people.

A few elements

Providing anything more than the briefest of TEI introductions is beyond the scope of this text. This section outlines the most minimal of TEI elements and how TEI documents can be processed using XML tools.

The simplest of TEI documents contains header (teiHeader) and text sections. The teiHeader section contains a number of sub elements used to provide meta data about the document. The text section is further divided into three more sections (front, body, and back). Here is a list of the major TEI elements and brief descriptions of each:

- TEI.2 - the root element of a TEILite document
- teiHeader - a container for the meta data of a TEI document
- fileDesc - a mandatory sub element of teiHeader describing the TEI file
- titleStmt - a place holder for the author and title of a work

- title - the title of the document being encoded
- author - the author of the document being encoded
- publicationStmt - free text describing how the document is being published
- sourceDesc - a statement describing where the text was derived
- text - the element where the text of the document is stored
- front - denotes the front matter of a book or manuscript
- body - the meat of the matter, the book or manuscript itself
- back - the back matter of a book or manuscript
- date - a date
- p - a paragraph
- lg - a line group intended for things like poems
- l - a line in a line group

Using the elements above is it possible to create a perfectly valid, but rather brain-dead, TEI document, such as the following:

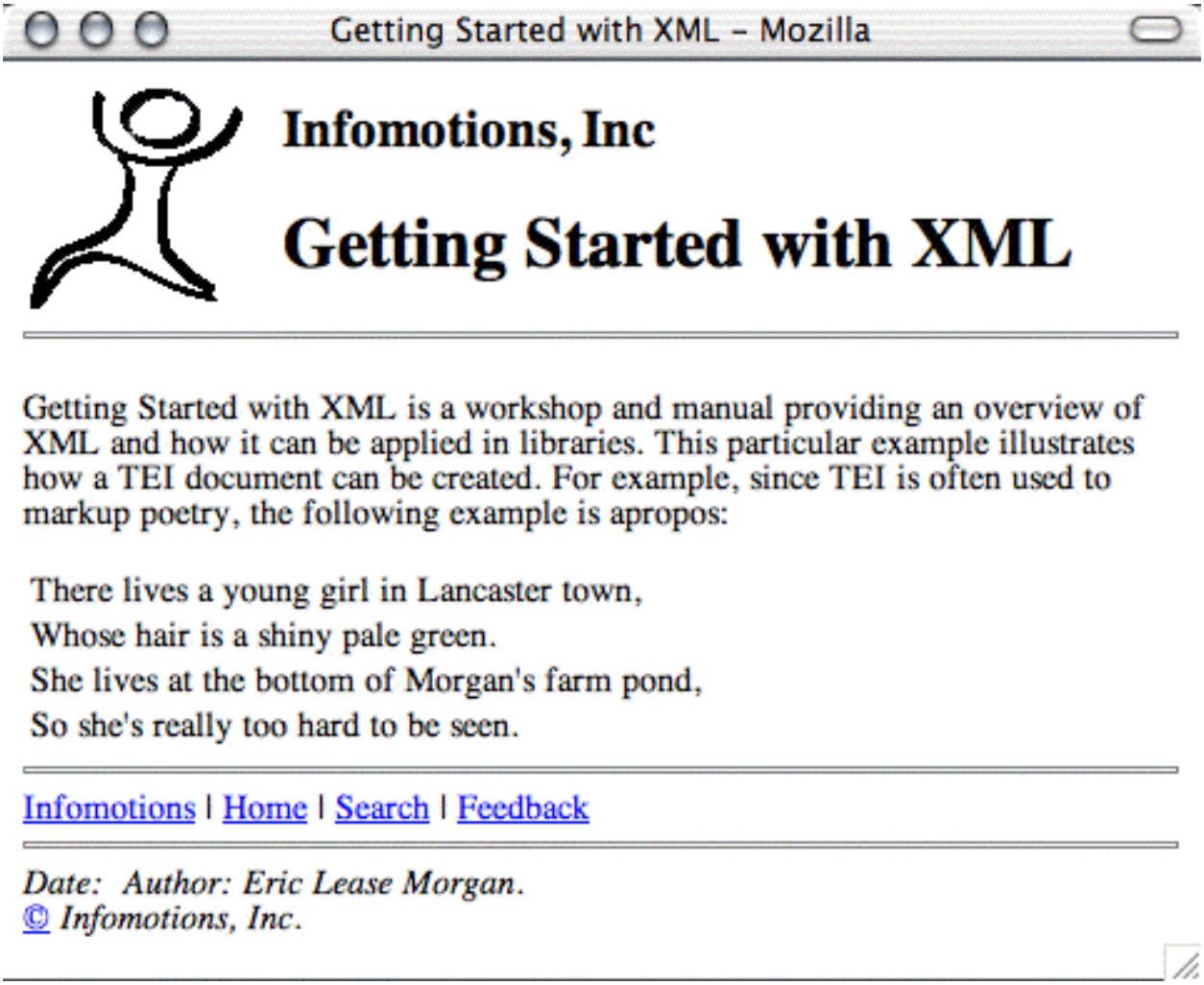
```
<TEI.2>

 <teiHeader>
  <fileDesc>
   <titleStmt>
    <title>Getting Started with XML</title>
    <author>Eric Lease Morgan</author>
   </titleStmt>
   <publicationStmt><p>Originally published in <date>March
   2003</date> for an Infopeople workshop.</p></publicationStmt>
   <sourceDesc><p>There was no original source; this document was
   born digital.</p></sourceDesc>
  </fileDesc>
 </teiHeader>

 <text>

  <front></front>

  <body>

   <p>
   Getting Started with XML is a workshop and manual providing an
   overview of XML and how it can be applied in libraries. This
   particular example illustrates how a TEI document can be
   created. For example, since TEI is often used to markup poetry,
   the following example is apropos:
   </p>

   <lg>
        <l>There lives a young girl in Lancaster town,</l>
        <l>Whose hair is a shiny pale green.</l>
        <l>She lives at the bottom of Morgan's farm pond,</l>
        <l>So she's really too hard to be seen.</l>
   </lg>
```

```
  </body>
  <back></back>
 </text>
</TEI.2>
```

TEI files have historically been created and then indexed/rendered with a set of middleware programs such as XPAT. With the increasing availability of XML technologies such as CSS and XSLT, a number of stylesheets have become available. The official TEI website offers links a few of these things, and the XSLT stylesheets work quite well.

Assuming you were to save the TEI text above with the filename getting.xml, you would be able to create a very nice XHTML document using the XSLT stylesheets and xstlproc like this: **xsltproc tei-stylesheets/teihtml.xsl tei-getting.xml** .

