

Getting Started with XML: A Manual and Workshop

Eric Lease Morgan

Getting Started with XML: A Manual and Workshop

by Eric Lease Morgan

This manual is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This manual is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this manual if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

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/>].

Editions:

1. The first public release of this document was dated Saturday, February 22, 2003.
2. The second edition was dated Monday, April 21, 2003 (Dingus Day).
3. This is the third edition of this document, Sunday, October 26, 2003 (getting ready for MCN).
4. This is the fourth edition of this document, Tuesday, 27, 2004 (Shining a LAMP on XML 'n Monterey).

Table of Contents

| | |
|---|------|
| Preface | vii |
| About the book | vii |
| About the author | viii |
| Disclaimer | viii |
| I. General introduction to XML | 1 |
| 1. Introduction | 3 |
| What is XML and why should I care? | 3 |
| 2. A gentle introduction to XML markup | 6 |
| XML syntax | 6 |
| XML documents always have one and only one root element | 6 |
| Element names are case-sensitive | 7 |
| Elements are always closed | 7 |
| Elements must be correctly nested | 7 |
| Elements' attributes must always be quoted | 8 |
| There are only five entities defined by default | 8 |
| Namespaces | 8 |
| XML semantics | 9 |
| Exercise - Checking XML syntax | 10 |
| 3. Creating your own markup | 11 |
| Purpose and components | 11 |
| Exercise - Creating your own XML mark up | 13 |
| 4. Document type definitions | 15 |
| Defining XML vocabularies with DTDs | 15 |
| Names and numbers of elements | 16 |
| PCDATA | 17 |
| Sequences | 17 |
| Putting it all together | 17 |
| Exercise - Writing a simple DTD | 19 |
| Exercise - Validating against a system DTD | 20 |
| Exercise - Fixing an XML document by hand | 20 |
| II. Stylesheets with CSS & XSLT | 22 |
| 5. Rendering XML with cascading style sheets | 24 |
| Introduction | 24 |
| display | 25 |
| margin | 26 |
| text-indent | 26 |
| text-align | 26 |
| list-style | 26 |
| font-family | 27 |
| font-size | 27 |
| font-style | 27 |
| font-weight | 27 |
| Putting it together | 28 |
| Tables | 29 |
| Exercise - Displaying XML Using CSS | 30 |
| 6. Transforming XML with XSLT | 32 |
| Introduction | 32 |
| A few XSLT elements | 32 |
| Exercise - Hello, World | 33 |
| Exercise - XML to text | 34 |
| Exercise - XML to text, redux | 36 |
| Exercise - Transform an XML document into XHTML | 37 |
| Yet another example | 40 |

| | |
|---|-----|
| Exercise - Transform an XML document with an XSLT stylesheet | 42 |
| Displaying tabular data | 42 |
| Manipulating XML data | 44 |
| Using XSLT to create other types of text files | 46 |
| Exercise - XML to XHTML | 47 |
| Exercise - Displaying TEI files in your browser | 47 |
| Exercise - Transform MODS into XHTML | 48 |
| III. Specific XML vocabularies | 49 |
| 7. XHTML | 51 |
| Introduction | 51 |
| Exercise - Writing an XHTML document | 54 |
| Exercise - Convert MARC to XHTML | 55 |
| Exercise - Transform MARCXML-like (SFX) to XHTML | 55 |
| 8. MARCXML | 56 |
| About MARCXML | 56 |
| Exercise - Convert MARC to MARCXML | 58 |
| Exercise - Validating schema | 58 |
| 9. MODS | 59 |
| About MODS | 59 |
| Exercise - Transform MARCXML to MODS | 60 |
| Exercise - Transform MARCXML to MODS, redux | 60 |
| 10. EAD | 62 |
| Introduction | 62 |
| Example | 63 |
| 11. CIMI XML Schema for SPECTRUM | 67 |
| Introduction | 67 |
| Exercise - Updating and validating an XML file with an XML schema | 69 |
| 12. RDF | 70 |
| Introduction | 70 |
| Exercise | 72 |
| Exercise - Transform MARCXML to RDF | 73 |
| 13. TEI | 74 |
| Introduction | 74 |
| A few elements | 74 |
| Exercise | 77 |
| Exercise - Transform TEI to HTML | 78 |
| 14. DocBook | 79 |
| Introduction | 79 |
| Processing with XSLT | 82 |
| Exercise - Make this manual | 84 |
| IV. LAMP | 86 |
| 15. XML and MySQL | 88 |
| About XML and MySQL | 88 |
| Exercise - Transform MODS to SQL | 88 |
| Getting XML output from MySQL | 89 |
| 16. XML and Perl | 93 |
| A simple XSLT transformer | 93 |
| Batch processing | 94 |
| A bit about DOM and SAX | 95 |
| 17. Indexing and searching XML with swish-e | 98 |
| About swish-e | 98 |
| Exercises | 99 |
| Indexing XHTML | 99 |
| Indexing other XML formats | 102 |
| 18. Apache | 104 |
| About Apache | 104 |
| CGI interfaces to XML indexes | 105 |
| Simple XHTML files | 105 |

| | |
|---|-----|
| Improving display of search results | 106 |
| Modern browsers and XML search results | 108 |
| Transforming raw XML on the fly | 108 |
| V. Appendices | 110 |
| A. Harvesting metadata with OAI-PMH | 112 |
| What is the Open Archives Initiative? | 112 |
| The Problem | 112 |
| The Solution | 113 |
| Verbs | 113 |
| Responses -- the XML stream | 115 |
| An Example | 117 |
| Exercise - Making CIMI Schema data available via OAI-PMH | 118 |
| Exercise - Making MARCXML data available via OAI | 119 |
| Conclusion | 120 |
| B. An Introduction to the Search/Retrieve URL Service (SRU) | 121 |
| Introduction | 121 |
| The Problems SRW and SRU are Intended to Solve | 121 |
| SRW and SRU as Web Services | 122 |
| Explain | 122 |
| Scan | 123 |
| SearchRetrieve | 124 |
| A Sample Application: Journal Locator | 125 |
| SRW/U and OAI-PMH | 129 |
| OCKHAM | 129 |
| Summary | 131 |
| Acknowledgements | 131 |
| References | 131 |
| C. Selected readings | 132 |
| XML in general | 132 |
| Cascading Style Sheets | 132 |
| XSLT | 132 |
| DocBook | 132 |
| XHTML | 133 |
| RDF | 133 |
| EAD | 133 |
| TEI | 133 |
| OAI-PMH | 133 |

Preface

About the book

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 seven 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.

Highlights 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 seven 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
- Using LAMP-esque open source software (Linux, Apache, MySQL, Perl) to manipulate and provide access to XML content .

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, RDF, etc.
8. XML and MySQL, Perl, swish-e, and Apache.
9. Web services (OAI and SRU)
10. Selected reading list

About the author

Professionally speaking, Eric Lease Morgan is a librarian first and a computer user second. His goal is to discover new ways to use computers to provide better library services and ultimately increase useful knowledge and understanding.

During the day Eric is the Head of the Digital Access and Information Architecture Department at the University Libraries of Notre Dame. As such he and Team DAIAD help the Libraries do stuff digital. In the evening and on the weekends Eric spends much of his time doing more library work but under the rubric of Infomotions, Inc. While wearing this hat Eric writes computer programs, provides consulting services, and maintains his infomotions.com domain where his photo gallery, Alex Catalogue of Electronic Texts, and Musing on Information are the showcases.

Eric has a BA in Philosophy from Bethany College, Bethany, WV (1982). He has an MIS from Drexel University (1987). He had been witting software since 1978 and been giving it away at least a decade before the term "open source" was coined. In 1998, Eric helped develop and popularize the idea of MyLibrary, a user-driven, customizable interface to collections of library services and content -- a portal.

Recognized numerous times by his peers, Eric won the 1991 Meckler Computers in Libraries Grand Prize and was runner-up in 1990, received three Apple Library of Tomorrow (ALOT) grants, won the 2002 Bowker/Ulrich Serials Librarianship Award, was designated as one of the "Top Librarian Personalities on the Web" in a 2002 issue of Searcher Magazine, was deemed a "2002 Mover & Shaker" in Library Journal, and won the the 2004 LITA HiTech Award for excellent communication and contributions to the profession.

In his copious spare time, Eric can be seen playing disc (frisbee) golf and folding defective floppy disks into intricate origami flora and fauna.

Disclaimer

This workbook has grown larger and larger over the past couple of years. Through the process it has lost some of its coherency. I know there are spelling and gramatical errors. The whole thing is in need of a good editor. If you enjoy editing, and if you have some understanding of DocBook, then don't hesitate to "apply within".

Eric Lease Morgan (eric_morgan@infomotions.com [mailto:eric_morgan@infomotions.com])

Part I. General introduction to XML

Table of Contents

| | |
|---|----|
| 1. Introduction | 3 |
| What is XML and why should I care? | 3 |
| 2. A gentle introduction to XML markup | 6 |
| XML syntax | 6 |
| XML documents always have one and only one root element | 6 |
| Element names are case-sensitive | 7 |
| Elements are always closed | 7 |
| Elements must be correctly nested | 7 |
| Elements' attributes must always be quoted | 8 |
| There are only five entities defined by default | 8 |
| Namespaces | 8 |
| XML semantics | 9 |
| Exercise - Checking XML syntax | 10 |
| 3. Creating your own markup | 11 |
| Purpose and components | 11 |
| Exercise - Creating your own XML mark up | 13 |
| 4. Document type definitions | 15 |
| Defining XML vocabularies with DTDs | 15 |
| Names and numbers of elements | 16 |
| PCDATA | 17 |
| Sequences | 17 |
| Putting it all together | 17 |
| Exercise - Writing a simple DTD | 19 |
| Exercise - Validating against a system DTD | 20 |
| Exercise - Fixing an XML document by hand | 20 |

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. Ad-

ditionally, 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 ')
7. When necessary, namespaces must be employed to eliminate vocabulary clashes.

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 in 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 `<p>` and `</p>` 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 elements 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 its 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; the 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 to 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 `"`;

Namespaces

The concept of a "namespace" is used to avoid clashes in XML vocabularies.

Remember, the X in XML stands for extensible. This means you are allowed to create your own XML vocabulary. There is no centralized authority to dictate what all the valid vocabularies are and how they are used. Fine, but with so many XML vocabularies there are bound to be similarities between them. For example, it is quite likely that different vocabularies will want some sort of date element. Others will want a name or description element. In each of these vocabularies the values expected to be stored in date, name, or description elements may be different. How to tell the difference? Namespaces.

Namespaces have a one-to-one relationship with a URI (Universal Resource Identifier), and namespace attributes defined with URIs can be inserted into XML elements to denote how an element is to be used. Namespace attributes always begin with "xmlns". Namespace attributes always end with some sort of identifier call the "local part". These two things are always delimited by a colon and finally equated with a URI. For example, a conventional namespace defining the Dublin Core namespace is written as:

```
xmlns:dc="http://purl.org/dc/elements/1.1/"
```

where:

- xmlns denotes a namespace
- dc denotes the name of the namespace, the local part
- <http://purl.org/dc/elements/1.1/> is a unique identifier (URI)

This whole namespace thing become very useful when an XML document uses two or more XML vocabularies. For example, it is entirely possible (if not necessary) to have more than one vocabulary in RDF streams. There is one vocabulary used to describe the RDF structure, and there is another vocabulary used to describe the metadata. The example below is a case in point:

```
<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:RDF>
```

Here you have two vocabularies going on. One is defined as rdf and associated with the URI <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. The second one is defined as dc and associated with <http://purl.org/dc/elements/1.1/>. The namespace local parts are then associated with various elements as needed.

One final note. URI are simply unique identifiers. They often take the shape of URLs, but need not point to anything accessible over the Web. They are just strings of text.

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 its 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 Melville,

I have been reading your ideas concerning 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 highlight 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 safe. 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>Melville Dewey</name>
    <address_one>Columbia University</address_one>
```

```
<address_two>New York, NY</address_two>
</addressee>

<greeting>Dear Melville,</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 salutation 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. 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>  
</pets>
```

```
<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 an element called `pets` and it is allowed to include just one 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 has 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 addressee element, one greeting element, at least one paragraph or at least one list element, and one closing element.
2. The date element contains plain 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>
      Melville Dewey
    </name>
    <address_one>
      Columbia University
    </address_one>
    <address_two>
      New York, NY
    </address_two>
  </addressee>
  <greeting>
    Dear Melville,
  </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 beginnings of an element declaration for each ele-

ment 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/>].

Exercise - Validating against a system DTD

In this exercise you will use `xmllint` to validate a locally defined (system) DTD. While using a remote service such as the one mentioned above is easy, it really behooves you to be more self-reliant than that.

- 1. Install `xmllint`. `Xmllint` is a validation program written against a set of C libraries called `libxml2` and `libxslt`. Installing these libraries on Unix follows the normal installation processes: download, unzip, untar, configure, make, and install. On Windows is it much easier to download the pre-compiled binaries making sure you download the necessary `.dll` files. All of these files have been saved in a directory on the CD called `libxml`. Simply copy this directory to your C drive and/or add the `libxml` directory to your `PATH` environment variable. Once done you should be able to enter `xmllint` and `xsltproc` from the command line and see lot's of help text.
- 2. Open a command prompt and change directories to the getting-started directory of the workshop's distribution.
- 3. Validate `letter.xml` against `letter.dtd` using this command: **`xmllint --dtdvalid letter.dtd letter.xml`** . If everything goes well you should see a stream of XML without any errors. If you want to repress the stream of XML then add `--noout`: **`xmllint --noout --dtdvalid letter.dtd letter.xml`** .

To good to be true? Change something about `letter.xml` to make it invalid, and try validating it again to demonstrate that `xmllint` is working correctly.

Exercise - Fixing an XML document by hand

If you have not had the joy of trying to fix an XML file based on the output of `xmllint`, then here is your chance. In this exercise you will make an XML document validate against a DTD.

- 1. Browse the content of the directory `xml-data/ead/broken/`. It contains sets of well-formed XML documents that validated against an older version of an EAD (Encoded Archival Description) DTD.
- 2. Open an EAD file in your favorite text editor, say `ncw.xml`. Notice its structure. At first glance, especially to the uninitiated, the file seems innocuous.
- 3. Open a command prompt and change directories to the root of the workshop's distribution.

4. Validate ncw.xml against the latest version of the EAD DTD supplied on the CD: **xmllint --noout - -dtdvalid dtds/ead/ead.dtd xml-data/ead/broken/ncw.xml** . You should get output looking something like this:

xml-data/ead/broken/ncw.xml:5: element eadheader: validity error : Syntax of value for attribute langencoding of eadheader is not valid

xml-data/ead/broken/ncw.xml:21: element archdesc: validity error : Element archdesc content does not follow the DTD, expecting (runner* , did , (accessrestrict | accruals | acqinfo | altformavail | appraisal | arrangement | bibliography | bioghist | controlaccess | custodhist | descgrp | fileplan | index | odd | originalsloc | otherfindaid | phystech | prefercite | processinfo | relatedmaterial | scopecontent | separatedmaterial | userrestrict | dsc | dao | daogrp | note)*), got (did admininfo scopecontent bioghist controlaccess dsc)

xml-data/ead/broken/ncw.xml:21: element archdesc: validity error : No declaration for attribute langmaterial of element archdesc

xml-data/ead/broken/ncw.xml:21: element archdesc: validity error : No declaration for attribute legalstatus of element archdesc

xml-data/ead/broken/ncw.xml:35: element admininfo: validity error : No declaration for element admininfo

Document xml-data/ead/broken/ncw.xml does not validate against dtds/ead/ead.dtd

Yuck!

5. You can make the document validate by opening up ncw.xml in your text editor and then:
- deleting the space in the langencoding attribute of the eadheader element
 - deleting the admininfo element and all of its children
 - deleting the langmaterial name/value from the archdesc element
 - deleting the legalstatus name/value from the archdesc element
6. Save your changes, and validate the document again.