Here some same output:



Be forewarned. The XSL stylesheet is configured in such a way to always save documents with the file name index.html even if you specify the -o option. You will have to edit the file named teixsl-html/teihtml-param.xsl to season your XHTML output to taste.

Exercise

In this exercise you will render a TEI file using CSS and XSLT.

1. Render a TEI file using CSS

    A. Open the file named tei.xml in NoteTab.

    B. Add the following XML processing instruction to the top of the file: <?xml-stylesheet type='text/css' href='tei-dancer.css'?>

    C. Save the file.

    D. Open tei.xml in your Web browser. Enjoy.

    E. Change the value of href in tei.xml's XML processing instruction to tei-oucs.css.

    F. Save the file.

    G. Open tei.xml in your Web browser. Again, enjoy.

2. Transform the TEI file into XHTML

    A. Run the following command: **xsltproc teixsl-html/teihtml.xsl tei.xml**

    B. Open the resulting index.html file in your Web browser. Interesting.

    C. Edit the file named tei-stylesheets/teihtml-param.xsl and change the value of institution from Oxford University Computing Services to your name.

    D. Save teixsl-html/teihtml-param.xsl.

    E. Repeat Steps #1 and #2. Cool!

Chapter 9. EAD

Introduction

EAD stands for Encoded Archival Description, and it is an SGML/XML vocabulary used to markup archival finding aids. Like TEI it has its roots deep in SGML, and like TEI has only recently become XML compliant.

As you may or may not know, finding aids are formal descriptions of things usually found in institutional archives. These things are not limited to manuscripts, notes, letters, and published and published works of individuals or groups but increasingly include computer programs and data, film, video, sound recordings, and realia. Because of the volume of individual materials in these archives, items in the archives are usually not described individually but as collections. Furthermore, items are usually not organized by subject but more likely by date since the chronological order things were created embodies the development of the collections' ideas. Because of these characteristics, items in archives are usually not described using MARC and increasingly described using EAD.

According to the EAD DTD, there are only a few elements necessary to create a valid EAD document, but creating an EAD document with just these elements would not constitute a very good finding aid. Consequently, the EAD Application Guidelines suggest the following elements:

- ead - the root of an EAD document
- eadheader - a container for meta data about the EAD document
- eadid - a unique code for the EAD document
- filedesc - a container for the bibliographic description of the EAD document
- titlestmt - a container for things like author and title of the EAD document
- titleproper - the title of the EAD document
- author - the names of individuals or group who created the EAD document
- publicationstmt - a container for publication information
- publisher - the name of the party distributing the EAD document
- date - a date
- profiledecs - a container bundling information about the EAD encoding procedure

- creation - a place to put the names of persons or places about the EAD encoding procedure
- langusage - a list of the languages represented in the EAD document
- language - an element denoting a specific language
- archdesc - a container for the bulk of an EAD finding aid; the place were an archival item is described
- did - a container for an individual descriptive unit
- repository - the institution or agency responsible for providing the intellectual access to the material
- corpname - a name identify a corporate identity
- origination - the name of a person or group responsible for the assembly of the materials in the collection
- persname - a personal name
- famname - a family name
- unittitle - a title of described materials
- unitdate - the creation year, month, and day of the described materials
- physdesc - a container for describing the physical characteristics of a collection
- unitid - a unique reference number -- control number -- for the described material
- abstract - a narrative summary describing the materials
- bioghist - a concise history or chronology placing the materials in context
- scopecontent - a summary describing the range of topic covered in the materials
- controlaccess - a container used to denote controlled vocabulary terms in other collections or indexes
- dsc - a container bundling hierarchical groups of other sub-items in the collection
- c - a container describing logical section of the described material
- container - usually an integer used in conjunction with a number of attributes denoting the physical extent of the described materials

Whew!

Example

Here is an EAD file. Can you figure out what it is?

```
<ead>
 <eadheader>
  <eadid>
   ELM001
  </eadid>
  <filedesc>
   <titlestmt>
    <titleproper>
     The Eric Lease Morgan Collection
```

```
    </titleproper>
    <author>
     Created by Eric Lease Morgan
    </author>
   </titlestmt>
   <publicationstmt>
    <publisher>
     Infomotions, Inc.
    </publisher>
    <date>
     20030218
    </date>
   </publicationstmt>
   <profiledesc>
    <creation>
     This file was created using a plain text editor.
    </creation>
    <langusage>
     This file contains only one language,
     <language>
      English
     </language>
     .
    </langusage>
   </profiledesc>
  </filedesc>
 </eadheader>
 <archdesc level='otherlevel'>
  <did>
   <repository>
    <corpname>
     Infomotions, Inc.
    </corpname>
   </repository>
   <origination>
    <persname>
     Eric Lease Morgan
    </persname>
   </origination>
   <unittitle>
    Papers
   </unittitle>
   <unitdate>
    1980-2001
   </unitdate>
   <physdesc>
    A collection of four boxes of mostly 8.5 x 11 inch pieces of
    paper kept in my garage.
   </physdesc>
   <unitid>
    B0001
   </unitid>
   <abstract>
    Over the years I have kept various things I have written.
    This collection includes many of those papers. I'm sure they
    will add the body of knowledge when I'm gone.
   </abstract>
  </did>
  <biohist>
   <p>
    Eric was born in Lancaster, PA. He went to college in
    Bethany, WV. He lived in Charlotte and Raleigh, NC for
    fifteen years. He now lives in South Bend, IN.
   </p>
  </biohist>
  <scopecontent>
   <p>
    This collection consists of prepublished works, photographs,
    drawings, and significant email messages kept over the years.
   </p>
  </scopecontent>
```

```
  <controlaccess>
   <p>
    It is unlikely there are any controlled vocabulary terms in
    other systems where similar materials can be located.
   </p>
  </controlaccess>
  <dsc type='othertype'>
   <c level='otherlevel'>
    <did>
     <container type='box'>
      1
     </container>
     <unittitle>
      Box 1
     </unittitle>
     <unitdate>
      1980-1984
     </unitdate>
    </did>
    <did>
     <container type='box'>
      1
     </container>
     <unittitle>
      Box 2
     </unittitle>
     <unitdate>
      1985-1995
     </unitdate>
    </did>
    <did>
     <container type='box'>
      1
     </container>
     <unittitle>
      Box 3
     </unittitle>
     <unitdate>
      1995-1998
     </unitdate>
    </did>
    <did>
     <container type='box'>
      1
     </container>
     <unittitle>
      Box 4
     </unittitle>
     <unitdate>
      1999-2001
     </unitdate>
    </did>
   </c>
  </dsc>
 </archdesc>
</ead>
```

Chapter 10. DocBook

Introduction

DocBook is a DTD designed for marking up computer-related documents. The DTD has been evolving for more than a decade and its roots stem from the O'Reilly publishers, the publishers of many computer-related manuals. Like XHTML and TEI, DocBook is intended to mark up narrative texts, but DocBook includes a number of elements specific to its computer-related theme. Some of these elements include things like screenshot, programlisting, and command.

There are a wide range of basic documents types in DocBook. The most commonly used are book and article. Book documents can contain things like a preface, chapters, and appendices. These things can be further subdivided into sections containing paragraphs, lists, figures, examples, and program listings. (This manual/workbook has been marked up as a DocBook book file.) Articles are very much like books but don't contain chapters.

In order to create a valid DocBook file, the file must contain a document type declaration identifying the version of DocBook being employed. Since the DocBook DTD is evolving, there are many different declarations. Here is a declaration for an article using version 4.2 of the DTD:

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
 "http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
```

Here is a listing of some of the more commonly used elements in a DocBook article:

- article - the root element
- articleinfo - a container used to contain meta data about the article
- author - a container for creator information
- firstname - the first name of an author
- surname - the last name of an author
- email - an email address

- pubdate - the date when the article is/was published
- abstract - a narrative description of the article
- title - an element used often throughout the article and book types assigning headings to containers
- such as articles, books, sections, examples, figures, etc.
- section - a generic part of a book or article
- para - a paragraph
- command - used to denote a computer command, usually entered from at the command line
- figure - a container usually an image
- graphic - the place holder for an image
- ulink - a hypertext reference
- programlisting - a whole or part of a computer program
- orderedlist - a numbered for lettered list
- itemizedlist - a bulleted list
- listitem - an item in either an orderedlist or a itemized list

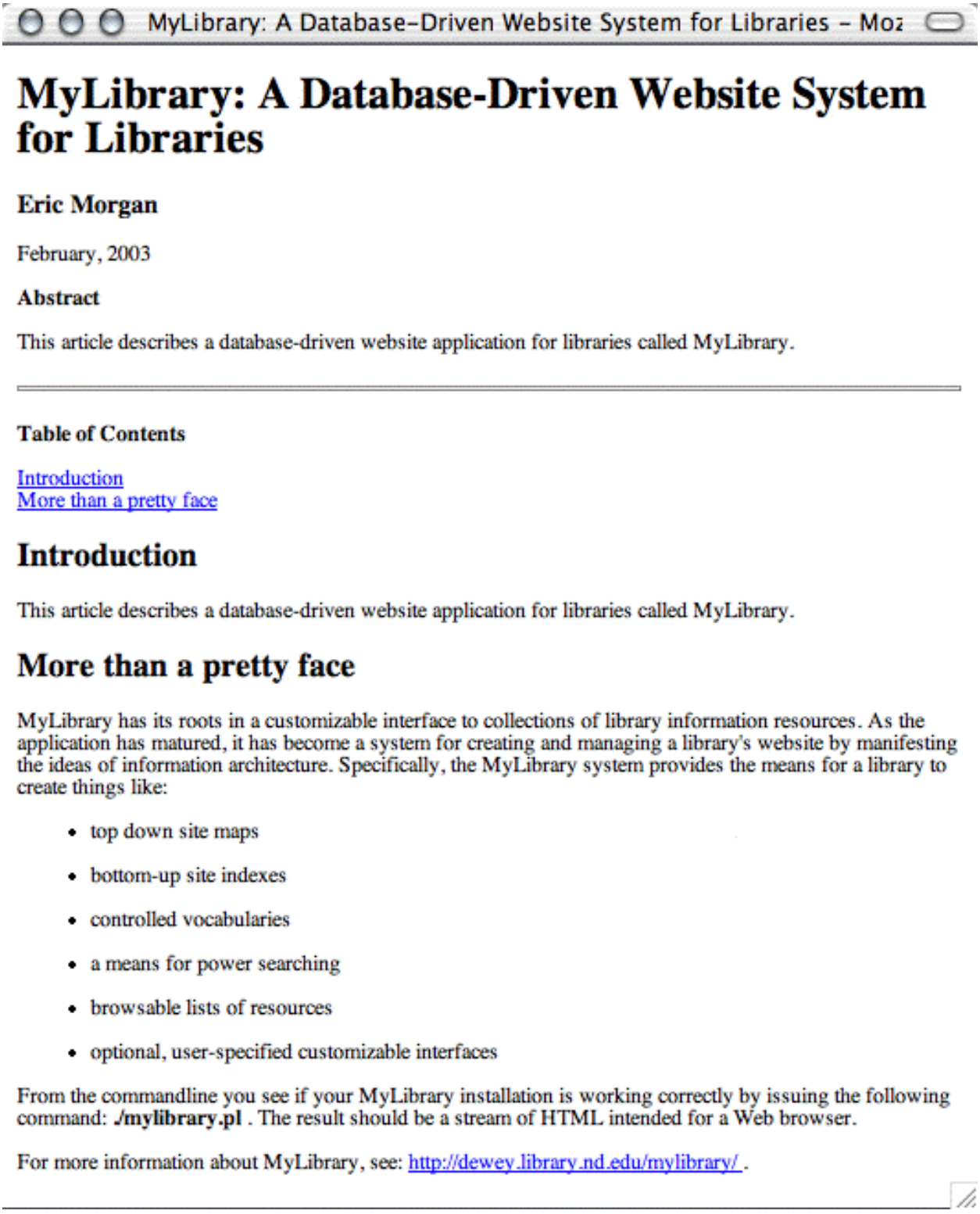
Using some of the tags above, the following example DocBook article was created.

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
  "http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
<article>
  <articleinfo>
    <author>
      <firstname>
        Eric
      </firstname>
      <surname>
        Morgan
      </surname>
    </author>
    <title>MyLibrary: A Database-Driven Website System for Libraries</title>
    <pubdate>
      February, 2003
    </pubdate>
    <abstract>
      <para>
        This article describes a database-driven website
        application for libraries called MyLibrary.
      </para>
    </abstract>
  </articleinfo>
  <section>
    <title>Introduction</title>
    <para>
      This article describes a database-driven website
      application for libraries called MyLibrary.
    </para>
  </section>
  <section>
    <title>More than a pretty face</title>
    <para>
```