Part II. Stylesheets with CSS & XSLT

Table of Contents

| | |
|--|----|
| 5. Rendering XML with cascading style sheets | 24 |
| Introduction | 24 |
| display | 25 |
| margin | 26 |
| text-indent | 26 |
| text-align | 26 |
| list-style | 26 |
| font-family | 27 |
| font-size | 27 |
| font-style | 27 |
| font-weight | 27 |
| Putting it together | 28 |
| Tables | 29 |
| Exercise - Displaying XML Using CSS | 30 |
| 6. Transforming XML with XSLT | 32 |
| Introduction | 32 |
| A few XSLT elements | 32 |
| Exercise - Hello, World | 33 |
| Exercise - XML to text | 34 |
| Exercise - XML to text, redux | 36 |
| Exercise - Transform an XML document into XHTML | 37 |
| Yet another example | 40 |
| Exercise - Transform an XML document with an XSLT stylesheet | 42 |
| Displaying tabular data | 42 |
| Manipulating XML data | 44 |
| Using XSLT to create other types of text files | 46 |
| Exercise - XML to XHTML | 47 |
| Exercise - Displaying TEI files in your browser | 47 |
| Exercise - Transform MODS into XHTML | 48 |

Chapter 5. 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>
```

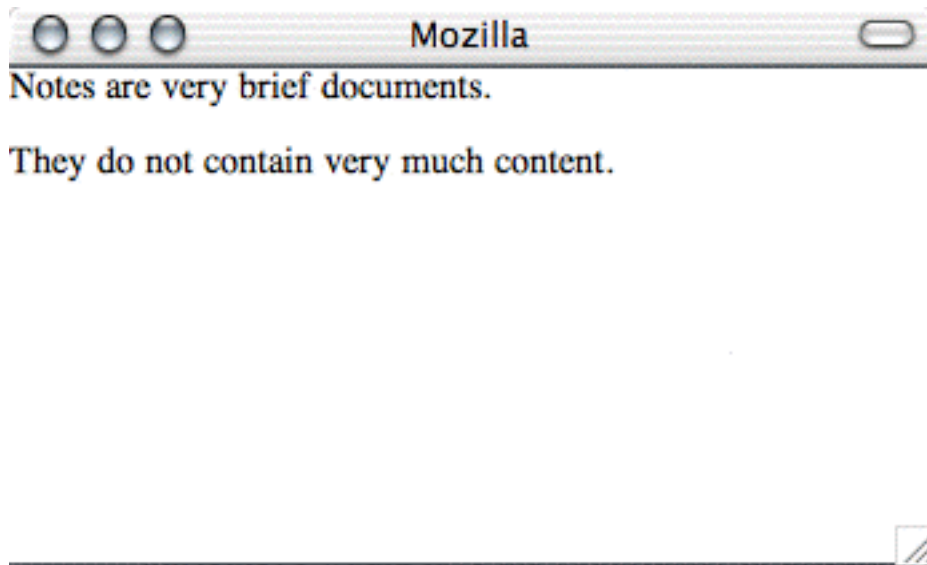
The first thing you will notice about the XML document is the addition of the very first line, an XML processing instruction. This particular instruction tells the application reading the XML file to render it using a CSS file named `note.css`. The balance of the XML file should be familiar to you.

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 mileage 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;
}

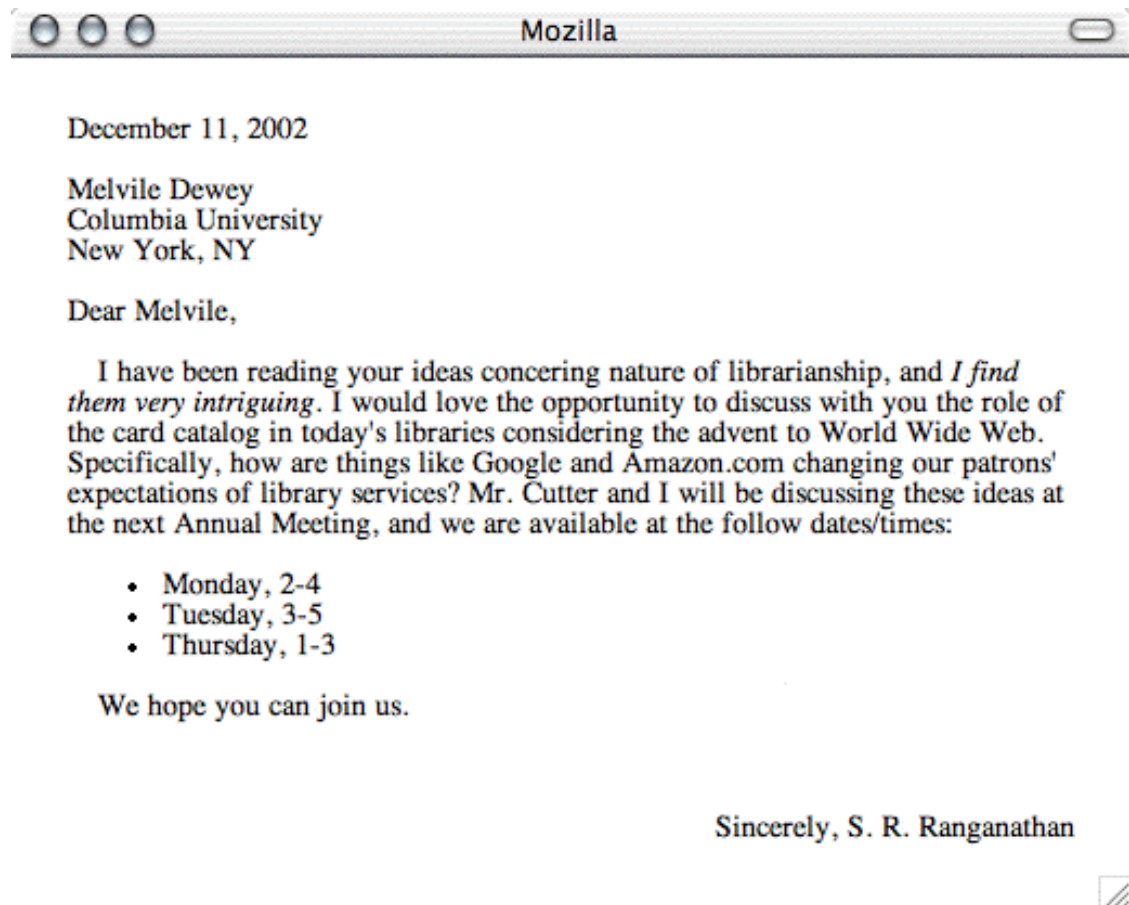
italics {
  display: inline;
  font-style: italic;
}

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 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;

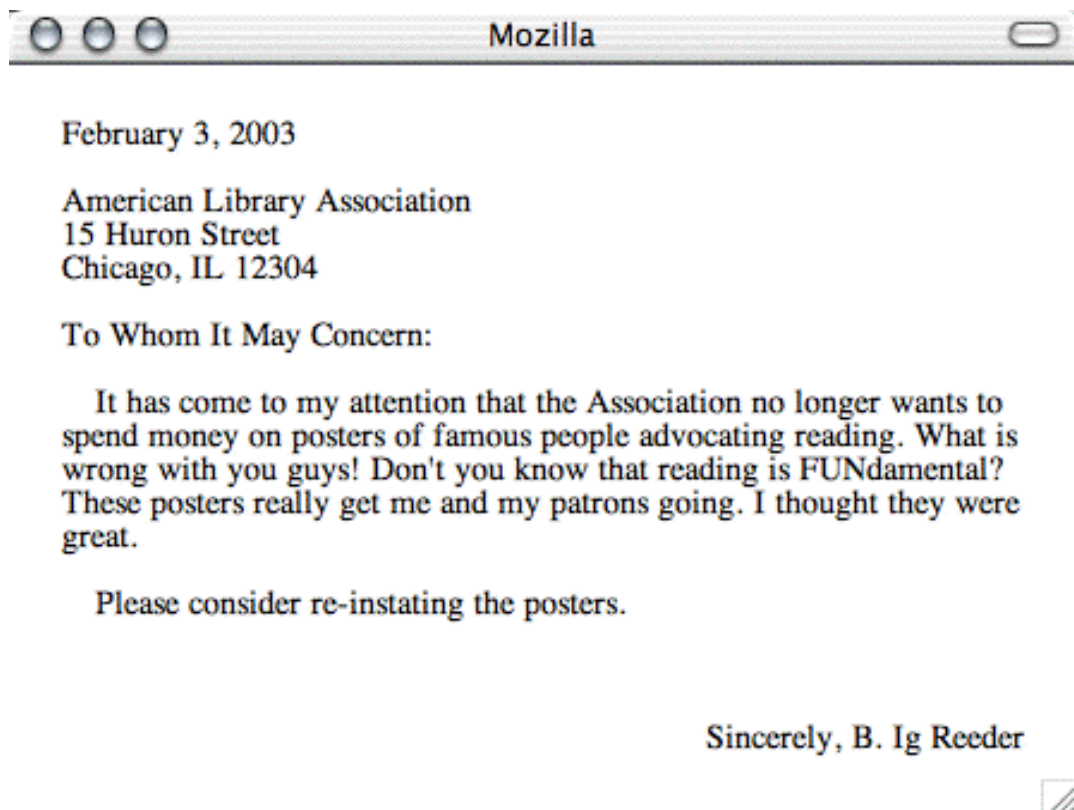
Table-caption is used to give an overall description of the table. Table-header-group is used to denote the labels for the column headings.

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 6. 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: Transformation (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

A few XSLT elements

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. Here is a simple list of some of those commands:

- **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.

Exercise - Hello, World

In this exercise you will transform the simplest of XML documents using XSLT.

Here is a very simple XML document:

```
<content>Hello, World!</content>
```

Our goal is to transform this document into a plain text output. To do that we will use this XSLT stylesheet:

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

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

  <!-- match the root element -->
```

```
<xsl:template match="/content">

  <!-- output the contents of content and a line-feed -->
  <xsl:value-of select='.'/>
  <xsl:text>&#xa;</xsl:text>

  <!-- clean up -->
</xsl:template>

</xsl:stylesheet>
```

This is what the stylesheet does:

1. Defines itself as an XML file
2. Defines itself as an XSLT stylesheet
3. Defines the output format as plain text
4. Looks for the root element, content.
5. Outputs the value of the root element
6. Outputs a line-feed character
7. Closes the opened elements

Give the stylesheet a try:

1. Opening a command prompt.
2. Change directories to the getting-started directory of the workshop's directory.
3. Transform the hello-world.xml file: **xsltproc hello-world.xsl hello-world.xml** .

Exercise - XML to text

In a previous exercise you created a letter (ala.xml) using our own mark up. We will now use XSLT to transform it to plain text file.

As review, open ala.xml. Not too complicated.

To create a plain text version of this file, we will use the following stylesheet:

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

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

  <!-- let's get started -->
```

```
<xsl:template match="/letter">

  <!-- date -->
  <xsl:text>&#xa;</xsl:text>
  <xsl:value-of select='normalize-space(date)'/>
  <xsl:text>&#xa;&#xa;</xsl:text>

  <!-- address -->
  <xsl:value-of select='normalize-space(addressee/name)'/>
  <xsl:text>&#xa;</xsl:text>
  <xsl:value-of select='normalize-space(addressee/address_one)'/>
  <xsl:text>&#xa;</xsl:text>
  <xsl:value-of select='normalize-space(addressee/address_two)'/>
  <xsl:text>&#xa;&#xa;</xsl:text>

  <!-- greeting -->
  <xsl:value-of select='normalize-space(greeting)'/>
  <xsl:text>&#xa;&#xa;</xsl:text>

  <!-- paragraphs -->
  <xsl:for-each select='paragraph'>
    <xsl:value-of select='normalize-space(.)'/>
    <xsl:text>&#xa;&#xa;</xsl:text>
  </xsl:for-each>

  <!-- closing -->
  <xsl:value-of select='normalize-space(closing)'/>
  <xsl:text>&#xa;&#xa;</xsl:text>

</xsl:template>

</xsl:stylesheet>
```

Here is how the stylesheet works:

1. Like before, the file is denoted as an XML file and specifically an XSLT stylesheet.
2. Like before, output is defined as plain o' text.
3. The root of the XML to be transformed is located.
4. The stylesheet outputs a line feed, outputs the value of the date element while removing extraneous white space, and outputs two more line feeds.
5. The stylesheet continues in this fashion for the address and greeting elements.
6. The stylesheet loops through each paragraph element, normalizes the content it finds, and outputs line feeds after each one.
7. The stylesheet finishes by reading the closing element, and then closing all opened elements.

Try using the stylesheet:

1. Open a command prompt and change directories to the getting-started directory.
2. Transform the document: **xsltproc ala2txt.xsl ala.xml** .

Exercise - XML to text, redux

In this exercise you will learn how to make XSLT file a bit more modular and less like CSS files.

Many of the elements of the file ala.xml where intended to be processed similarly. Computer programmers don't like to do the same thing over and over again. They want to code it once and leave it at that.

Note the following stylesheet:

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

  <!-- plain ol' text; get rid of some white space -->
  <xsl:output method='text' />
  <xsl:strip-space elements='*' />

  <!-- let's get started -->
  <xsl:template match="/letter">

    <!-- add a line-feed for formatting's sake -->
    <xsl:text>&#xa;</xsl:text>

    <!-- do the work -->
    <xsl:apply-templates />

  </xsl:template>

  <!-- trap all the various elements -->
  <xsl:template match='date | address_two | greeting | paragraph | closing'>
    <xsl:value-of select='normalize-space(.)' />
    <xsl:text>&#xa;&#xa;</xsl:text>
  </xsl:template>

  <xsl:template match='name | address_one'>
    <xsl:value-of select='normalize-space(.)' />
    <xsl:text>&#xa;</xsl:text>
  </xsl:template>

  <xsl:template match='list'>
    <xsl:apply-templates />
    <xsl:text>&#xa;</xsl:text>
  </xsl:template>

  <xsl:template match='item'>

    <!-- insert a tab, an asterisk, and a space for formatting -->
    <xsl:text>&#x9;*&#x20;</xsl:text>
    <xsl:value-of select='normalize-space(.)' />
    <xsl:text>&#xa;</xsl:text>
  </xsl:template>

</xsl:stylesheet>
```

It works very much like the previous examples but with a number of exceptions:

1. The use of the strip-space eliminates extraneous spaces within elements
2. The use of apply-templates is very important. When this element is encountered the XSLT processor transverses the XML input for elements. When it finds them the processor looks for a matching template containing the current element. If it finds a matching template, then processing is done within the template element. Otherwise the element's content is output.
3. Notice how the date, address_two, greeting, paragraph, and closing elements are all processed the same.
4. Notice how the name and address_one elements are processed the same
5. The list and item elements are a bit tricky. The list element is first trapped and then the item element is rendered. Processing then returns to the template for list elements and a simple line feed is output

Transform letter.xml with letter2text.xsl:

1. Open a command prompt to the getting-started directory of the workbook's CD.
2. Transform letter.xml: **xsltproc letter2txt.xsl letter.xml .**

Exercise - Transform an XML document into XHTML

Below is our first XSLT/XHTML 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 xsltproc 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, specifically, 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>
```

```
</p>
</xsl:template>

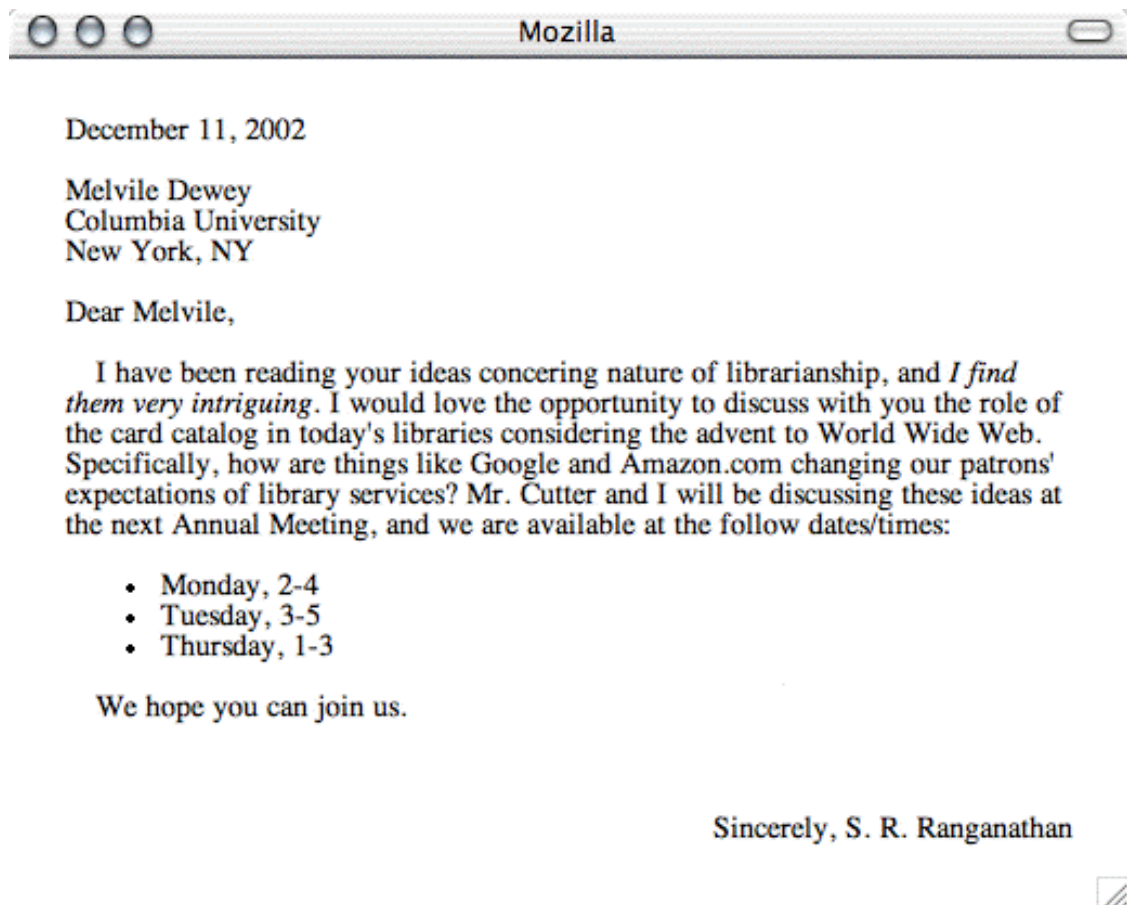
<!-- closing -->
<xsl:template match="closing">
  <p style="margin-top: 3em; text-align: right">
    <xsl:apply-templates/>
  </p>
</xsl:template>

<!-- 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.