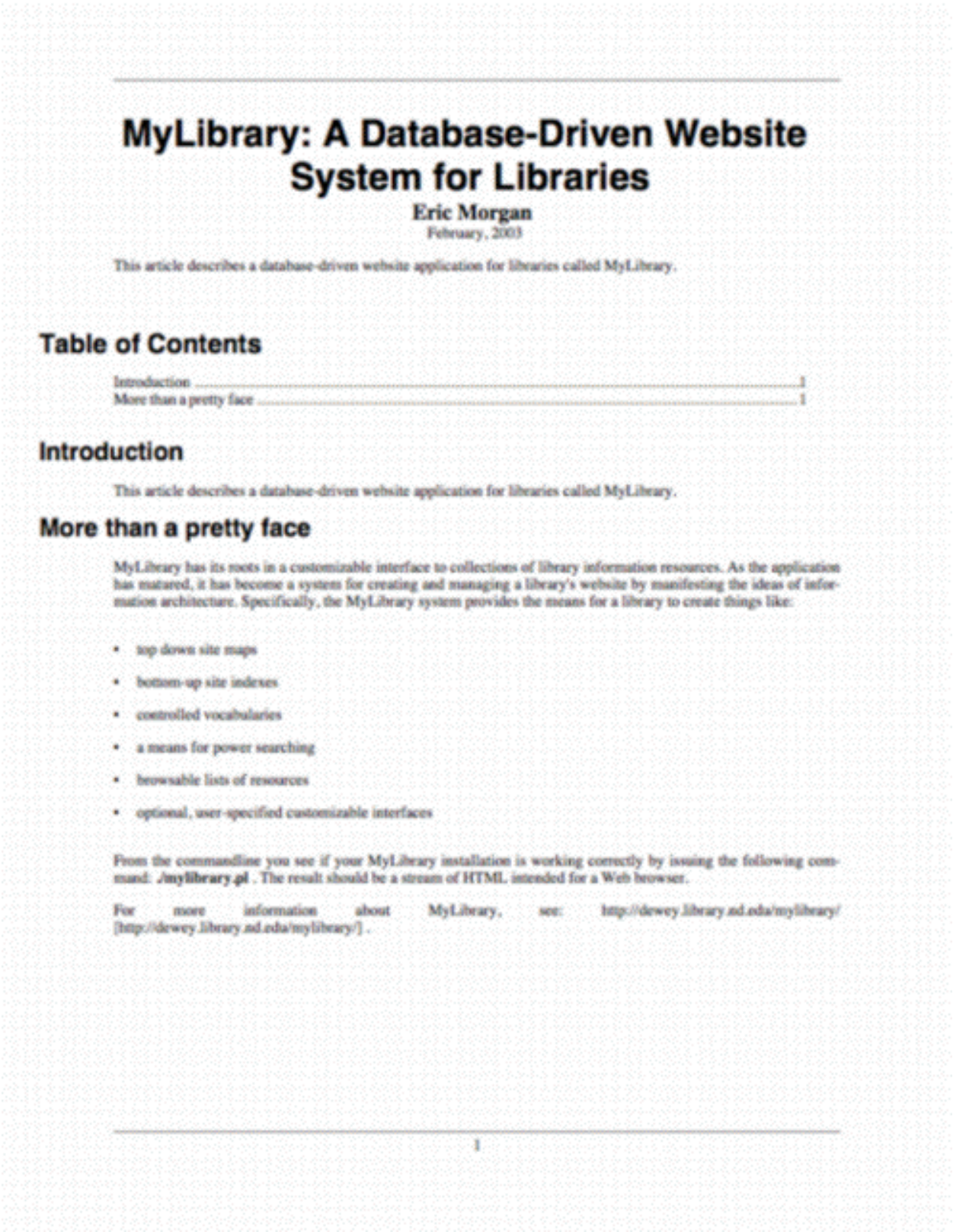
```
      MyLibrary has its roots in a customizable interface to
      collections of library information resources. As the
      application has matured, it has become a system for
      creating and managing a library's website by manifesting
      the ideas of information architecture. Specifically, the
      MyLibrary system provides the means for a library to create
      things like:
    </para>
    <itemizedlist>
      <listitem>
        <para>
          top down site maps
        </para>
      </listitem>
      <listitem>
        <para>
          bottom-up site indexes
        </para>
      </listitem>
      <listitem>
        <para>
          controlled vocabularies
        </para>
      </listitem>
      <listitem>
        <para>
          a means for power searching
        </para>
      </listitem>
      <listitem>
        <para>
          browsable lists of resources
        </para>
      </listitem>
      <listitem>
        <para>
          optional, user-specified customizable interfaces
        </para>
      </listitem>
    </itemizedlist>
    <para>
      From the command line you see if your MyLibrary installation
      is working correctly by issuing the following command:
      <command>
        ./mylibrary.pl
      </command>
      . The result should be a stream of HTML intended for a Web browser.
    </para>
    <para>
      For more information about MyLibrary, see:
      <ulink url="http://dewey.library.nd.edu/mylibrary/">
        http://dewey.library.nd.edu/mylibrary/
      </ulink>
      .
    </para>
  </section>
</article>
```

Processing with XSLT

One of the very nice things about DocBook is its support. There are various XSLT stylesheets available for DocBook allowing you to use your favorite XSLT processor to transform your DocBook files into things like XHTML, HTML, PDF, Unix man pages, Windows help files, or even PowerPoint-like slides. For example, you could use a command like this to transform the above DocBook file into HTML: **xsltproc -o docbook-article.html docbook-stylesheets/html/docbook.xsl docbook-article.xml** . The result is an HTML file named docbook-article.html looking something like this in an Web browser.



Alternatively, you can create PDF documents from the DocBook files by using something like FOP and the appropriate XSLT stylesheet. For example, this command might create your PDF document: **fop.sh -xml docbook-article.xml -xsl /docbook/stylesheets/fo/docbook.xsl docbook-article.pdf** . The result is a PDF document looking something like this:



MyLibrary: A Database-Driven Website System for Libraries

Eric Morgan
February, 2003

This article describes a database-driven website application for libraries called MyLibrary.

Table of Contents

Introduction

This article describes a database-driven website application for libraries called MyLibrary.

More than a pretty face

MyLibrary has its roots in a customizable interface to collections of library information resources. As the application has matured, it has become a system for creating and managing a library's website by manifesting the ideas of information architecture. Specifically, the MyLibrary system provides the means for a library to create things like:

- top down site maps
- bottom-up site indexes
- controlled vocabularies
- a means for power searching
- browsable lists of resources
- optional, user-specified customizable interfaces

From the commandline you see if your MyLibrary installation is working correctly by issuing the following command: **./mylibrary.pl** . The result should be a stream of HTML intended for a Web browser.