Yet another example

Here is yet another example of transforming an XML document into an HTML document. The XSLT file below is intended to convert a CIMI Schema document (an XML vocabulary used to describe objects in museum collections) into an HTML file. Once processed, this XSLT file will:

1. output an HTML declaration
2. find the root of the CIMI Schema document
3. output the beginnings of an HTML document
4. loop through all the object elements of the CIMI Schema document outputting an unordered list of hypertext links pointing to a set of images
5. output the end of an HTML document

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



```

<!-- cimi2html.xsl - convert a CIMI Schema document into a rudimentary HTML file -
<!-- Eric Lease Morgan (emorgan@nd.edu) - October 20, 2003 -->

<!-- lots o' credit goes to Stephen Yearl of Yale who helped with XSL weirdness!

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                 xmlns:c="http://www.cimi.org/wg/xml_spectrum/Schema-v1.5"
                 version="1.0">

  <!-- output an HTML header -->
  <xsl:output method='html'
    doctype-public='-//W3C//DTD XHTML 1.0 Transitional//EN'
    doctype-system='http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'
    indent='no' />

  <!-- find the root of the input -->
  <xsl:template match="/">

    <!-- start the XHTML output -->
    <html>
      <body>
        <h1>Water Collection</h1>
        <ol>

          <!-- find all the Schemas object -->
          <xsl:apply-templates />

        </ol>
      </body>
    </html>
  </xsl:template>

  <!-- trap the objects of the file -->
  <xsl:template match="//c:object">

    <!-- extract the parts of the object we desire and format them -->
    <li>
      <a>
        <xsl:attribute name='href'>
          <xsl:value-of select='./c:reproduction/c:location' />
        </xsl:attribute>
        <xsl:value-of select='./c:identification/c:object-title/c:title' />
      </a> -
      <xsl:value-of select='./c:identification/c:comments' />
      (Collected by
      <xsl:value-of select='./c:acquisition/c:source/c:source/c:person/c:name/c:fore
      <xsl:value-of select='./c:acquisition/c:source/c:source/c:person/c:name/c:surn
      on
      <xsl:value-of select='./c:acquisition/c:accession-date/c:year' />
      -
      <xsl:value-of select='./c:acquisition/c:accession-date/c:month' />
      -
      <xsl:value-of select='./c:acquisition/c:accession-date/c:day' />
      .)
    </li>

  </xsl:template>

</xsl:stylesheet>

```

Exercise - Transform an XML document with an XSLT stylesheet

In this exercise you will transform an XML document using XSLT.

1. Create a directory on your computer's desktop.
2. Copy all the *.dll files from the CD to your newly created directory.
3. Copy all the *.exe files from the CD to your newly created directory.
4. Copy cimi2html.xsl and water.xml from the CD to your newly created directory.
5. Open a new terminal window by running cmd.exe from the Start menu's Run command.
6. Change directories to your newly created directory.
7. Transform the XML document into an HTML document using this command: **xsltproc -o water.html cimi2html.xsl water.xml** .
8. Open the newly created file named water.html in your Web browser.
9. In this part of the exercise you will change the content of the output.
10. Open cimi2html.xsl in your text editor.
11. Add a signature as a footer; insert `<p> Brought to you by [yourname]. </p>` after the `` element of the XSLT file.
12. Process the XML again: **xsltproc -o water.html cimi2html.xsl water.xml** .
13. Open and/or reload the output, water.html, in your browser.
14. Go to Step #2 and make some other changes until you get tired.

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>

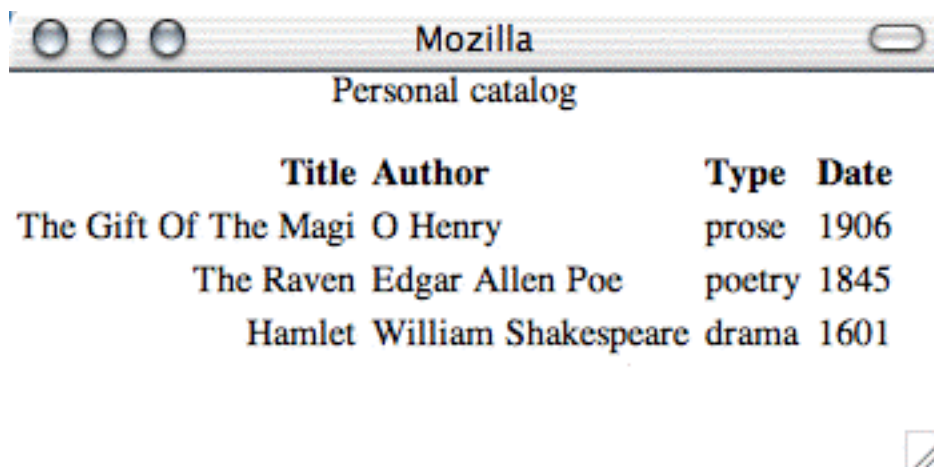
<!-- work -->
<xsl:template match="work">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

<!-- 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:



| Title | Author | Type | Date |
|----------------------|---------------------|--------|------|
| The Gift Of The Magi | O Henry | prose | 1906 |
| The Raven | Edgar Allen Poe | poetry | 1845 |
| Hamlet | William Shakespeare | drama | 1601 |

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)"/>
</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>

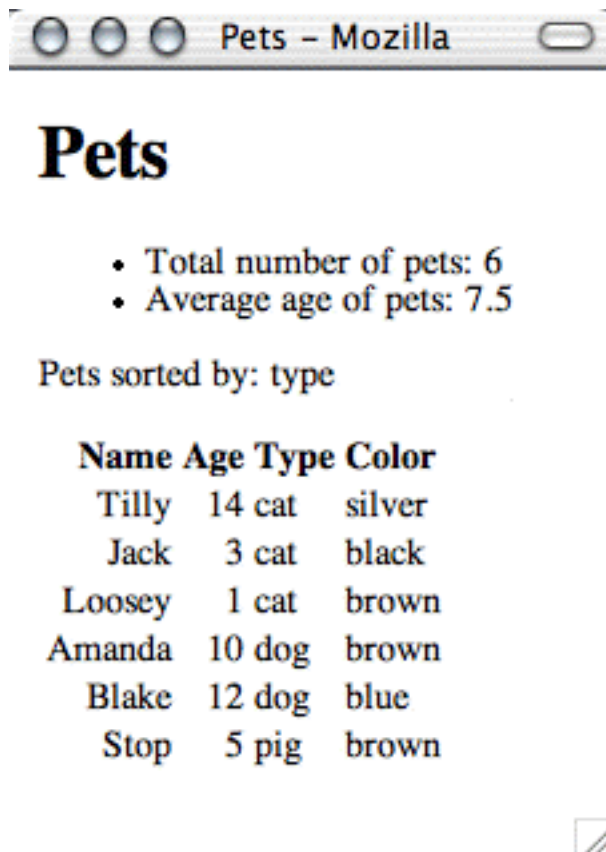
<!-- name -->
<xsl:template match="name">
  <td style="text-align: right"><xsl:value-of select="."/></td>
</xsl:template>

<!-- 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.xml 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.xml -->

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

```
</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', 'silv
INSERT INTO pets (name, age, type, color) WITH VALUES ('Amanda', '10', 'dog', 'bro
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', 'brow
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.

Exercise - XML to XHTML

[INSERT letter2xhtml.xsl HERE]

Exercise - Displaying TEI files in your browser

In this exercise you will learn how to use your Web browser to transform and display XML data using TEI files as examples. This exercise assumes you are using a rather modern browser such as a newer version of Mozilla, Firefox, or Internet Explorer since these browsers have XSLT transformation capability built-in.

1. Open the XSLT stylesheet `xslt/tei2html.xsl` in your Web browser. Notice how it initializes a valid XHTML DTD declaration as well as provides the framework for rich metadata in the head element. Notice how the content of the body is created by searching the document for `div1` tags to build a table of contents. Finally, notice how the balance of the body's content is created by trapping a limited number of TEI elements and transforming them into HTML.
2. Open a TEI file, say `xml-data/tei/machiavelli-prince-1081003648.xml` in your Web browser. It should display the raw XML data.
3. Open `xml-data/tei/machiavelli-prince-1081003648.xml` in your favorite text editor and make the following text the second XML processing instruction in the file: `<?xml-stylesheet type='text/xsl' href='../xslt/tei2html.xsl'?>`. In other words insert the processing instruction before the beginning of the DTD declaration. Save the edited file.
4. Open, again, `xml-data/tei/machiavelli-prince-1081003648.xml` in your Web browser. If all goes well, then you should see nicely rendered text that is more human readable than the raw XML. This

happens because your Web browser read the XML file. Learned that it needed an XSLT file for transformation. Retrieved the XSLT file. Combined the XSLT file with the raw XML, and displayed the results.

Exercise - Transform MODS into XHTML

MODS is an XML bibliographic vocabulary; it is very similar to MARC. In this exercise you will transform a set of MODS data into individual XHTML files.

1. Open `xml-data/mods/single/catalog.xml` in your favorite text editor. Examine the element names and take notice how they are similar to the fields of standard bibliographic data sets.
2. Open `xslt/mods2xhtml.xsl` in your favorite text editor or Web browser. Notice how the stylesheet declaration imports a namespace called `mods`. Notice how the namespace prefixes all of the MODS element names. Notice how the parameter named `filename` is assigned a value from the MODS record. Notice the use of `xsl:document element/command`. It is used to define where output files will be saved. Finally, notice how the XSLT tries create rich and valid XHTML.
3. Open a command prompt and change to the root of the workbook's distribution.
4. Transform a set of MODS records into a set of XHTML files with the following command: `xsltproc xslt/mods2xhtml.xsl xml-data/mods/single/catalog.xml`.
5. Lastly, open the folder/directory named `xml-data/xhtml/mods2xhtml`. You should see sets of numbered files, and each file is a valid XHTML file containing bibliographic information from the `catalog.xml`.
6. As an extra exercise, use `xmllint` to validate the structure and syntax of one or more of the XHTML files.

Part III. Specific XML vocabularies

Table of Contents

| | |
|---|----|
| 7. XHTML | 51 |
| Introduction | 51 |
| Exercise - Writing an XHTML document | 54 |
| Exercise - Convert MARC to XHTML | 55 |
| Exercise - Transform MARCXML-like (SFX) to XHTML | 55 |
| 8. MARCXML | 56 |
| About MARCXML | 56 |
| Exercise - Convert MARC to MARCXML | 58 |
| Exercise - Validating schema | 58 |
| 9. MODS | 59 |
| About MODS | 59 |
| Exercise - Transform MARCXML to MODS | 60 |
| Exercise - Transform MARCXML to MODS, redux | 60 |
| 10. EAD | 62 |
| Introduction | 62 |
| Example | 63 |
| 11. CIMI XML Schema for SPECTRUM | 67 |
| Introduction | 67 |
| Exercise - Updating and validating an XML file with an XML schema | 69 |
| 12. RDF | 70 |
| Introduction | 70 |
| Exercise | 72 |
| Exercise - Transform MARCXML to RDF | 73 |
| 13. TEI | 74 |
| Introduction | 74 |
| A few elements | 74 |
| Exercise | 77 |
| Exercise - Transform TEI to HTML | 78 |
| 14. DocBook | 79 |
| Introduction | 79 |
| Processing with XSLT | 82 |
| Exercise - Make this manual | 84 |

Chapter 7. XHTML



Introduction

XHTML is 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 nest 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 holder 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>
    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 browsers should display something like 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.

Exercise - Convert MARC to XHTML

As you know, MARC is a data structure/file format used to bibliographically describe the many things in libraries. It is also the format used to record authority information as well as holdings. In this exercise you will convert MARC data into a rudimentary XHTML file format.

1. Using your text editor, open the file named marc/single/catalog.marc. Notice the standard format for MARC. Very compact. Very efficient in terms of disc space. Designed for sequential read access from tape drives. Not very human readable. For example, what is the title of the first record?
2. Install Perl. (This process is more completely described in a later section of the workbook. Alternatively, you may be able to simply copy the Perl directory from the CD to the C drive of your Windows computer.)
3. Install MARC::Record. MARC::Record is the Perl module to use when reading and writing MARC data. If you are using Unix, then you should be able to sudo cpan and then install MARC::Record at the cpan prompt. Under Windows you can use ppm and then enter install MARC::Record.
4. Open the file named bin/marc2xhtml.pl in your text editor. Notice how it includes the necessary Perl modules, reads the command line input, initializes a constant, and finally loops through each MARC record from the input in order to output XHTML files. It does this by selectively reading data from each record and stuffing the resulting values into meta elements of the XHTML file's head.
5. Open a command prompt and change directories to the workbook's root.
6. Convert some MARC records by issuing this command: perl bin/marc2xhtml.pl /getting-started/marc/single/catalog.marc /getting-started/xhtml/marc2xhtml/. Be forewarned. You must enter full and complete paths as input. Otherwise the script will get confused.

When done, you should see sets of XHTML files in the output directory. Open one or more using your Web browser. As an extra exercise, use xmllint to validate selected files from the output.

Exercise - Transform MARCXML-like (SFX) to XHTML

[INSERT sfx2html.xsl HERE]

Chapter 8. MARCXML

About MARCXML

MARCXML is vocabulary supporting a "roundtrip" conversion of MARC data to XML. MARCXML has a place for each and every MARC bibliographic field, indicator, subfield, and well as the MARC leader. It is entirely possible to convert a MARC record (or file of MARC records) into a MARCXML file (or file of MARCXML records). Similarly, it is entirely possible to go from a MARCXML file to a MARC file. In neither case will any data be lost in translation. Thus, it supports "roundtrip" conversions.

Below is a MARC record describing a book. Hard carriage returns have been added for display purposes:

```
00707cam 22002411
4500001000800000000500170000800800410002503500210006690600450008701000170
013203500180014904000270016705000150019410000270020924500210023626000510
025730000110030850400300031965100210034998500210037099100430039109900310
0434 6459314 19980421192420.0 721031s1944 nyu b 000 0 eng
9(DLC) 44008235 a7 bcbc coclcrpl du encip f19 gy-gencatlg a
44008235 a(OCOLC)480984 aDLC cODaWU dOCOLC dDLC 00 aF1226 b.S8 1
aStroke, Hudson, d1893- 10 aTimeless Mexico. aNew York, bHarcourt,
Brace and company c[1944] a436 p. aBibliography: p. 425-430.
0 aMexico xHistory. eOCLC REPLACEMENT bc-GenColl hF1226 i.S8 tCopy
1 WOCLREP astrode-timeless-1071960935
```

Here is exactly the same data represented as MARCXML:

```
<?xml version="1.0" encoding="UTF-8"?>
<collection xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.loc.gov/MARC21/slim
    http://www.loc.gov/standards/marcxml/schema/MARC21slim.xsd"
  xmlns="http://www.loc.gov/MARC21/slim">
  <record>
    <leader>00707cam 22002411 4500</leader>
    <controlfield tag="001">6459314</controlfield>
    <controlfield tag="005">19980421192420.0</controlfield>
    <controlfield tag="008">721031s1944 nyu b 000 0 eng </controlfield>
    <datafield tag="035" ind1=" " ind2=" ">
      <subfield code="9">(DLC) 44008235</subfield>
    </datafield>
    <datafield tag="906" ind1=" " ind2=" ">
      <subfield code="a">7</subfield>
      <subfield code="b">cbc</subfield>
      <subfield code="c">oclcrlpl</subfield>
      <subfield code="d">u</subfield>
      <subfield code="e">ncip</subfield>
      <subfield code="f">19</subfield>
      <subfield code="g">y-gencatlg</subfield>
    </datafield>
    <datafield tag="010" ind1=" " ind2=" ">
      <subfield code="a">44008235</subfield>
    </datafield>
```



```
<datafield tag="035" ind1=" " ind2=" ">
  <subfield code="a">(OCoLC)480984</subfield>
</datafield>
<datafield tag="040" ind1=" " ind2=" ">
  <subfield code="a">DLC</subfield>
  <subfield code="c">ODaWU</subfield>
  <subfield code="d">OCoLC</subfield>
  <subfield code="d">DLC</subfield>
</datafield>
<datafield tag="050" ind1="0" ind2="0">
  <subfield code="a">F1226</subfield>
  <subfield code="b">.S8</subfield>
</datafield>
<datafield tag="100" ind1="1" ind2=" ">
  <subfield code="a">Strode, Hudson,</subfield>
  <subfield code="d">1893-</subfield>
</datafield>
<datafield tag="245" ind1="1" ind2="0">
  <subfield code="a">Timeless Mexico.</subfield>
</datafield>
<datafield tag="260" ind1=" " ind2=" ">
  <subfield code="a">New York,</subfield>
  <subfield code="b">Harcourt, Brace and company</subfield>
  <subfield code="c">[1944]</subfield>
</datafield>
<datafield tag="300" ind1=" " ind2=" ">
  <subfield code="a">436 p.</subfield>
</datafield>
<datafield tag="504" ind1=" " ind2=" ">
  <subfield code="a">Bibliography: p. 425-430.</subfield>
</datafield>
<datafield tag="651" ind1=" " ind2="0">
  <subfield code="a">Mexico</subfield>
  <subfield code="x">History.</subfield>
</datafield>
<datafield tag="985" ind1=" " ind2=" ">
  <subfield code="e">OCLC REPLACEMENT</subfield>
</datafield>
<datafield tag="991" ind1=" " ind2=" ">
  <subfield code="b">c-GenColl</subfield>
  <subfield code="h">F1226</subfield>
  <subfield code="i">.S8</subfield>
  <subfield code="t">Copy 1</subfield>
  <subfield code="w">OCLCREP</subfield>
</datafield>
<datafield tag="099" ind1=" " ind2=" ">
  <subfield code="a">strode-timeless-1071960935</subfield>
</datafield>
</record>
</collection>
```

Yes, the MARC record is much more compact, and the MARCXML file is much more verbose. On the other hand, the MARCXML file is much easier to read by humans as well as computers. To prove the point, look at the MARC record and ask yourself, "What is the title of this book?". Now ask the same question of the MARCXML file. Hmmm...

More importantly, there are many more tools enabling the manipulation of XML data structures than MARC data structures. Despite the fact that MARC was born in the mid-1960's, there is an overwhelming desire by information professionals to repurpose XML data as opposed to MARC data. This is why "MARC must die". It is a data structure that has outlived its usefulness. The information contained in a

MARC records is extraordinarily valuable, but the way the encoding scheme, while academically interesting, is archane.

Exercise - Convert MARC to MARCXML

In this exercise you will convert a MARC data file into a single MARCXML file.

1. Install MARC::File::XML. Using Windows you can use ppm and then install MARC::File::XML. Using Unix you can sudo cpan and then install MARC::File::XML at the cpan prompt.
2. After installation a file named marc2xml will be installed in your path, but it is provided in the bin directory for ease-of-use. Open bin/marc2xml.pl in your editor and notice how short the code is.
3. Open a command prompt and change directories to the workbook's root.
4. Enter this command: **perl bin/marc2xml.pl marc/single/catalog.marc | more** . Notice how the output is a stream of MARCXML data.
5. Enter the command again, but this time pipe the output to a file: **perl bin/marc2xml.pl marc/single/catalog.marc > xml-data/marcxml/single/catalog.xml** . Open xml-data/marcxml/single/catalog.xml in your Web browser or editor to verify that the process worked. As an extra exercise, use xmllint to validate the output.

Exercise - Validating schema

In this exercise you will validate MARCXML data against a schema.

1. Open dttds/marcxml/MARC21slim.xsd in your text editor or Web browser. This file is schema. Its purpose is to define the shape and structure of XML documents in the same way DTDs define shape and structure. Schema files are XML files which make them easier to work with than DTDs. They also provide the means for more granular element and attribute definitions. For example, an data can be specified to be in a YYYY-MM-DD format. DTDs do not offer this sort of flexibility. MARC21slim.xsd is used to define the shape of MARCXML files.
2. Open a comand prompt and change to the root of the workshop's directory.
3. Validate the file you created in the previous exercise with this command: **xmllint --noout --schema dttds/marcxml/MARC21slim.xsd xml-data/marcxml/single/catalog.xml** . The result should be "xml-data/marcxml/single/catalog.xml validates".

Based on the author's experience, xmllint's support of schema is not nearly as strong as its support for DTDs. Using xmllint to validate other XML files using schema produces unusual results. Your mileage will vary.

Chapter 9. MODS

About MODS

"MODS" is an acronym for Metadata Object Description Schema. It is a bibliographic schema with library applications in mind. Its elements are a subset of MARC 21 (and therefore MARCXML). It is designed to be more human-friendly than MARC 21 relying on language-based element names as opposed to numeric codes. Instead of denoting a title the code 245 in MARC/MARCXML it is called "title" in MODS. In short, it is a schema designed to build on the good work of MARC and traditional cataloging, and at the same time it is designed to meet the needs of today's environment. Below is a MODS representation of the MARC/MARCXML data from the previous chapter.

```
<mods xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://www.loc.gov/mods/v3" version="3.0"
      xsi:schemaLocation="http://www.loc.gov/mods/v3
      http://www.loc.gov/standards/mods/v3/mods-3-0.xsd">
  <titleInfo>
    <title>Timeless Mexico</title>
  </titleInfo>
  <name type="personal">
    <namePart xmlns:xlink="http://www.w3.org/1999/xlink">Strode, Hudson</namePart>
    <namePart type="date">1893-</namePart>
    <role>
      <roleTerm authority="marcrelator" type="text">creator</roleTerm>
    </role>
  </name>
  <typeOfResource xmlns:xlink="http://www.w3.org/1999/xlink">text</typeOfResource>
  <genre authority="marc">bibliography</genre>
  <originInfo xmlns:xlink="http://www.w3.org/1999/xlink">
    <place>
      <placeTerm type="code" authority="marccountry">nyu</placeTerm>
    </place>
    <place>
      <placeTerm type="text">New York</placeTerm>
    </place>
    <publisher>Harcourt, Brace and company</publisher>
    <dateIssued>[1944]</dateIssued>
    <dateIssued encoding="marc">1944</dateIssued>
    <issuance>monographic</issuance>
  </originInfo>
  <language>
    <languageTerm authority="iso639-2b" type="code">eng</languageTerm>
  </language>
  <physicalDescription>
    <form authority="marcform">print</form>
    <extent>436 p.</extent>
  </physicalDescription>
  <note>Bibliography: p. 425-430.</note>
  <subject xmlns:xlink="http://www.w3.org/1999/xlink" authority="lcsch">
    <geographic>Mexico</geographic>
    <topic>History</topic>
  </subject>
  <classification authority="lcc">F1226 .S8</classification>
  <identifier type="lccn">44008235</identifier>
  <recordInfo xmlns:xlink="http://www.w3.org/1999/xlink">
    <recordContentSource authority="marcorg">DLC</recordContentSource>
    <recordCreationDate encoding="marc">721031</recordCreationDate>
    <recordChangeDate encoding="iso8601">19980421192420.0</recordChangeDate>
```

```
<recordIdentifier>6459314</recordIdentifier>
</recordInfo>
</mods>
```

Exercise - Transform MARCXML to MODS

In this exercise you will transform a MARCXML file into a MODS file.

1. Open `xslt/MARC21slim2MODS3.xsl` in your text editor or Web browser. This file is available at the Library of Congress's MARCXML website, and probably the longest and most complicated XSLT stylesheet in the workbook.
2. Open a command prompt and change directories to the workbook's root.
3. Transform a MARCXML file into a MODS file using this command: **`xsltproc xslt/MARC21slim2MODS3.xsl xml-data/marcxml/single/catalog.xml`** . The result should be streams of XML.
4. Transform the data again, but this time save it to a file: **`xsltproc xslt/MARC21slim2MODS3.xsl xml-data/marcxml/single/catalog.xml > xml-data/mods/single/catalog.xml`** . The advantage of this file, just like the advantage of MARC files and MARCXML files, is all your data is on a single file. As such it is easy to transport from computer to computer.

Exercise - Transform MARCXML to MODS, redux

In this exercise you will transform a single MARCXML file containing many MARCXML records into a set of many individual MODS files. You will find these individual files easier to index.

1. In your text editor, open `xslt/marcxml2mods.xsl`. This is an altered version of the stylesheet used in the previous exercise. It has been altered in such a way to loop through records in a MARCXML file and save them as individual files. This is done through the `xsl:for-each`, `xsl:param`, and `xsl:document` elements/commands. Notice the hard-coded href attribute of the `xsl:document` element:

```
<xsl:for-each select="marc:collection/marc:record">
  <xsl:param name="filename" select='marc:datafield[@tag="099"]/marc:subfield[@tag="a"]>
  <xsl:document method="html" href='xml-data/mods/many/{ $filename }.xml'>
    <mods version="3.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.loc.gov/mods/v3
        http://www.loc.gov/standards/mods/v3/mods-3-0.xsd">
      <xsl:call-template name="marcRecord"/>
    </mods>
  </xsl:document>
</xsl:for-each>
```

2. Open up a command prompt, and change directories to the workbook's root directory.
3. Transform MARCXML data into sets of MODS files: **xsltproc xslt/marcxml2mods.xsl xml-data/marcxml/single/catalog.xml** .
4. Open up the directory xml-data/mods/many/ and browse its contents. You should see sets of MODS files. These files will be fodder for later exercises.

Chapter 10. 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 unpublished 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
- language - 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 where 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 identifying 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>
      <language>
        This file contains only one language,
        <language>
          English
        </language>
      </language>
    </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 11. CIMI XML Schema for SPECTRUM

Introduction

CIMI (formerly known as the Consortium for the Computer Interchange of Museum Information) is an organization assisting the museum community build and adopt standards in an effort to improve the sharing of data and information. To this end, CIMI undertook a study to explore and adapt an XML schema called SPECTRUM from an organization called mda in the United Kingdom. The result is the CIMI XML Schema for SPECTRUM. As described on the CIMI website:

The CIMI XML Schema will enable museums to encode rich descriptive information relating to museum objects, including associated information about people, places and events surrounding the history of museum objects, as well as information about their management and use within museums. The CIMI XML Schema will be useful for migrating data, the sharing of information between applications, and as an interchange format of OAI (Open Archives Initiative) metadata harvesting.

The root element of a CIMI Schema file is the interchange element, and within the interchange element are one or more record elements. Each record element is made up of a data element and a metadata element, and the data elements are described with address, object, organisation, people, person, or place elements. Consequently, a skeleton CIMI Schema document consisting of a single record might look like this:

```
<interchange xmlns="http://www.cimi.org/wg/xml_spectrum/Schema-v1.5">
  <record>
    <data>
      <address />
      <object />
      <organisation />
      <people />
      <person />
      <place />
    </data>
    <metadata />
  </record>
</interchange>
```

Through the liberal use of the remaining elements of the schema, museums ought to be able to track and record characteristics of their collections, objects in their collections, and exhibits.

Just for fun, below is a valid CIMI XML Schema document briefly describing a collection of my own, my water collection. (Yes, I collect water. In fact, I have about 150 waters from all over the world.)

```
<?xml version="1.0"?>
<interchange xmlns="http://www.cimi.org/wg/xml_spectrum/Schema-v1.5">
  <record>
    <data>
```

```
<object>
  <acquisition>
    <accession-date>
      <day>04</day>
      <month>04</month>
      <year>2002</year>
    </accession-date>
    <source>
      <source>
        <person>
          <name>
            <forename>Eric</forename>
            <initials>L.</initials>
            <surname>Morgan</surname>
          </name>
        </person>
      </source>
    </source>
  </acquisition>
  <description>
    <material>water and rocks in a blue glass bottle</material>
  </description>
  <identification>
    <comments>This water was collected by my family and me when
we went to Wales and stayed in a castle.</comments>
    <object-number>brides-bay</object-number>
    <object-title>
      <title>St. Bride's Bay at Newgale (Roch Castle), Wales</title>
    </object-title>
  </identification>
  <reproduction>
    <location>http://infomotions.com/gallery/water/Images/1.jpg</location>
    <type>image/jpeg</type>
  </reproduction>
</object>
</data>
<metadata/>
</record>
<record>
  <data>
    <object>
      <acquisition>
        <accession-date>
          <day>02</day>
          <month>06</month>
          <year>2002</year>
        </accession-date>
        <source>
          <source>
            <person>
              <name>
                <forename>Eric</forename>
                <initials>L.</initials>
                <surname>Morgan</surname>
              </name>
            </person>
          </source>
        </source>
      </acquisition>
      <description>
        <material>water and rocks in a plastic bottle</material>
      </description>
      <identification>
        <comments>On our way out into the sea in Wales, I
```

```
    collected this water.</comments>
  <object-number>whitsands-bay</object-number>
  <object-title>
    <title>Whitesand Bay at St. David's, Wales</title>
  </object-title>
</identification>
<reproduction>
  <location>http://infomotions.com/gallery/water/Images/0.jpg</location>
  <type>image/jpeg</type>
</reproduction>
</object>
</data>
<metadata/>
</record>
</interchange>
```

Exercise - Updating and validating an XML file with an XML schema

In this exercise you will validate an XML file against an XML schema, and then you will edit the XML document.

1. Create a new directory on your computer's desktop.
2. Copy all the *.dll files from the CD to your newly created directory.
3. Copy all the *.exe files from the CD to your newly created directory.
4. Copy all the *.xsd files from the CD to your newly created directory.
5. Save the file named water.xml to the newly created directory.
6. Open a new terminal window by running cmd.exe from the Start menu's Run command.
7. Validate water.xml using xmllint program: **xmllint --schema schema-v1.5.xsd water.xml**.
8. Open the water.xml in your text editor.
9. Copy the entire contents of one of the file's record elements.
10. Add an additional record to the XML file by pasting the contents of the clipboard after the last record element.
11. Change the values of the record's forename, initials, and surname.
12. Change the value of record's comments and object-number.
13. Validate your edits by running the xmllint program again.
14. If errors occur, then read the error messages and try to fix the problem(s). Otherwise, add some more records to the file, read the XSD file and try to add some data to the metadata elements of water.xml, or run the xmllint program without any command line options to learn more about what xmllint can do.

Chapter 12. 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>
```