For more information about MyLibrary, see: http://dewey.library.nd.edu/mylibrary/ [http://dewey.library.nd.edu/mylibrary/] .

1

While the look and feel of the resulting HTML and PDF documents may not be exactly what you want, you can al-

ways created your own XSLT stylesheets using the ones provided by the DocBook community as a template.

Chapter 11. RDF

Introduction

The Resource Description Framework (RDF) is a proposal for consistently encoding metadata in an XML syntax. The grand idea behind RDF is the creation of the Semantic Web. If everybody were to create RDF describing their content, then computers would be able to find relationships between documents that humans would not necessarily discover on their own. At first glance, the syntax will seem a bit overwhelming, but it is not that difficult. Really.

RDF is very much like the idea of encoding meta data in the meta tags of HTML documents. Using the HTML model, meta tags first define a name for the meta tag, say, title. Next, the content attribute of the HTML meta tag is the value for the title, such as Gone With The Wind. For example, the following meta tag may appear in an HTML document: <meta name="title" content="Gone With The Wind"/>. These name/value pairs are intended to describe the HTML document where they are encoded. HTML documents have URL's. These three things, the name, the value, and the URL form what's called, in RDF parlance, a triplet. RDF is all about creating these triplets; it is all about creating name/value pairs and using them to describe the content at URLs.

It is not uncommon to take advantage of the Dublin Core in the creation of these name/value pairs in RDF files. The Dublin Core provides a truly standard set of element names used to describe Internet resources. The fifteen core element names are:

1. title
2. creator
3. subject
4. description
5. publisher
6. contributor
7. date
8. type
9. format

10. identifier
11. source
12. language
13. relation
14. coverage
15. rights

By incorporating an RDF document type declaration as well as the RDF and Dublin Core name spaces into an XML document, it is possible to describe content residing at remote URLs in a standardized way. The valid RDF file below describes three websites using a few Dublin Core elements for the name/value pairs. Once you get past the document type declaration and namespace definitions, the only confusing part of the file is the rdf:Bag element. This particular element is intended to include lists of similar items. In this case, a list of subject terms.

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF PUBLIC "-//DUBLIN CORE//DCMES DTD 2002/07/31//EN"
 "http://dublincore.org/documents/2002/07/31/dcmes-xml/dcmes-xml-dtd.dtd">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:dc="http://purl.org/dc/elements/1.1/">

 <rdf:Description rdf:about="http://www.AcronymFinder.com/">
  <dc:title>Acronym Finder</dc:title>
  <dc:description>The Acronym Finder is a world wide
  web (WWW) searchable database of more than 169,000
  abbreviations and acronyms about computers,
  technology, telecommunications, and military acronyms
  and abbreviations.</dc:description>
  <dc:subject>
   <rdf:Bag>
    <rdf:li>Astronomy</rdf:li>
    <rdf:li>Literature</rdf:li>
    <rdf:li>Mathematics</rdf:li>
    <rdf:li>Music</rdf:li>
    <rdf:li>Philosophy</rdf:li>
   </rdf:Bag>
  </dc:subject>
 </rdf:Description>

 <rdf:Description rdf:about="http://dewey.library.nd.edu/eresources/astronomy.html">
  <dc:title>All Astronomy resources</dc:title>
  <dc:description>This is a list of all the astronomy
  resources in the system.</dc:description>
  <dc:subject>
   <rdf:Bag>
    <rdf:li>Astronomy</rdf:li>
    <rdf:li>Mathematics</rdf:li>
   </rdf:Bag>
  </dc:subject>
 </rdf:Description>

 <rdf:Description rdf:about="http://dewey.library.nd.edu/eresources/literature.html">
  <dc:title>All Literature resources</dc:title>
  <dc:description>This is a list of all the literature
  resources in the system.</dc:description>
  <dc:subject>
   <rdf:Bag>
    <rdf:li>Literature</rdf:li>
    <rdf:li>Philosophy</rdf:li>
   </rdf:Bag>
  </dc:subject>
 </rdf:Description>
```

```
</rdf:RDF>
```

RDF encodings are intended to be embedded in other XML files or exist as stand alone documents. For example, RDF encodings could be embedded in the notesStmt element of TEI files and provide a standardized way to describe the files' content. Sets of TEI files could be read by computers and a catalog could be created accordingly. Unfortunately this doesn't always work because things like the TEI DTD don't support the addition of RDF data. Even though the TEI file might be syntactically valid, the semantic described by the DTD does not take into account RDF. To include RDF data in HTML/XHTML documents, it is suggested by the standard to include a link element in the HTML file's header pointing to the RDF description, something like this: <link rel="meta" type="application/rdf+xml" href="myfile.rdf" />. A computer program could then follow the href attribute in order to read the RDF file.

Exercise

1. In this exercise you will expose the characteristics of Internet documents using RDF.

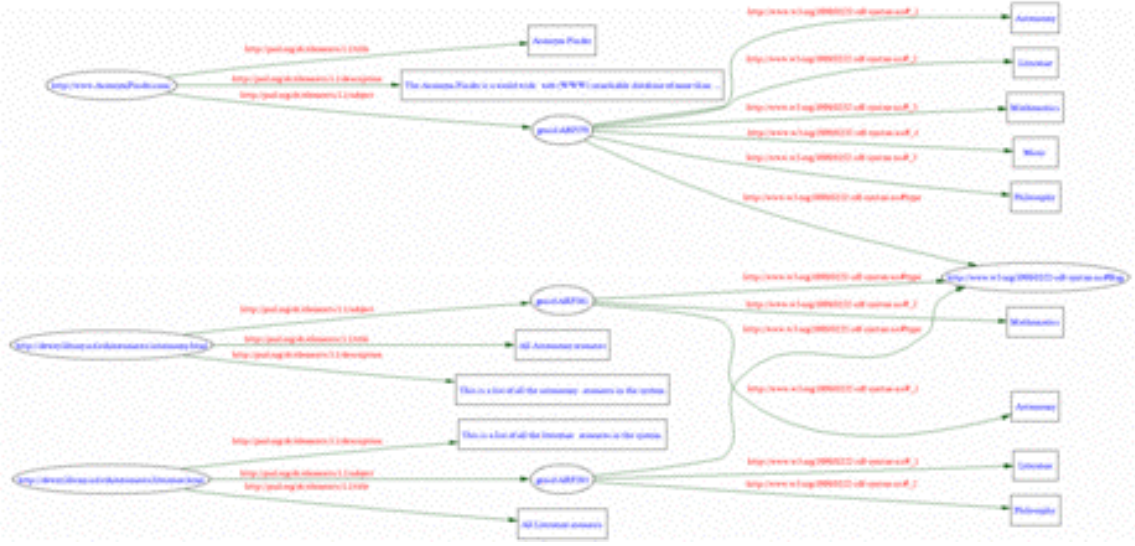
   A. Open the file named rdf.xml in NotePad.

   B. Select the entire contents of the file and copy it to the clipboard.

   C. Point your Web browser to the RDF Validator [http://www.w3.org/RDF/Validator/] .

   D. Paste the copied text into the textarea and submit the form.

   E. Examine the resulting graph, and it should look something like this:



   F. Using rdf.xml as a template, delete all but one of the rdf:Descriptions, and edit the remaining rdf:Description to describe your library's home page.

   G. Repeat Steps #2 through #5.

Chapter 12. Harvesting metadata with OAI-PMH

Note: This is a pre-edited version of a previously published article, Eric Lease Morgan "What is the Open Archives Initiative?" interChange: Newsletter of the International SGML/XML User's Group 8(2):June 2002, pgs. 18-22.

The article describes the intent of the Open Archives Initiative and illustrates a way to implement version 1.1 of the protocol. As of this writing, the protocol has been renamed to the Open Archives Initiative-Protocol for Metadata Harvesting, and it is now at version 2.0. Don't let this dissuade you from reading this section. The majority of it is still quite valid.

What is the Open Archives Initiative?

In a sentence, the Open Archives Initiative (OAI) is a protocol built on top of HTTP designed to distribute, gather, and federate meta data. The protocol is expressed in XML. This article describes the problems the OAI is trying to address and outlines how the OAI system is intended to work. By the end of the article you will be more educated about the OAI and hopefully become inspired to implement your own OAI repository or even become a service provider. The conical home page for the Open Archives Initiative is http://www.openarchives.org/ [http://www.openarchives.org/] .

The Problem

Simply stated, the problem is, "How do I identify and locate the information I need?"

We all seem to be drinking from the proverbial fire hose and suffering from at least a little bit of information overload. Using Internet search engines to find the information we need and desire literally return thousands of hits. Items in these search results are often times inadequately described making the selection of particular returned items a hit or miss proposition. Bibliographic databases -- indexes of scholarly, formally published journal and magazine literature -- overwhelm the user with too many input options and rarely return the full-text of identified articles. Instead, these databases leave the user with a citation requiring a trip to the library where they will have to navigate a physically large space and hope the article is on the shelf.

From a content provider's point of view, the problem can be stated conversely, "How do I make people aware of the data and information I disseminate?"

There are many people, content providers, who have information to share to people who really need it. Collecting, organizing, and maintaining the information is only half the battle. Without access these processes are meaningless. Additionally, there may be sets of content providers who have sets of information with something in common such as subject matter (literature, mathematics, gardening), file format (images, texts, sounds), or community (a library, a business, user group). These sets of people may want to co-operate by assimilating information about their content

together into a single-source search engine and therefore save the time of the user by reducing the number of databases people have to search as well as provide access to the provider's content.

The Solution

The OAI addresses the problems outlined above by articulating a method -- a protocol built on top of HTTP -- for sharing meta data buried in Internet-accessible databases. The protocol defines two entities and the language whereby these two entities communicate. The first entity is called a "data provider" or a "repository". For example, a data provider may have a collection of digital images. Each of these images may be described with a set of qualities: title, accession number, data, resolution, narrative description, etc. Alternatively, a data provider may be a pre-print archive -- a collection of pre-published papers, and therefore each of the items in the archive could be described using title, author, data, summary, and possibly subject heading. Another example could be a list of people, experts in field of study. The qualities describing this collection may be name, email address, postal address, telephone number, and institutional affiliation.

Thus, the purpose of the first OAI entity -- the data provider -- is to expose the qualities of its collection -- the meta data -- to a second entity, a "service provider". The purpose of the service provider is to harvest the meta data from one or more data providers in turn creating a some sort of value-added utility. This utility is undefined by the protocol but could include things such as a printed directory, a federated index available for searching, a mirror of a data provider, a current awareness service, syndicated news feeds, etc.

In summary, the OAI defines two entities (data provider and service provider) and a protocol for these two entities to share meta data between themselves. The balance of this article describes the protocol in greater detail.

Verbs

The OAI protocol consists of only a few "verbs" (think "commands"), and a set of standardized XML responses. All of the verbs are communicated from the service provider to a data provider via an HTTP request. They are a set of one or more name/value pairs embedded in a URL (as in the GET method) or encoded in the HTTP header (as in the POST method). Most of the verbs can be qualified with additional name/value pairs. The simplest verb is "Identify", and a real example of how this might be passed to a data provider via the GET method includes the following:

http://www.infomotions.com/alex/oai/?verb=Identify [http://www.infomotions.com/alex/oai/?verb=Identify]

The example above assumes there is some sort of OAI-aware application saved as the default executable in the /alex/oai directory of the www.infomotions.com host. This application takes the name/value pair, verb=Identify, as input and outputs an XML stream confirming itself as an OAI data provider.

Other verbs work in a similar manner but may include a number of qualifiers in the form of additional name/value pairs. For example, the following verb requests a record, in the Dublin Core meta data format, describing Mark Twain's The Prince And The Pauper:

http://www.infomotions.com/alex/oai/?verb=GetRecord&metadataPrefix=oai_dc&identifier=twain-prince-30
[http://www.infomotions.com/alex/oai/?verb=GetRecord&metadataPrefix=oai_dc&identifier=twain-prince-30]

Again, the default application in the /alex/oai directory takes the value of the GET request as input and outputs a reply in the form of an XML stream.

All six of the protocol's verbs are enumerated and very briefly described below:

1. Identify - This verb is used to verify that a particular service is an OAI repository. The reply to an Identify command includes things like the name of the service, a URL where the services can be reached, the version number of the protocol the repository supports, and the email address to contact for more information. This is by far the easiest verb. Example:

   http://www.infomotions.com/alex/oai/?verb=Identify [http://www.infomotions.com/alex/oai/?verb=Identify]

2. ListMetadataFormats - Meta data takes on many formats, and this command queries the repository for a list of meta data formats the repository supports. In order to be OAI compliant, a repository must at least support the Dublin Core. (For more information about the Dublin Core meta data format see http://dublin.or/ and http://www.iso.or/standards/resources/Z39-85.pdf.) Example:

   http://www.infomotions.com/alex/oai/?verb=ListMetadataFormats
   [http://www.infomotions.com/alex/oai/?verb=ListMetadataFormats]

3. List sets - The data contained in a repository may not necessarily be homogeneous since it might contain information about more than one topic or saved in more than one format. Therefore the verb List sets is used to communicate a list of topic or collections of data in a repository. It is quite possible that a repository has no sets, and consequently a reply would be contain no set information. Example:

   http://www.infomotions.com/alex/oai/?verb=ListSets [http://www.infomotions.com/alex/oai/?verb=ListSets]

4. ListIdentifiers - It is assumed each item in a repository is associated with some sort of unique key -- an identifier. This verb requests a lists of the identifiers from a repository. Since this list can be quite long, and since the information in a repository may or may not significantly change over time, this command can take a number optional qualifiers including a resumption token, date ranges, or set specifications. In short, this command asks a repository, "What items do you have?" Example:

   http://www.infomotions.com/alex/oai/?verb=ListIdentifiers
   [http://www.infomotions.com/alex/oai/?verb=ListIdentifiers]

5. GetRecord - This verb provides the means of retrieving information about specific meta data records given a specific identifier. It requires two qualifiers: 1) the name of an identifier, and 2) name of the meta data format the data is expected to be encoded in. The result will be a description of an item in the repository. Example:

   http://www.infomotions.com/alex/oai/?verb=GetRecord&metadataPrefix=oai_dc&identifier=twain-new-36
   [http://www.infomotions.com/alex/oai/?verb=GetRecord&metadataPrefix=oai_dc&identifier=twain-new-36]

6. ListRecords - This command is a more generalized version of GetRecord. It allows a service provider to retrieve data from a repository without knowing specific identifiers. Instead this command allows the contents of a repository to be dumped en masse. This command can take a number of qualifiers too specifying data ranges or set specifications. This verb has one required qualifier, a meta data specification. Example:

   http://www.infomotions.com/alex/oai/?verb=ListRecords&metadataPrefix=oai_dc
   [http://www.infomotions.com/alex/oai/?verb=ListRecords&metadataPrefix=oai_dc]

Responses -- the XML stream

Upon receiving any one of the verbs outlined above it is the responsibility of the repository to reply in the form of an XML stream, and since this communication is happening on top of the HTTP protocol, the HTTP header's content-type must be text/xml. Error codes are passed via the HTTP status-code.

All responses have a similar format. They begin with an XML declaration. The root of the XML stream always echoes the name of the verb sent in the request as well as a listing of name spaces and schema. This is followed by a date stamp and an echoing of the original request.

For each of the verbs there are a number of different XML elements expected in the response. For example, the Identify verb requires the elements: repositoryName, baseURL, protocolVersion, and adminEmail. Below is very simple but valid reply to the Identify verb:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Identify
 xmlns="http://www.openarchives.org/OAI/1.0/OAI_Identify"
```

```
 xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
 xsi:schemaLocation="http://www.openarchives.org/OAI/1.0/OAI_Identify
 http://www.openarchives.org/OAI/1.0/OAI_Identify.xsd">

 <responseDate>2002-02-16T09:40:35-7:00</responseDate>
 <requestURL>http://www.infomotions.com/alex/oai/index.php?verb=Identify</requestURL>

 <!-- Identify-specific content -->
 <repositoryName>Alex Catalogue of Electronic Texts</repositoryName>
 <baseURL>http://www.infomotions.com/alex/</baseURL>
 <protocolVersion>1.0</protocolVersion>
 <adminEmail>eric_morgan@infomotions.com</adminEmail>
</Identify>
```