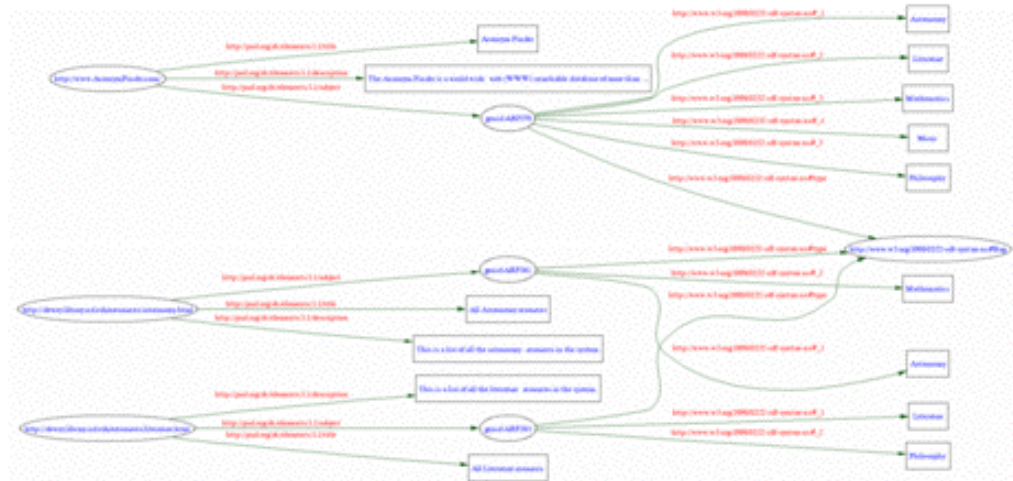
```
<rdf:Description rdf:about="http://dewey.library.nd.edu/eresources/literature.htm"
  <dc:title>All Literature resources</dc:title>
  <dc:description>This is a list of all the literature
resources in the system.</dc:description>
  <dc:subject>
    <rdf:Bag>
      <rdf:li>Literature</rdf:li>
     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

Exercise - Transform MARCXML to RDF

[INSERT MARC21slim2RDFDC.xsl HERE]

Chapter 13. 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>

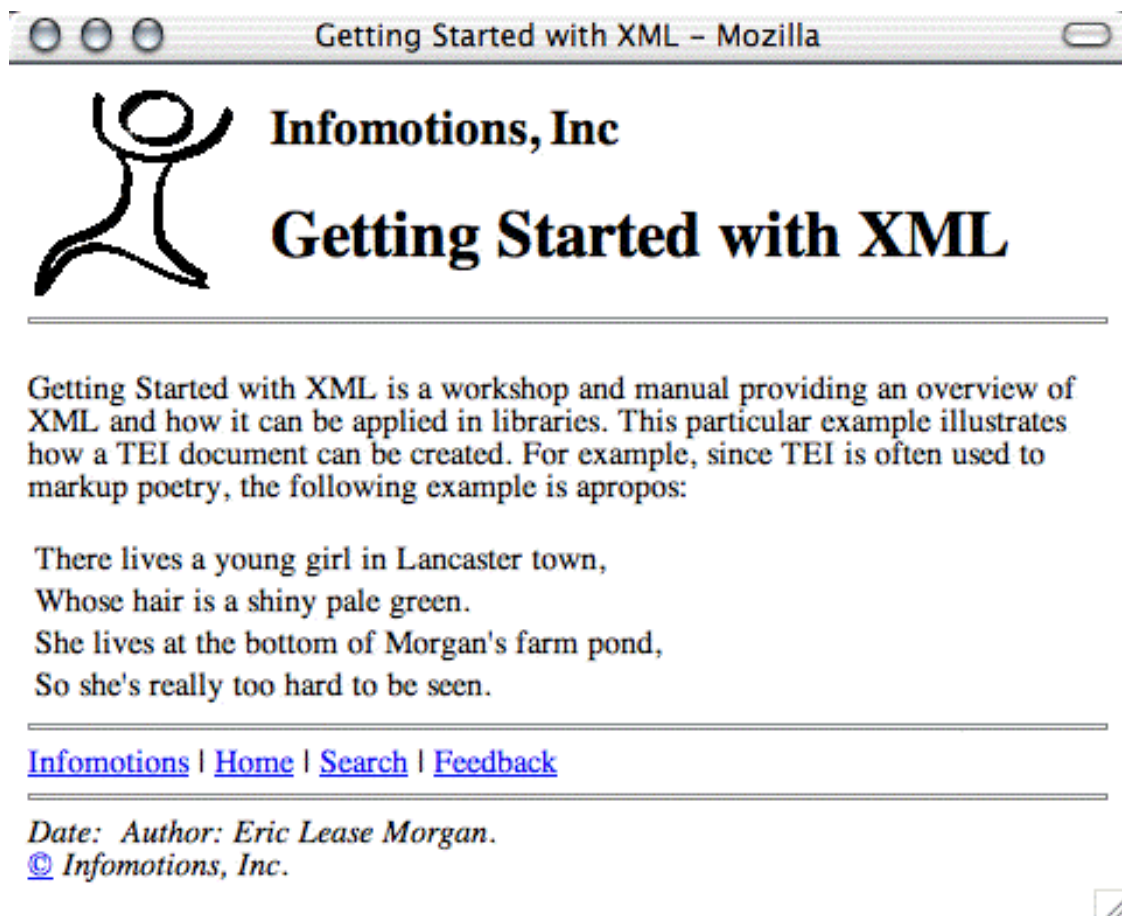
</text>

</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 `xsltproc` 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!

Exercise - Transform TEI to HTML

In this exercise you will transform one or more TEI files into rudimentary HTML files.

1. Browse the contents of the xml-data/tei directory. There you will find sets of valid TEI files ultimately destined for the author's Alex Catalogue of Electronic Texts.
2. Open the XSTL stylesheet at xslt/tei2html.xsl. It is a relatively uncomplicated stylesheet outputting a standard HTML DTD declaration, a full head element, table of contents, and simply formatted body elements. Probably the trickiest aspect of the stylesheet is the way it uses the TEI rend attribute to display HTML p elements.
3. Open up a command prompt and change to the root of the workshop's directory.
4. Transform Mark Twain's A Connecticut Yankee In King Arthurs Court using **xsltproc: xsltproc -o twain.html xslt/tei2html.xsl xml-data/tei/twain-connecticut-1081310767.xml** .
5. The results should be a file named twain.html in the root of the workshop's directory. Open it in your browser and then view the source.

Chapter 14. 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>
```

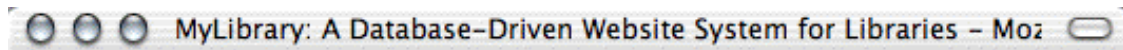


```
<para>
  This article describes a database-driven website
  application for libraries called MyLibrary.
</para>
</section>
<section>
  <title>More than a pretty face</title>
  <para>
    MyLibrary has its roots in a customizable interface to
    collections of library information resources. As the
    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.



MyLibrary: A Database-Driven Website System for Libraries

Eric Morgan

February, 2003

Abstract

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

Table of Contents

[Introduction](#)

[More than a pretty face](#)

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/>.



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/stylesheet/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 | 1 |
| More than a pretty face | 1 |

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/>].

While the look and feel of the resulting HTML and PDF documents may not be exactly what you want, you can always create your own XSLT stylesheets using the ones provided by the DocBook community as a template.

Exercise - Make this manual

This workbook has been marked up using DocBook version 4.2. Given the workbook's raw source files,

sets of XSLT stylesheets, xmllint, and xsltproc you can validate and recreate an XHTML version of the text. Here's how:

1. Open `handout/getting-started.xml` in your text editor. Notice how the file is declared as a DocBook version 4.2 document. Notice how each of the workbook's chapters are defined as entities.
2. Open a command prompt and change directories to the handout directory of the workbook's distribution.
3. Validate the document: **`xmllint --noout --valid getting-started.xml`** . You should get no errors.
4. Change directories to `xslt/docbook-xsl-1.65.1` and browse the contents of the `xhtml` subdirectory. There you will find `docbook.xsl`, the root of the XSLT instructions for creating XHTML output.
5. Change directories back to the handbook directory.
6. Transform the raw DocBook files into XHTML: **`xsltproc -o getting-started.html ../xslt/docbook-xsl-1.65.1/xhtml/docbook.xsl getting-started.xml`**
7. The result should be a file named `getting-started.html` in the handbook directory. Open the file in your Web browser.

If your system has `make` (or `nmake` on Windows) installed, then you should be able to "make" the various versions of this work with simple commands such as: 1) **`make check`** , 2) **`make html`** , and 3) **`make xhtml`** . If you have FOP installed (a processor for creating PDF documents from XML input), then you could do **`make pdf`** and **`make all`** as well. These things are facilitated through the file called `Makefile`. It's a whole lot easier to use `make` commands than typing long `xsltproc` commands.

Part IV. LAMP

Table of Contents

| | |
|---|-----|
| 15. XML and MySQL | 88 |
| About XML and MySQL | 88 |
| Exercise - Transform MODS to SQL | 88 |
| Getting XML output from MySQL | 89 |
| 16. XML and Perl | 93 |
| A simple XSLT transformer | 93 |
| Batch processing | 94 |
| A bit about DOM and SAX | 95 |
| 17. Indexing and searching XML with swish-e | 98 |
| About swish-e | 98 |
| Exercises | 99 |
| Indexing XHTML | 99 |
| Indexing other XML formats | 102 |
| 18. Apache | 104 |
| About Apache | 104 |
| CGI interfaces to XML indexes | 105 |
| Simple XHTML files | 105 |
| Improving display of search results | 106 |
| Modern browsers and XML search results | 108 |
| Transforming raw XML on the fly | 108 |

Chapter 15. XML and MySQL

About XML and MySQL

MySQL is a relational database application, pure and simple. Billed as "The World's Most Popular Open Source Database" MySQL certainly has a wide support in the Internet community. Many people think MySQL can't be very good because it is free, especially Oracle database administrators. True, it does not have all the features of Oracle, nor does it require a specially trained person to keep it up and running. A part of the LAMP suite, MySQL compiles easily on a multitude of platforms. It comes as a pre-compiled binary for Windows. It has been used to manage millions of records and gigabytes of data. Fast and robust, it supports the majority of people's relational database needs. With version 4.0 it supports procedures and triggers, features long desired by people who wanted "a real relational database application". For the purposes of this workbook, the simple XML functionality introduced in 4.0 is more important.

Installing and running a stand alone MySQL server on a Windows computer is relative easy:

1. download the distribution
2. unpack it into its own directory
3. run the Setup application using the default settings
4. open a command prompt
5. start the server (`c:\mysql\bin\mysqld --standalone`)

Installing MySQL on a Unix/Linux host is a bit more involved but pretty much follows the normal open source distribution method:

1. download
2. unzip
3. untar
4. `./configure`
5. `make`
6. `sudo make install`
7. set up the server's root password
8. initialize the server's necessary databases

Exercise - Transform MODS to SQL

In this exercise you will transform a file containing MODS records into an SQL file.

1. Open `xml-data/mods/single/catalog.xml` in your Web browser or text editor. Notice how it a set of MODS records with human-readable element names denoting bibliographic fields.

2. Open `xslt/mods2sql.xml` in your Web browser or text editor. Notice how it outputs plain text. Notice how it outputs SQL commands to create a database table called `items`. Notice how it loops through each MODS record selectively extracting Dublin Core-like elements and putting them into SQL insert statements.
3. Open a command prompt and change to the root of the workshop's distribution.
4. Use `xsltproc` to transform the MODS data into SQL and save it to a file: **`xsltproc xslt/mod2sql.xml xml-data/mods/single/catalog/catalog.xml > sql/catalog/catalog.sql`** .
5. When `xsltproc` is finished, open `sql/catalog/catalog.sql` to see the fruits of your labors. You should see oodles of SQL insert statements.

Getting XML output from MySQL

Given some input you will be able to extract data in a rudimentary XML flavor. Begin by creating a new database called `catalog` using the `mysqladmin` command: **`mysqladmin -p root create catalog`** .

Using data created from the previous exercise you should now be able to fill this new database with content using the `mysql` client. First, change directories to the root level of this workbook's distribution, and then run this command: **`mysql -u root catalog < sql/catalog/catalog.sql`** .

You should now be able to see the fruits of your labors by running the `mysql` client again and exploring your new database. First, connect to the newly populated database: **`mysql -u root catalog`** . Once connected run the following SQL commands at the prompt to see what happened and what is in the database:

- **`show databases;`**
- **`use catalog;`**
- **`show tables;`**
- **`explain items;`**
- **`select title from items;`**
- **`select title from items order by sort_title;`**
- **`select count(id) as 'number of records' from items;`**
- **`select title, creator from items limit 1, 10;`**
- **`select concat(title, ' ', responsibility) as 'title/author list' from items limit 1, 10;`**
- **`select title from items where title like '%blues%';`**

When finished, quit the `mysql` client by entering `\q` at the `mysql` prompt.

MySQL provides the ability to dump the content of a database in an XML form. To do this you use the `mysqldump` command with the `--xml` option. Something like this should work: **`mysqldump --xml -u root catalog`** . The output will look something like this:

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

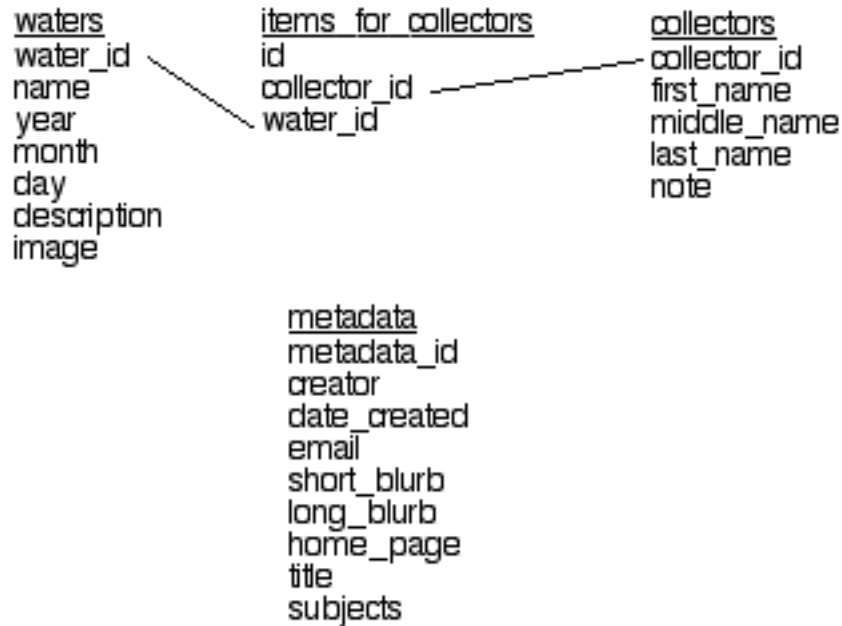
```
<mysqldump>
  <database name="catalog">
    <table name="items">
      <row>
        <field name="id">1</field>
        <field name="creator">Abraham, Gerald</field>
        <field name="title">The concise Oxford history of music</field>
        <field name="sort_title">concise Oxford history of music</field>
        <field name="subjects">Music; History and criticism</field>
        <field name="publisher">Oxford University Press</field>
        <field name="date">1979</field>
        <field name="call_number">ML160 .A27</field>
        <field name="extent">968 p. : ill., music ; 26 cm.</field>
        <field name="notes">Includes index. Bibliography: p. 864-912.</field>
        <field name="responsibility">Gerald Abraham.</field>
      </row>
      <!-- many, many rows deleted here. -->
      <row>
        <field name="id">567</field>
        <field name="creator">Zicree, Marc Scott.</field>
        <field name="title">The twilight zone companion</field>
        <field name="sort_title">twilight zone companion</field>
        <field name="subjects"></field>
        <field name="publisher">Silman-James Press</field>
        <field name="date">1992</field>
        <field name="call_number">PN1992.77.T87 Z52 1992</field>
        <field name="extent">p. cm.</field>
        <field name="notes">by Marc Scott Zicree. Includes index.</field>
        <field name="responsibility">by Marc Scott Zicree.</field>
      </row>
    </table>
  </database>
</mysqldump>
```

While pretty vanilla, the output is a perfect representation of the database's content. Using XSLT it would be possible to reformat all or parts of this output into other textual representations.

The catalog database is a flat file. It contains only a single set of rows and columns. As such it does not provide the means to save redundant data very well. For example, each item in the catalog may have multiple authors as well as multiple subjects. These individual authors and subjects may appear in multiple items. In relational database parlance, this is called a many-to-many relationship. In order to more efficiently represent this condition relational database applications support the creation of multiple tables within a single database. In this case, one table could contain information about items. Another could contain authors. A third could contain subjects. Finally, there could be "join" tables containing unique identifiers from the other tables facilitating relationships between them. The process of defining these relationships is called "normalization".

A true relational database is included in this workshop's distribution. The database is a small part of the author's water collection. ("Yes, I collect water.") Each water in the collection was collected by one or more persons, and each person may have collected many waters. Thus, the water collection includes at least one many-to-many relationship. Each water is uniquely identified with an ID, title, date, a short blurb, and a JPEG image. Each item is then "joined" to one or more collectors through a related table. Finally, the entire collection is described with a single metadata record. The database's entity relationship diagram looks like this:

simple water collection ER diagram



To explore this database and MySQL's ability to export XML, you need to create the database and fill it with content. Again, change directories to the root of the workshop's distribution and run these MySQL commands:

1. **mysqladmin -u root create waters**
2. **mysql -u root waters < sql/water-collection/water-collection.sql**

Next, use your favorite text editor to open and read the various .sql files in the sql/water-collection directory. The statements in these files are a bit more complicated than the previous examples. This is because of the relational nature of the database.

Finally, use the mysql client with the --xml option to query the waters database and print XML reports. It is easier if you change directories the water-collection directory first and then:

- **mysql --xml -u root waters < get-collectors-01.sql**
- **mysql --xml -u root waters < get-collectors-02.sql**
- **mysql --xml -u root waters < get-collectors-03.sql**
- **mysql --xml -u root waters < get-collectors-04.sql**
- **mysql --xml -u root waters < get-waters-01.sql**
- **mysql --xml -u root waters < get-waters-02.sql**

- **mysql --xml -u root waters < get-waters-03.sql**
- **mysql --xml -u root waters < get-waters-04.sql**
- **mysql --xml -u root waters < get-waters-and-collectors-01.sql**
- **mysql --xml -u root waters < get-waters-and-collectors-02.sql**
- **mysql --xml -u root waters < get-waters-and-collectors-03.sql**

As before, it is possible save the output of these commands to text files and manipulate them using XSLT.

Databases, especially relational databases, are the electronic tool to use to manage lists of things. Lists of employees. Lists of books. Lists of Internet resources. Relational databases are not well suited for managing the content of narrative texts such as the content of books or journal articles. Yes, it is good for describing books and journals, but not managing the book's content itself. It is too difficult to create discrete parts of books. Chapters? Sections? Paragraphs? Sentences? Words? Databases are good at managing lists of things because they excel at the ability of editing specific aspects of a list.

In the form of a report XML is a pretty good way to represent the content of a database, but it does not lend itself very well to editing. There are a few "native XML databases" available supporting querying and updating of XML documents as a whole. A good example is Sleepycat Software's Berkeley DBXML, but few of these native databases seem to be taking the XML world by storm. Time will tell whether or not XML documents can be manipulated in the same way relational database content is manipulated, but right now XML seems to be firmly planted more as a reporting format as opposed to a format for repeated editing.

Chapter 16. XML and Perl

There is no doubt about it, XSLT is programming language. It is a language enabling you to read XML and output text, whether it be in the form of XML or not. But computing tasks involve more than the output of text. There are databases, outside applications and data sets, hardware, and of course human interfaces to take into account. In the big scheme of things XSLT provides limited functionality. There is the need for more all-inclusive programming techniques. The use of Perl is such a technique. It is an excellent scripting language that can glue together the necessary components for a more complete solution. Without getting involved in any programming religious wars, this section outlines a few examples of how you can use Perl to manipulate XML.

A simple XSLT transformer

The functions in libxml2 and libxslt, the C libraries underlying xmllint and xsltproc, have been made available to Perl scripts through two modules: XML::LibXML and XML::LibXSLT. Considering this it is possible to create the Perl equivalent of xsltproc. Here is the outline of such an application. It is saved in the workshop's distribution as bin/xsltproc.pl:

```
#!/usr/bin/perl

# require the necessary modules
use strict;
use XML::LibXML;
use XML::LibXSLT;

# get input
my $xslt_file = $ARGV[0];
my $xml_file = $ARGV[1];

# initialize
my $parser = XML::LibXML->new;
my $xslt = XML::LibXSLT->new;

# parse the input files
my $style = $parser->parse_file($xslt_file);
my $doc = $parser->parse_file($xml_file);

# validate stylesheet
my $stylesheet = $xslt->parse_stylesheet($style);

# transform the source file
my $results = $stylesheet->transform($doc);

# print the result to STDOUT
print $stylesheet->output_string($results);

# done
exit;
```

Here is how the program works:

1. The necessary modules (XML::LibXML and XML::LibXSLT) are "included" in the script.
2. An XSLT stylesheet file name and an XML data file name are read from the command line.

3. XML and XSLT parser objects are initialized.
4. Both of the input files are parsed.
5. The stylesheet is validated.
6. The XML file is transformed using the stylesheet.
7. The transformed document is sent to STDOUT.
8. The program quits.

Give `xsltproc.pl` a go using the source XSLT and XML data files supplied with the workshop's distribution. From the root of the workshop distribution, examples might include:

- `bin/xsltproc.pl xslt/letter2xhtml.xsl getting-started/letter.xml`
- `bin/xsltproc.pl xslt/tei2html.xsl xml-data/tei/poe-cask-1072537129.xml`
- `bin/xsltproc.pl xslt/mods2xhtml-nosave.xsl xml-data/mods/many/adler-development-1072276659.xml`

Batch processing

Sometimes it will be necessary to read many XML documents to create the desired result. This is not very easy using pure XSLT, but Perl can come to the rescue. `Title-index.pl` is a program that reads all of the TEI files in a given directory, extracts the title and author information from each file, and creates an title/author index. The heart of the program is a subroutine called `process_files`:

```
sub process_files {  
    # get the name of the found file  
    my $file = $File::Find::name;  
  
    # make sure it has the correct extension  
    next if ($file !~ m/\.xml$/);  
  
    # parse the file and extract the necessary data; s o s l o w !  
    print "Processing $file... \n";  
    my $doc = $parser->parse_file($file);  
    my $root = $doc->getDocumentElement;  
    my @header = $root->findnodes('teiHeader');  
  
    # extract the desired data  
    my $author = $header[0]->findvalue('fileDesc/titleStmt/author');  
    my $title = $header[0]->findvalue('fileDesc/titleStmt/title');  
  
    # save it  
    push @all_items, ({author=>$author, title=>$title, file=>$file});  
}
```

After all the files in the given directory are processed and the `@all_items` array has been filled, execu-

tion returns to the main program where each item in @all_items is read, re-formatted, and sent to STDOUT. Output looks like this:

- Cash boy by Alger Horatio, Jr. (1834-1899) - /xml-data/tei/alger-cash-1072580839.xml
- Cast upon the breakers by Alger Horatio, Jr. (1834-1899) - /xml-data/tei/alger-cast-1072723382.xml
- My watch: An instructive little tale by Twain, Mark (1835-1910) - /xml-data/tei/twain-my-1072419743.xml
- New crime: Legislation needed by Twain, Mark (1835-1910) - /xml-data/tei/twain-new-1072420178.xml
- Niagara by Twain, Mark (1835-1910) - /xml-data/tei/twain-niagara-1079905834.xml

Give title-index.pl a try against the set of TEI data supplied with the workshop. Be forewarned. You must supply the full path to the TEI directory. Otherwise the script will get confused. An example includes: **bin/title-index.pl /lamp/xml-data/tei/**.

A bit about DOM and SAX

XML is a predicable data structure. After all, that is the whole point. The XML may reside as a file. It may reside in computer memory. It may be manifested as a stream of bits coming over an Internet connection. In any case, the data needs to be read by the computer and something needs to be done with it. The process, the reading of the data by the computer is called parsing, and the Internet community has articulated two standardized ways to facilitate processing. One is called DOM and the other is called SAX.

DOM (Document Object Model) views XML as a tree-like data structure. XML has a single root. "Remember the six simple rules regarding XML files?" From that root grow innumerable branches, limbs, twigs, and leaves -- elements, sub-elements, sub-sub-elements, and ultimately data. DOM provides a standard set of object oriented methods for accessing sections, subsections, or any other specific part of XML. The method getElement, in the previous example, is a DOM method used to return all of the data from the root element. Other methods include but are certainly not limited to:

- createDocument - initialize an XML document
- setDocumentElement - create the root of an XML document
- createElement - add an element to an XML document
- createAttribute - add an attribute to an element
- getAttribute - returns the value of an attribute

Since the DOM has been articulated, many programming languages and programming libraries have implemented it. XML::LibXML is one example. There are advantages and disadvantages to DOM. On the down side, DOM implementations require the entire XML data, file, or stream to be read into memory before processing can begin. This can cause huge memory requirements for the computer. On the other hand DOM processing allows the programmer to jump from section to section of XML with relative ease.

SAX (Simple API for XML) is an event-based parser. SAX reads XML data element by element watching them open and close. As they do, "events" are called and computer programs read the data and pro-

cess it accordingly. As a data structure, XML is a FILO (First In, Last Out) stack, and as the parser makes it way through the stack computing tasks take place. SAX has the advantage over DOM in that it is not memory intensive. Its primary disadvantage is the lack of moving around the data structure. There really aren't very many events a SAX program needs to trap:

- `start_document` - triggered when the document is opened
- `end_document` - triggered when the document is closed
- `start_element` - triggered as an element is opened
- `end_element` - triggered as an element is closed
- `characters` - triggered as after an element is opened but before it is closed

Below is a rudimentary Perl SAX script reading any URL pointing to XML data. While the data is being read it will output what elements are opened, closed, and their content.

```
#!/usr/bin/perl

use strict;
use XML::SAX::ParserFactory;

my $handler = MyHandler->new();
my $parser = XML::SAX::ParserFactory->parser(Handler => $handler);

$parser->parse_uri($ARGV[0]);

exit;

package MyHandler;

sub new {
    my $type = shift;
    return bless {}, $type;
}

sub start_element {
    my ($self, $element) = @_;
    print "Starting element $element->{Name}\n";
}

sub end_element {
    my ($self, $element) = @_;
    print "Ending element $element->{Name}\n";
}

sub characters {
    my ($self, $characters) = @_;
    print "characters: $characters->{Data}\n";
}

1;
```

A program from the workbook's distribution (`fix-ead.pl`) uses SAX to read many files from a directory and convert them to a version of EAD/XML that validates against the latest version of the EAD DTD.

Use your favorite text editor to open `fix-ead.pl`. Notice how it combines the `process_files` subroutine of the previous example with the SAX event-driven parser to loop through a set of files in one directory, reading the files' data, setting and unsetting flags, and outputting results to a second directory. Fix the EAD files in `xml-data/ead/broken` and save them to `xml-data/ead/fixed` using this command: **`bin/fix-ead.pl /lamp/xml-data/ead/broken/ /lamp/xml-data/ead/fixed/`** . Use `xmllint` to selectively validate the newly created EAD files against the workbook's version of the EAD DTD.

Chapter 17. Indexing and searching XML with swish-e

About swish-e

Indexing is one half of the information retrieval coin. The other half being databases. This section outlines how to use swish-e to index, search, and retrieve XML files.

Swish-e is an uncomplicated indexer/search engine that works on Unix/Linux computers as well as Windows. Once built you feed the swish-e binary a configuration file and/or a set of command line switches to index content. This content can be individual files on a file system, files retrieved by crawling a website, or a stream of content from another application such as a database.

The indexing half of swish-e is able to index plain text files (including XML data) or data structured as HTML-like streams. Using supplied plug-ins swish-e can index some binary files such as Microsoft Word and Excel documents and PDF documents. The plug-ins work by converting the documents into plain text files and streaming these conversions to the indexer. Searches against these kinds of indexes return pointers to the originals and thus facilitate retrieval. The indexes created by swish-e are portable from file system to file system.

The same binary that creates the indexes can be used to search them. Swish-e supports relevance ranking, Boolean operations, right-hand truncation, field searching, and nested queries. Later versions of swish-e come with C, Perl, and PHP APIs allowing developers to create stand alone applications or CGI interfaces to their indexes.

Swish-e is not perfect. For example, searches against indexes return only properties of located items (such as title, author, abstract, etc.) and pointers to original documents. It is very difficult to return only parts of a particular document, say paragraph number three of a TEI file. Without a lot of intervention, swish-e does not support exact field matching so queries like "title=time" will return Time, Time Magazine, as well as Time and Time Again. Lastly, and probably most importantly, the swish-e indexer does not support Unicode very well. The two-byte nature of Unicode characters confuses the indexer.

Swish-e is an unsung hero. It's inherently open nature allows for the creation of some very smart search engines supporting things like spelling correction, thesaurus intervention, and "best bet" implementations. Of all the different types of information services librarians provide, access to indexes is one of the biggest ones. With swish-e librarians could create their own indexes and rely on commercial bibliographic indexers less and less.

Installing swish-e is almost a trivial matter. On Unix/Linux systems you simply:

1. download the archive
2. unzip it
3. untar it
4. change into the distribution's directory
5. configure (**./configure**)
6. **make**
7. test (**make check**)

8. `install (sudo make install)`

For best results, it is a good idea have previously installed the libxml2 family of C libraries. To install the Perl module you change to the Perl directory and enter the usual commands for installing modules:

1. `perl Makefile.PL`
2. `make`
3. `make test`
4. `sudo make install`

Installation on Windows is just as easy. Download the distribution, run the resulting .exe file, and the installer will do the rest. Be careful. It is a good idea to use the installer's defaults since applications using swish-e will need access to the necessary dynamically linked library (dll) files. These dll files need to be in your PATH environment variable. If you have previously installed Perl, the installer will install the modules as well as the swish-e application.

After installation, you will have local access to the voluminous documentation, not to mention the HTML-based documentation in the distribution's html directory:

- `swish-e -h`
- `man swish-e`
- `man SWISH-CONFIG`
- `man SWISH-RUN`
- `man SWISH-FAQ`
- `man SWISH-LIBRARY`
- `perldoc SWISH::API`

Exercises

These exercises demonstrate how to create and search simple indexes of XML documents. The exercises build on earlier exercises in this workbook by using the data created in those exercises.

Indexing XHTML

Swish-e excels at indexing rich and well-structured HTML files. In a previous exercise sets of XHTML files were created from MARC records. They were saved in the `xml-data/xhtml/marc2xhtml` directory. Open any one of these files in your text editor and notice the structure of each of their head elements, such as this one (line have been hard-wrapped for readability):

```
<head>
  <title>Biology, psychology, and medicine</title>
  <meta name="id"          content="adler-biology-1072276585" />
```

```
<meta name="brief"      content="Biology, psychology, and medicine,  
                        by Moritmer J. Adler and V. J. McGill. Pref.  
                        by Franz Alexander."/>  
<meta name="author"     content="Adler, Mortimer Jerome, 1902-"/>  
<meta name="year"       content="1963"/>  
<meta name="title"      content="Biology, psychology, and medicine"/>  
<meta name="publisher"  content="Chicago, Encyclopedia Britannica [1963]"/>  
<meta name="pagination" content="xx, 395 p. illus. 22 cm."/>  
<meta name="note"       content="Bibliography: p. 385-395."/>  
<meta name="subject"    content="Biology Outlines, syllabi, etc.  
                        Psychology Outlines, syllabi, etc. Mind and body."/>  
</head>
```

Each meta element is comprised of name and content attributes. By configuring the swish-e indexing process to look at these attribute pairs you can make each of them field searchable. When this functionality is combined with the free text indexing against the document's body element swish-e indexes become very useful.

Swish-e supports many command-line arguments for indexing, but it is usually much easier to write a configuration file instead. Below is a configuration file (swish-indexes/xhtml.cfg) for indexing the content of the marc2xhtml directory:

```
IndexDir xml-data/xhtml/marc2xhtml  
IndexFile swish-indexes/xhtml.idx  
IndexOnly .html  
MetaNames id brief author year title publisher pagination note subject  
PropertyNames id brief author year title publisher pagination note subject
```

Each line denotes a characteristic of the indexing process:

- where is the data (the marc2xhtml directory)
- what is the location and name of the resulting index (xhtml.idx)
- what files should be indexes (only .html files)
- what meta data fields should be indexed (all of them)
- what meta data fields should be available for display (all of them)

Here's how to create your first index:

1. Edit the path statements in xhtml.cfg to suit your operating system. For example, on Windows you might have to change xml-data/xhtml/marc2xhtml to xml-data\xhtml\marc2xhtml.
2. Open a terminal session on your computer, change to the root level of the workshop's distribution directory and run swish-e with this command: **swish-e -c swish-indexes/xhtml.cfg** .