The output of the ListMetadataFormats verb requires information about what meta data formats are supported by the repository. Therefore, the response of a ListMetadataFormats request includes a metadataFormat element with a number of children: metadataPrefix, schema, metadataNamespece. Here is an example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ListMetadataFormats
 xmlns="http://www.openarchives.org/OAI/1.0/OAI_ListMetadataFormats"
 xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
 xsi:schemaLocation="http://www.openarchives.org/OAI/1.0/OAI_ListMetadataFormats
 http://www.openarchives.org/OAI/1.0/OAI_ListMetadataFormats.xsd">

 <responseDate>2002-02-16T09:51:49-7:00</responseDate>
 <requestURL>http://www.infomotions.com/alex/oai/index.php?verb=ListMetadataFormats</requestURL>

 <!-- ListMetadataFormats-specific content -->
 <metadataFormat>
  <metadataPrefix>oai_dc</metadataPrefix>
  <schema>http://www.openarchives.org/OAI/dc.xsd</schema>
  <metadataNamespace>http://purl.org/dc/elements/1.1/</metadataNamespace>
 </metadataFormat>

</ListMetadataFormats>
```

About the simplest example can be illustrated with the ListIdentifiers verb. A response to this command might look something like this where, besides the standard output, there is a single additional XML element, identifier:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ListIdentifiers
 xmlns="http://www.openarchives.org/OAI/1.0/OAI_ListIdentifiers"
 xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
 xsi:schemaLocation="http://www.openarchives.org/OAI/1.0/OAI_ListIdentifiers
 http://www.openarchives.org/OAI/1.0/OAI_ListIdentifiers.xsd">

 <responseDate>2002-02-16T10:03:09-7:00</responseDate>
 <requestURL>http://www.infomotions.com/alex/oai/index.php?verb=ListIdentifiers</requestURL>

 <!-- ListIdentifiers-specific content -->
 <identifier>twain-30-44</identifier>
 <identifier>twain-adventures-27</identifier>
 <identifier>twain-adventures-28</identifier>
 <identifier>twain-connecticut-31</identifier>
 <identifier>twain-extracts-32</identifier>
</ListIdentifiers>
```