```
$ swish-e -c swish-indexes/xhtml.cfg
Indexing Data Source: "File-System"
Indexing "xml-data/xhtml/marc2xhtml"
Removing very common words...
no words removed.
Writing main index...
Sorting words ...
Sorting 10,687 words alphabetically
Writing header ...
Writing index entries ...
  Writing word text: Complete
  Writing word hash: Complete
  Writing word data: Complete
10,687 unique words indexed.
13 properties sorted.
567 files indexed.  996,001 total bytes.  112,581 total words.
Elapsed time: 00:00:04 CPU time: 00:00:04
Indexing done!
```

You can now search your newly created index. For example, search for origami: **swish-e -f swish-indexes/xhtml.idx -w origami**. This particular example only has two parts:

- -f denotes what index to search
- -w denotes the query

The search results will look a lot like this:

```
$ swish-e -f swish-indexes/xhtml.idx -w origami
# SWISH format: 2.4.2
# Search words: origami
# Removed stopwords:
# Number of hits: 7
# Search time: 0.002 seconds
# Run time: 0.046 seconds
1000 xml-data/xhtml/marc2xhtml/kawai-colorful-1071929621.html "Colorful origami" 1
962 xml-data/xhtml/marc2xhtml/gross-origami-1071930127.html "Origami" 1939
962 xml-data/xhtml/marc2xhtml/montroll-origami-1071930205.html "Origami sea life"
916 xml-data/xhtml/marc2xhtml/lang-complete-1071929822.html "Complete book of orig
916 xml-data/xhtml/marc2xhtml/honda-world-1071930499.html "World of origami" 1369
916 xml-data/xhtml/marc2xhtml/engel-folding-1072054962.html "Folding the universe"
866 xml-data/xhtml/marc2xhtml/honda-how-1071929753.html "How to make origami" 1496
```

The last few lines of the output are divided into four parts:

1. the relevance score
2. the pointer to the file matching the query
3. the title of the file
4. the size of the file

If you have Lynx, a text-based browser, then you should be able to run Lynx and point it to one of the files in the results like this: **lynx xml-data/xhtml/marc2xhtml/honda-how-1071929753.html** . Alternatively you could open any of the files in your graphical browser.

Swish-e supports a bevy of search operations including all the expected Boolean operations, phrase searching, field searching, right-hand truncation (implemented using a "*"), and nested queries. Consequently, all of the following swish-e queries are valid:

- **swish-e -f swish-indexes/xhtml.idx -w origami**
- **swish-e -f swish-indexes/xhtml.idx -w origami and colorful**
- **swish-e -f swish-indexes/xhtml.idx -w "colorful origami"**
- **swish-e -f swish-indexes/xhtml.idx -w title=origami**
- **swish-e -f swish-indexes/xhtml.idx -w author=honda**

Using the -p command line option you can alter the output to include the properties denoted in your configuration file. Thus the following command, **swish-e -f swish-indexes/xhtml.idx -w author=honda -p pagination** , will display the pagination as well as the default information (again, lines have been hard-wrapped for readability):

```
$ swish-e -f swish-indexes/xhtml.idx -w author=honda -p pagination
# SWISH format: 2.4.2
# Search words: author=honda
# Removed stopwords:
# Number of hits: 2
# Search time: 0.001 seconds
# Run time: 0.045 seconds
1000 xml-data/xhtml/marc2xhtml/honda-world-1071930499.html "World of origami" 1369
    "xii, 13-264 p. illus. (some col.) 31 cm."
1000 xml-data/xhtml/marc2xhtml/honda-how-1071929753.html "How to make origami" 149
    "37 p. col. illus. (part fold. mounted) 26 cm."
```

Indexing other XML formats

Indexing and searching other XML formats is very similar to indexing XHTML. First you create your content. Then you write a swish-e configuration file. Third, you index content. Fourth, you search. Last, you retrieve while optionally transforming your XML for display.

Let's index a set of MODS files. Open any of the MODS files in the xml-data/mods/many directory. Take note of the element names. Open swish-indexes/mods.cfg and take note of the MetaNames and PropertyNames directives. Notice how swishtitle has been added to the MetaNames directive. Swishtitle is a meta data value you get for free with swish-e, but in order for it to be searchable in your index you sometimes need to specifically include it in the configuration file. Index the MODS data with the command **swish-e -c swish-indexes/mods.cfg** , and give the following searches a whirl:

- **swish-e -f swish-indexes/mods.idx -w title=cannery**
- **swish-e -f swish-indexes/mods.idx -w love -p title**

- **swish-e -f swish-indexes/mods.idx -w origami -p swishtitle**

Like the output of the XHTML searching process, swish-e returns a pointer (file name) to documents matching your query. Unfortunately, XML files are not always very human readable, but using XSL you can transform the documents into something else. Consequently, you could manually pass parts of search results to xsltproc with a stylesheet, transform the document, and view it with something like this: **xsltproc xslt/mods2xhtml-nosave.xsl xml-data/mods/many/piper-folk-1074964323.xml > results.html; lynx results.html**

Swish-e can search multiple files with a single query by designating multiple index locations with the -f option. This is increasingly useful if the multiple files have similar MetaNames and PropertyNames values. Index the TEI data with the following command: **swish-e -c swish-indexes/tei.cfg** . Then search the index. Your queries can be much richer since the TEI files contain much more data:

- **swish-e -f swish-indexes/tei.idx -w love**
- **swish-e -f swish-indexes/tei.idx -w love and war**
- **swish-e -f swish-indexes/tei.idx -w love and war and art**
- **swish-e -f swish-indexes/tei.idx -w love and war and art and science**

Additional use of the -f and -p options produces broader results:

- **swish-e -f swish-indexes/tei.idx swish-indexes/xhtml.idx swish-indexes/mods.idx -w love and -p title**
- **swish-e -f swish-indexes/tei.idx swish-indexes/xhtml.idx swish-indexes/mods.idx -w art and science -p title**

As an extra exercise, index the set of "broken" EAD files using the ead.cfg file, and then search the resulting index while displaying the "scopecontent" in the output.

Indexing techniques and the use of relational databases are two sides of the same information retrieval coin. Relational databases are great tools especially for editing and maintaining data. While search is part of their equation, it is incumbered by complicated syntax and the lack of easy full text queries as well as relevance ranking. Indexing techniques, such as the ones implemented by swish-e, make search easy at the expense the inability to update the underlying data. By learning to combine the strengths of both relational database applications with indexes information providers can facilitate more powerful information retrieval systems.

Chapter 18. Apache

About Apache

Apache is the most popular Web (HTTP) server on the Internet and a standard open source piece of software. Its name doesn't really have anything to do with American Indians. Instead, its name comes from the way it is built. It is "a patchy" server, meaning that it is made up of many modular parts to create a coherent whole. This design philosophy has made the application very extensible. For example, there are the core modules that make up the server's ability to listen for connections, retrieve files, and return them to the requesting client (the "user agent" in HTTP parlance). There are other modules dealing with logging transactions and CGI (common gateway interface) scripting. Other modules allow you to rewrite incoming requests, manage email, implement the little-used HTTP PUT method, write other modules in Perl, or transform XML files using XSLT. Apache is currently at version 2.0, but for some reason many people are still using the 1.3 series.

Installing Apache is similar to installing Perl or Swish. For Windows you:

1. download the distribution
2. run the installer
3. open a command prompt
4. navigate to the Apache directory
5. start the server with `c:\apache\apache.exe --standalone`

On Unix/Linux you:

1. download the distribution
2. unzip it
3. untar it
4. `./configure`
5. `make`
6. `sudo make install`
7. start the server (`/usr/local/apache/bin/apachectl start`)

Being "a patchy" server, Apache is very extensible. `Mod_perl` is such an extension. It allows Perl programmers to write scripts to handle low level HTTP server functions. One of these scripts (actually a set of scripts) is called `AxKit`, an XSLT transformation engine. `AxKit` allows content providers to save XML files on their servers, have them associated with XSLT stylesheets, and have the XML files transformed on the fly by the stylesheets before they are sent to the user-agent for display.

Installing `mod_perl` and `AxKit` are beyond the scope of this workbook, but the author's water collection has been implemented as an `AxKit` service. The entire user end of the system is made up of two files (an XML data file and an XSLT stylesheet) plus a set of JPEG images. No CGI scripts. As the Apache server receives requests the query strings are read and passed on to the stylesheet. The stylesheet reads the

name/value pairs of the query string, branches accordingly, and transforms parts of the XML data into HTML, which is returned to the Web browser.

CGI interfaces to XML indexes

In a previous section of the workbook you learned how to create and search simple indexes of XML files. In this section various techniques for making these indexes searchable on the Web will be demonstrated.

Simple XHTML files

Swish-e comes with a Perl module called `SWISH::API` providing a means to search swish-e indexes through Perl scripts and without the need of the swish-e binary. By combining the standard Perl CGI module with `SWISH::API` it you can make your swish-e indexes available on the Web. The program `cgi-bin/xhtml.cgi` implements this idea. The heart of the program is a subroutine called `search`:

```
# the heart of the matter
sub search {

    # get the result page
    $html .= &results;

    # open the index
    my $swish = SWISH::API->new($INDEX);

    # create a search object
    my $search = $swish->New_Search_Object;

    # set the sort order
    $search->SetSort('title');

    # search
    my $results = $search->Execute($input->param('query'));

    # get the number of titles found
    $number_of_hits = $results->Hits;

    # initialize the hit list
    $hit_list .= '<ol>';

    # process each hit
    while (my $result = $results->NextResult) {

        # get the results
        my $detail = $result->Property('swishdocpath');
        my $brief = $result->Property('brief');

        $hit_list .= "<li>$brief (<a href='$detail'>detailed view</a></li>";

    }

    # finish the hit list
    $hit_list .= '</ol>';

}
```

It works by:

1. updating a globally defined HTML scalar
2. opening the index (\$INDEX) previously defined
3. creating a swish-e search object
4. setting the sort order
5. reading the query from the query string and searches
6. getting the number of returned hits
7. initializing a hit list
8. looping through each hit pulling out the desired properties for display
9. closing the hits list

The trickiest part of the routine is the definition of \$detail. This variable holds the path to the found item (swishdocpath). By creating a hot link using this variable the user of the index is able to view the full record.

To see how this works in real life:

1. Install Apache.
2. Edit Apache's httpd.conf file.
3. Start Apache.
4. Copy the xml-data/xhtml/marc2xhtml directory to Apache's htdocs directory.
5. Copy cgi-bin/xhtml.cgi to the htdocs directory.
6. Copy swish-indexes/xhtml.cfg to the htdocs directory.
7. Edit xhtml.cfg so IndexDir points to marc2xhtml (IndexDir marc2xhtml).
8. Edit xhtml.cfg so IndexFile outputs to xhtml.idx (IndexFile xhtml.idx).
9. Index marc2xhtml (**swish-e -c xhtml.cfg**).
10. Edit xhtml.cgi's first line so it points to Perl (Unix: `#!/usr/bin/perl` or Windows: `#!c:\perl\bin\perl`).
11. Open a command prompt and run xhtml.cgi from the command line. It should return a stream of XHTML.
12. Finally, open your Web browser and point it to xhtml.cgi.

(If you have gotten this far, the breath a sigh of relief. That was the most complicated exercise, and it is all downhill from here.)

Improving display of search results

The search results of the XHTML index were rudimentary. In this exercise you will see how the swish-e properties (and therefore XML elements) can be displayed in greater detail. Mods.cgi implements this

idea. It is just like the previous example except for part of the while loop in the search subroutine:

```
# process each hit
while (my $result = $results->NextResult) {

    # get the results
    my $detail    = $result->Property ('swishdocpath');
    my $title     = $result->Property ('title');
    my $author    = $result->Property ('namePart');
    my $extent    = $result->Property ('extent');
    my $note      = $result->Property ('note');
    my $topic     = $result->Property ('topic');
    my $publisher = $result->Property ('publisher');
    my $date      = $result->Property ('dateIssued');

    $hit_list .= "<li>$title (<a href='$detail'>full record</a>)
        <ul>
            <li><small><b>author</b>: $author</small></li>
            <li><small><b>extent</b>: $extent</small></li>
            <li><small><b>note</b>: $note</small></li>
            <li><small><b>topic</b>: $topic</small></li>
            <li><small><b>publisher</b>: $publisher</small></li>
            <li><small><b>date</b>: $date</small></li>
        </ul>
        <br />
    </li>";
}
```

As you can see, many of the index's properties are read from each result. These properties are then marked up in a number of unordered lists. Like before, the swishdocpath property is hyperlinked to provide access to the original document.

Give this script a whirl by:

1. Copying the xml-data/mods/many directory to Apache's htdocs directory.
2. Copy cgi-bin/mods.cgi to the htdocs directory.
3. Copy swish-indexes/mods.cfg to the htdocs directory.
4. Edit mods.cfg so IndexDir points to many (IndexDir many).
5. Edit mods.cfg so IndexFile outputs to mods.idx (IndexFile mods.idx).
6. Index the contents of many (**swish-e -c mods.cfg**).
7. Edit mods.cgi's first line so it points to Perl.
8. Open a command prompt and run mods.cgi from the command line. It should return a stream of XHTML.
9. Finally, open your Web browser and point it to mods.cgi.

Modern browsers and XML search results

Notice how the links in the previous exercise returned raw XML. This is because no stylesheet information was associated with the XML. This can be overcome by inserting an XML processing instruction into each XML file and indexing the files. Using the "fixed" EAD data from a previous exercise you can see this work accomplished through a CGI script called `cgi-bin/ead.cgi`:

1. Copy the `xml-data/ead/fixed` directory to Apache's `htdocs` directory.
2. Copy `cgi-bin/ead.cgi` to the `htdocs` directory.
3. Copy `swish-indexes/ead.cfg` to the `htdocs` directory.
4. Edit `ead.cfg` so `IndexDir` points to `many` (`IndexDir fixed`).
5. Edit `ead.cfg` so `IndexFile` outputs to `ead.idx` (`IndexFile ead.idx`).
6. Index the contents of `fixed` (**swish-e -c ead.cfg**).
7. Edit `ead.cgi`'s first line so it points to Perl.
8. Open a command prompt and run `ead.cgi` from the command line. It should return a stream of XHTML.
9. Finally, open a modern Web browser and point it to `ead.cgi`.

Transforming raw XML on the fly

You can not always rely on the end user having a modern browser. This is why it might be necessary to transform your raw XML into HTML before it is sent to the user-agent. This can be done by inserting a second CGI script into the hot link of the full document. This second script is really `xsltproc.pl` wrapped in a CGI interface. Open `cgi-bin/xsltproc.cgi` to see.

`Xsltproc.cgi` is incorporated into `tei.cgi`'s search results loop:

```
# process each hit
while (my $result = $results->NextResult) {

    # get the results
    my $detail = $result->Property ('swishdocpath');
    my $title  = $result->Property ('title');
    my $author = $result->Property ('author');
    $hit_list .= "<li>$author / $title
                  (<a href='./$XSLTPROC?xslt=$STYLE&xml=$detail'>full record</a>)</li>"
}
}
```

In this loop:

a few properties are read from the index

the hitlist is hotlinked to the full record using the global variables `$XSLTPROC` and `$STYLE`

Here's how to make it work:

1. Copy the xml-data/tei directory to Apache's htdocs directory.
2. Copy cgi-bin/xsltproc.cgi to the htdocs directory.
3. Copy cgi-bin/tei.cgi to the htdocs directory.
4. Copy swish-indexes/tei.cfg to the htdocs directory.
5. Edit tei.cfg so IndexDir points to tei (IndexDir tei).
6. Edit tei.cfg so IndexFile outputs to tei.idx (IndexFile tei.idx).
7. Index the contents of tei (**swish-e -c tei.cfg**).
8. Edit tei.cgi's first line so it points to Perl.
9. Edit xsltproc.cgi's first line so it points to Perl.
10. Open a command prompt and run tei.cgi from the command line. It should return a stream of XHTML.
11. Finally, open any Web browser and point it to tei.cgi.

If all goes well you should see pointers to the full text of your documents as well as be able to read the full texts on your screen.

Whew!

Part V. Appendices

Table of Contents

| | |
|---|-----|
| A. Harvesting metadata with OAI-PMH | 112 |
| What is the Open Archives Initiative? | 112 |
| The Problem | 112 |
| The Solution | 113 |
| Verbs | 113 |
| Responses -- the XML stream | 115 |
| An Example | 117 |
| Exercise - Making CIMI Schema data available via OAI-PMH | 118 |
| Exercise - Making MARCXML data available via OAI | 119 |
| Conclusion | 120 |
| B. An Introduction to the Search/Retrieve URL Service (SRU) | 121 |
| Introduction | 121 |
| The Problems SRW and SRU are Intended to Solve | 121 |
| SRW and SRU as Web Services | 122 |
| Explain | 122 |
| Scan | 123 |
| SearchRetrieve | 124 |
| A Sample Application: Journal Locator | 125 |
| SRW/U and OAI-PMH | 129 |
| OCKHAM | 129 |
| Summary | 131 |
| Acknowledgements | 131 |
| References | 131 |
| C. Selected readings | 132 |
| XML in general | 132 |
| Cascading Style Sheets | 132 |
| XSLT | 132 |
| DocBook | 132 |
| XHTML | 133 |
| RDF | 133 |
| EAD | 133 |
| TEI | 133 |
| OAI-PMH | 133 |

Appendix A. 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:

```
ht-  
tp://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-pri
```

nce-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:

```
ht-  
tp://www.infomotions.com/alex/oai/?verb=GetRecord&metadataPrefix=oai_dc&identifier=twain-n  
ew-36  
[http://www.infomotions.com/alex/oai/?verb=GetRecord&metadataPrefix=oai_dc&identifier=twai  
n-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, metadataNamespace. 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>
```