The last example shows a response to the GetRecord verb. It includes much more information than the previous ex-

amples, because it represents the real meat of the matter. XML elements include the record element and all the necessary children of a record as specified by the meta data format:

```
<?xml version="1.0" encoding="UTF-8" ?>
<GetRecord
 xmlns="http://www.openarchives.org/OAI/1.0/OAI_GetRecord"
 xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
 xsi:schemaLocation="http://www.openarchives.org/OAI/1.0/OAI_GetRecord
 http://www.openarchives.org/OAI/1.0/OAI_GetRecord.xsd">

 <responseDate>2002-02-16T10:09:35-7:00</responseDate>
 <requestURL>http://www.infomotions.com/alex/oai/index.php?verb=GetRecord&metadataPrefix=oai_dc&iden

 <!-- GetRecord-specific content -->
 <record>

  <header>
   <identifier>twain-tom-40</identifier>
   <datestamp>1999</datestamp>
  </header>

  <metadata>

   <!-- Dublin Core metadata -->
   <dc xmlns="http://purl.org/dc/elements/1.1/"
        xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
        xsi:schemaLocation="http://purl.org/dc/elements/1.1/
        http://www.openarchives.org/OAI/dc.xsd">

        <creator>Twain, Mark</creator>
        <title>Tom Sawyer, Detective</title>
        <date>1903</date>
        <identifier>http://www.infomotions.com/etexts/literature/american/1900-/twain-tom-40.txt</id
        <rights>This document is in the public domain.</rights>
        <language>en-US</language>
        <type>text</type>
        <format>text/plain</format>
        <relation>http://www.infomotions.com/alex/</relation>
        <relation>http://www.infomotions.com/alex/cgi-bin/concordance.pl?cmd=selectConcordance&bookc
        <relation>http://www.infomotions.com/alex/cgi-bin/configure-ebook.pl?handle=twain-tom-40</re
        <relation>http://www.infomotions.com/alex/cgi-bin/pdf.pl?handle=twain-tom-40</relation>
        <contributor>Morgan, Eric Lease</contributor>
        <contributor>Infomotions, Inc.</contributor>

   </dc>

  </metadata>

</record>

</GetRecord>
```

An Example

In an afternoon I created the very beginnings of an OAI data provider application using PHP. The source code to this application is available at http://www.infomotions.com/alex/oai/alex-oai-1.0.tar.gz. Below is a snippet of code implementing the ListIdentifiers verb. When this verb is trapped ListIdentifiers.php queries the system's underlying (MySQL) database for a list of keys and outputs the list as per the defined protocol:

```
<?php

 # begin the response
```

```
 echo '<ListIdentifiers
  xmlns="http://www.openarchives.org/OAI/1.0/OAI_ListIdentifiers"
  xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
  xsi:schemaLocation="http://www.openarchives.org/OAI/1.0/OAI_ListIdentifiers
  http://www.openarchives.org/OAI/1.0/OAI_ListIdentifiers.xsd">';
 echo '<responseDate>'. RESPONSEDATE . '</responseDate>';
 echo '<requestURL>' . REQUESTURL . '</requestURL>';

 # create an sql query and execute it
 $sql = "SELECT filename
  FROM titles
  WHERE filename like 'twain%'
  ORDER BY filename";
 $rows = mysql_db_query (DATABASE, $sql);
 checkResults();

 # process each found record
 while ($r = mysql_fetch_array($rows)) {

  # display it
  echo '<identifier>' . $r["filename"] . '</identifier>';

 }

 # finish the response
 echo '</ListIdentifiers>';

?>
```

Conclusion

This article outlined the intended purpose of the Open Archives Initiative (OAI) protocol coupled with a few examples. Given this introduction you may very well now be able to read the specifications and become a data provider. A more serious challenge includes becoming a service provider, and while Google may provide excellent searching mechanisms for the Internet as a whole, services implementing OAI can provide more specialized ways of exposing the "hidden Web".

Part III. Appendices

Appendix A. Selected readings

This is as short list of selected books and websites that can be used to suppliment your knowledge of XML.

XML in general

1. XML in a Nutshell by Elliotte Rusty Harold - A great overview of XML.
2. XML for the World Wide Web by Elizabeth Castro - A step-by-step introduction to many things XML. Very visual.
3. XML From the Inside Out [http://www.xml.com/] - A nice site filled with XML articles.
4. XML Tutorial [http://www.w3schools.com/xml/] - Test your skills with XML here.
5. DTD Tutorial [http://www.w3schools.com/dtd/] - Learn more about DTDs at this site.
6. Extensible Markup Language (XML) [http://www.w3.org/XML/] - The canonical home page for XML.
7. STG XML Validation Form [http://www.stg.brown.edu/service/xmlvalid/] - Check to see of your XML is correct here.

Cascading Style Sheets

1. Cascading Style Sheets, designing for the Web by Håkon Wium Lie - One of the more authoritative books on CSS.
2. Cascading Style Sheets [http://www.w3.org/Style/CSS/] - The cononical home page of CSS.
3. W3C CSS Validation Service [http://jigsaw.w3.org/css-validator/] - Check your CSS files here.
4. CSS Tutorial [http://www.w3schools.com/css/] - Test your knowledge of CSS with this tutorial.

XSLT

1. XSLT Programmer's Reference by Michael Kay - The most authoritative reference for XSLT
2. Extensible Stylesheet Language (XSL) [http://www.w3.org/Style/XSL/] - The cononical homepage for XSLT
3. XSL Tutorial [http://www.w3schools.com/xsl/] - These your XSLT skill here.

DocBook

1. DocDocBook, the definitive guide by Norman Walsh - A bit dated, but the best printed manual about DocBook.
2. DocBook Open Repository [http://docbook.sourceforge.net/] - The best place to begin exploring the Web for DocBook materials.

XHTML

1. Special Edition Using HTML and XHTML by Molly E. Holzchlag - A great overview of XHTML.
2. HyperText Markup Language (HTML) Home Page [http://www.w3.org/MarkUp/] - The cononical home page for HTML.
3. MarkUp Validation Service [http://validator.w3.org/] - Check your HTML markup here.

RDF

1. Resource Description Framework (RDF) [http://www.w3.org/RDF/] - The cononical home page for RDF.
2. RDF Validation Service [http://www.w3.org/RDF/Validator/] - Validate your RDF files here.
3. Dublin Core Metadata Initiative [http://www.dublincore.org/] - The official home page for the Dublin Core.
4. Semantic Web Activity [http://www.w3.org/2000/01/sw/] - Describes the purpose and goal of the Semantic Web.

EAD

1. Encoded Archival Description (EAD) [http://www.loc.gov/ead/] - The cononical home page for EAD
2. EAD Cookbook [http://jefferson.village.virginia.edu/ead/cookbookhelp.html] - A great instruction manual for things EAD.

TEI

1. TEI Website [http://www.tei-c.org/] - The official home page of TEI.
2. TEI Lite [http://www.tei-c.org/Lite/] - A thorough description of TEI Lite.
3. TEI Stylesheets [http://www.tei-c.org/Stylesheets/] - A short list of CSS and XSLT stylesheets for TEI

OAI-PMH

1. Open Archives Initiative [http://www.openarchives.org/] - The cononcial home page for OAI-PMH.