```
</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 examples, 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&metadataFormat=OAI-PMH</requestURL>

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

    <header>
      <identifier>twain-tom-40</identifier>
      <timestamp>1999</timestamp>
    </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-/tw
<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=selec
<relation>http://www.infomotions.com/alex/cgi-bin/configure-ebook.pl?handl
<relation>http://www.infomotions.com/alex/cgi-bin/pdf.pl?handle=twain-tom-
<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>';  
  
?>
```

Exercise - Making CIMI Schema data available via OAI-PMH

Given the necessary computer infrastructure, in this exercise you will make the data in from a small CIMI Schema XML for SPECTRUM file available via OAI-PMH.

A. Transform CIMI data into files

1. Open the file named water.xml in your favorite text editor, such as Notepad. The schema (CIMI Schema for SPECTRUM) used to define the contents of water.xml is designed to describe, inventory, and chronical the items in museum collections. Notice the structure of the file as well as the elements used to mark-up its content.
2. Create a directory on your computer's desktop.
3. Copy all the *.dll files from the CD to your newly created directory.
4. Copy all the *.exe files from the CD to your newly created directory.
5. Copy cimi2oaide.xsl and water.xml from the CD to your newly created directory.
6. Open a new terminal window by running cmd.exe from the Start menu's Run command.
7. Change directories to your newly created directory.
8. Transform water.xml into a set of XML files very easily read by OAI-PMH harvesters by using this command: **xsltproc cimi2oaide.xsl water.xml**.
9. Open any of the newly created files in your favorite text editor and notice how it contains Dublin Core metadata transformed from the original CIMI data.

B. Install OAI scripts

1. Assuming that Perl, a scripting language, is already installed on your computer, copy the directory named oai from the CD to your newly created directory on the desktop.
2. Copy all the newly created XML files from the first part of this exercise into the directory named data found in the newly created oai directory.
3. Change directories to the newly created CIMI directory, open the file named config.xml in your text editor, change the value of adminEmail, and save the changes.

4. Run the Perl script named `oai.pl` and give the script input such as `verb=Identify`, `verb=ListSets`, `verb=GetRecord&identifier=oai:water:brides-bay&metadataPrefix=oai_dc`, or `verb=ListRecords&metadataPrefix=oai_dc`. The output should be standard OAI-PMH XML streams. You're more than half way there.

C. Make scripts available via the Web

1. Assuming a Web server is installed on your computer, copy the `oai` directory to a place where its contents can be read by the server.
2. Open your Web browser and try opening a connection to the `oai.pl` script. The URL will look something like this: `http://www.example.edu/cgi-bin/oai/oai.pl`.
3. Try additional URLs but this time include the verbs from the second part of this exercise, above. You should see good o' XML streams in the form of OAI-PMH responses.
4. Use the Open Archives Initiative - Repository Explorer at `http://oai.dlib.vt.edu/cgi-bin/Explorer/oai2.0/testoai` to test your newly created OAI repository, and congratulations, you have made a set of data available via OAI-PMH.

Exercise - Making MARCXML data available via OAI

As an extra exercise transform MARCXML data into simple OAI files and make them available via the OAI protocol. This exercise is based on the previous exercise.

1. Use your text editor to open `xslt/MARC21slim2OAIDC.xsl`. This stylesheet was created by the good folks at the Library of Congress.
2. Use `xsltproc` to use the stylesheet to transform a previously created MARCXML file: `xsltproc xslt/MARC21slim2OAIDC.xsl xml-data/marc/many/milne-when-1071456632.xml`. The output will be a simple OAI stream in Dublin Core data.
3. Use `xsltproc` to save a number of file transformations to the data directory of the `oai` directory of the previous exercise. For example:
 - `xsltproc xslt/MARC21slim2OAIDC.xsl xml-data/marc/many/milne-when-1071456632.xml > /oai/data/milne-when-1071456632.xml`
 - `xsltproc xslt/MARC21slim2OAIDC.xsl xml-data/marc/many/mumford-herman-1072061308.xml > /oai/data/mumford-herman-1072061308.xml`
 - `xsltproc xslt/MARC21slim2OAIDC.xsl xml-data/marc/many/feiler-making-1074959890.xml > /oai/data/feiler-making-1074959890.xml`
 - `xsltproc xslt/MARC21slim2OAIDC.xsl xml-data/marc/many/willinger-red-1072543737.xml > /oai/data/willinger-red-1072543737.xml`

4. Use the Repository Explorer from the previous exercise to see your newly created data.

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".

Appendix B. An Introduction to the Search/Retrieve URL Service (SRU)



Note: This is a pre-edited version of an article appearing in Ariadne. It describes sibling Web Service protocols designed to define a standard form for Internet search queries as well as the structure of the responses.

Introduction

This article is an introduction to the "brother and sister" Web Service protocols named Search/Retrieve Web Service (SRW) and Search/Retrieve URL Service (SRU) with an emphasis on the later. More specifically, the article outlines the problems SRW/U are intended to solve, the similarities and differences between SRW and SRU, the complimentary nature of the protocols with OAI-PMH, and how SRU is being employed in a sponsored NSF (National Science Foundation) grant called OCKHAM to facilitate an alerting service. The article is seasoned with a bit of XML and Perl code to illustrate the points. The canonical home page describing SRW/U [1] is also a useful starting point.

The Problems SRW and SRU are Intended to Solve

SRW and SRU are intended to define a standard form for Internet search queries as well as the structure of the responses. The shape of existing queries illustrates the problem. The following URLs are searches for 'dogs and cats' against three popular Internet search engines:

- <http://www.google.com/search?hl=en&ie=ISO-8859-1&q=dogs+and+cats&btnG=Google+Search>
- <http://search.yahoo.com/search?fr=fp-pull-web-t&p=dogs+and+cats>
- <http://search.msn.com/results.aspx?FORM=MSNH&q=dogs%20and%20cats>

Even though the queries are the same, the syntax implementing the queries is different. What is worse is the structure of the responses. Each response not only contains search results but lots of formatting as well. SRW and SRU address these shortcomings by specifying the syntax for queries and results. Such specifications open up Internet-accessible search functions and allow for the creation of tools to explore the content of the 'hidden Web' more effectively. SRW/U allow people and HTTP user agents to query

Internet databases more seamlessly without the need of more expensive and complicated meta-search protocols.

SRW and SRU as Web Services

SRW/U are Web Services-based protocols for querying Internet indexes or databases and returning search results. Web Services essentially come in two flavours: REST (Representational State Transfer) and SOAP (Simple Object Access Protocol). A "REST-ful" Web Service usually encodes commands from a client to a server in the query string of a URL. Each name/value pair of the query string specifies a set of input parameters for the server. Once received, the server parses these name/value pairs, does some processing using them as input, and returns the results as an XML stream. The shape of the query string as well as the shape of the XML stream are dictated by the protocol. By definition, the communication process between the client and the server is facilitated over an HTTP connection.

OAI-PMH is an excellent example of a REST-ful Web Service. While OAI 'verbs' can be encoded as POST requests in an HTTP header, they are usually implemented as GET requests. These verbs act as commands for the OAI data repository which responds to them with a specific XML schema. "SOAP-ful" Web Services work in a similar manner, except the name/value pairs of SOAP requests are encoded in an XML SOAP 'envelope'. Similarly, SOAP servers return responses using the SOAP XML vocabulary. The biggest difference between REST-ful Web Services and SOAP requests is the transport mechanism. REST-ful Web Services are always transmitted via HTTP. SOAP requests and responses can be used by many other transport mechanisms including email, SSH (Secure Shell), telnet, as well as HTTP.

Web Services essentially send requests for information from a client to a server. The server reads the input, processes it, and returns the results as an XML stream back to the client. REST-ful Web Services encode the input usually in the shape of URLs. SOAP requests are marked up in a SOAP XML vocabulary. REST-ful Web Services return XML streams of varying shapes. SOAP Web Services return SOAP XML streams. Many people think implementing REST-ful Web Service is easier because it is simpler to implement and requires less overhead. SOAP is more robust and can be implemented in a larger number of networked environments.

SRW is a SOAP-ful Web Service. SRU is a REST-ful Web service. Despite the differences in implementation, they are really very similar since they both define a similar set of commands (known as "operations") and responses. Where OAI-PMH defines six 'verbs', SRW/U support three 'operations': explain, scan, and searchRetrieve. Like OAI, each operation is qualified with one or more additional name/value pairs.

Explain

Explain operations are requests sent by clients as a way of learning about the server's database/index as well as its functionality. At a minimum, responses to explain operations return the location of the database, a description of what the database contains, and what features of the protocol the server supports. Implemented in SRU, empty query strings on a URL are interpreted as an explain operation. When explicitly stated, a version parameter must be present. Therefore, at a minimum, explain operations can be implemented in one of two ways:

- `http://example.org/`
- `http://example.org/?operation=explain&version=1.1`

An example SRU response from an explain operation might look like the text below. It denotes:

- the server supports version 1.1 of the protocol

- records in this response are encoded in a specific DTD and records are encoded as XML within the entire XML stream
- the server can be found at a specific location
- the database is (very) briefly described
- the database supports only title searching
- the database returns records using Dublin Core
- the system will return no more than 9999 records at one time

```
<explainResponse>
<version>1.1</version>
  <record>
    <recordSchema>http://explain.z3950.org/dtd/2.0</recordSchema>
    <recordPacking>xml</recordPacking>
    <recordData>
      <explain>
        <serverInfo>
          <host>example.org</host>
          <port>80</port>
          <database>/</database>
        </serverInfo>
        <databaseInfo>
          <title>An example SRU service</title>
          <description lang='en' primary='true'>
            This is an example SRU service.
          </description>
        </databaseInfo>
        <indexInfo>
          <set identifier='info:srw/cql-context-set/1/dc-v1.1' name='dc' />
          <index>
            <title>title</title>
            <map>
              <name set='dc'>title</name>
            </map>
          </index>
        </indexInfo>
        <schemaInfo>
          <schema identifier='info:srw/schema/1/dc-v1.1'
            sort='false' name='dc'>
            <title>Dublin Core</title>
          </schema>
        </schemaInfo>
        <configInfo>
          <default type='numberOfRecords'>9999</default>
        </configInfo>
      </explain>
    </recordData>
  </record>
</explainResponse>
</section>
```

Scan

Scan operations list and enumerate the terms found in the remote database's index. Clients send scan requests and servers return lists of terms. The process is akin to browsing a back-of-the-book index where a person looks up a term in the back of a book and 'scans' the entries surrounding the term. At a minimum, scan operations must include a scan clause (scanClause) and a version number parameter. The scan clause contains the term to look for in the index. A rudimentary request and response follow:

<http://example.org/?operation=scan&scanClause=dog&version=1.1>

```
<scanResponse>
  <version>1.1</version>
  <terms>
    <term>
      <value>doesn't</value>
      <numberOfRecords>1</numberOfRecords>
    </term>
    <term>
      <value>dog</value>
      <numberOfRecords>1</numberOfRecords>
    </term>
    <term>
      <value>dogs</value>
      <numberOfRecords>2</numberOfRecords>
    </term>
  </terms>
</scanResponse>
</section>
```

SearchRetrieve

SearchRetrieve operations are the heart of the matter. They provide the means to query the remote database and return search results. Queries must be articulated using the Common Query Language (CQL) [2]. These queries can range from simple free text searches to complex Boolean operations with nested queries and proximity qualifications.

Servers do not have to implement every aspect of CQL, but they have to know how to return diagnostic messages when something is requested but not supported. The results of searchRetrieve operations can be returned in any number of formats, as specified via explain operations. Examples might include structured but plain text streams or data marked up in XML vocabularies such as Dublin Core, MARCXML, MODS (Metadata Object Description Schema), etc. Below is a simple request for documents matching the free text query 'dogs':

<http://example.org/?operation=searchRetrieve&query=dog&version=1.1>

In this case, the server returns three (3) hits and by default includes Dublin Core title and identifier elements. The record itself is marked up in some flavor of XML as opposed to being encapsulated as a string embedded the XML:

```
<searchRetrieveResponse>
  <version>1.1</version>
  <numberOfRecords>3</numberOfRecords>
  <records>
    <record>
      <recordSchema>info:srw/schema/1/dc-v1.1</recordSchema>
```

```
<recordPacking>xml</recordPacking>
<recordData>
  <dc>
    <title>The bottom dog</title>
    <identifier>http://example.org/bottom.html</identifier>
  </dc>
</recordData>
</record>
<record>
  <recordSchema>info:srw/schema/1/dc-v1.1</recordSchema>
  <recordPacking>xml</recordPacking>
  <recordData>
    <dc>
      <title>Dog world</title>
      <identifier>http://example.org/dog.html</identifier>
    </dc>
  </recordData>
</record>
<record>
  <recordSchema>info:srw/schema/1/dc-v1.1</recordSchema>
  <recordPacking>xml</recordPacking>
  <recordData>
    <dc>
      <title>My Life as a Dog</title>
      <identifier>http://example.org/my.html</identifier>
    </dc>
  </recordData>
</record>
</records>
</searchRetrieveResponse>
```

A Sample Application: Journal Locator

In an attempt to learn more about SRU, the author created a simple SRU interface to an index of journal titles, holdings, and locations. Like many academic libraries, the University Libraries of Notre Dame subscribe to physical and electronic journals. Many of the electronic journals are accessible through aggregated indexes such as Ebscohost Academic Search Elite. Since the content of these aggregated indexes is in a constant state of flux, it is notoriously difficult to use traditional catalogueuing techniques to describe journal holdings. Consequently, the Libraries support the finding of journal titles through its catalogue as well as through tools/services such as SFX (Special Effects) and SerialsSolutions. Unfortunately, when patrons ask the question "Does the library have access to journal...?", they need to consult two indexes: the catalogue and an interface to SFX.

Journal Locator is an example application intended to resolve this problem by combining the holdings in the catalogue with the holdings in SFX into a single search interface. By searching this combined index, patrons are presented with a list of journal titles, holding statements, and locations where the titles can be found. The whole thing is analogous to those large computer printouts created in the early to mid-1980s listing a library's journal holdings. Here is the process for creating Journal Locator:

1. dump sets of MARC records encoded as serials from the catalogue
2. transform the MARC records into sets of simple XHTML files
3. dump sets of SFX records as an XML file
4. transform the XML file into more sets of simple XHTML files

5. index all the XHTML files
6. provide an SRU interface to search the index

Here at the Notre Dame we use scripts written against a Perl module called MARC::Record [3] to convert MARC data into XHTML. We use xsltproc [4] to transform XML output from SFX into more XHTML. We use swish-e [5] to index the XHTML, and finally, we use a locally written Perl script to implement an SRU interface to the index. The interface is pretty much the basic vanilla flavour, i.e. supporting only explain and searchResponse operations. It returns raw XML with an associated XSLT (Extensible Stylesheet Language Transformations) stylesheet, and consequently the interface assumes the patron is using a relatively modern browser with a built-in XSLT processor. Journal Locator is not a production service.

A rudimentary SRU explain operation returns an explain response. The response is expected to be transformed by the XSLT stylesheet specified in the output into an XHTML form. Queries submitted through the form are sent to the server as SRU searchRetrieve operations. Once the query string of the URL is parsed by the server, the search statement is passed on to a subroutine. This routine searches the index, formats the results, and returns them accordingly. An example SRU searchRetrieve request may include:

<http://dewey.library.nd.edu/morgan/sru/search.cgi?operation=searchRetrieve&query=dog&version=1.1>

Here is an abbreviated version of the search subroutine in the Perl script. Notice how it searches the index, initialises the XML output, loops through each search result, closes all the necessary elements, and returns the result:

```
sub search {  
    # get the input  
    my ($query, $style) = @_;  
  
    # open the index  
    my $swish = SWISH::API->new($INDEX);  
  
    # create a search object  
    my $search = $swish->New_Search_Object;  
  
    # do the work  
    my $results = $search->Execute($query);  
  
    # get the number of hits  
    my $hits = $results->Hits;  
  
    # begin formatting the response  
    my $response = "<?xml version='1.0' ?>\n";  
    $response .= "<?xml-stylesheet type='text/xsl' href='$style' ?>\n";  
    $response .= "<searchRetrieveResponse>\n";  
    $response .= "<version>1.1</version>\n";  
    $response .= "<numberOfRecords>$hits</numberOfRecords>\n";  
  
    # check for hits  
    if ($hits) {  
        # process each found record  
        $response .= "<records>\n";  
        my $p = 0;  
        while (my $record = $results->NextResult) {  
            $response .= "<record>\n";  
            $response .= "<recordSchema>" .  

```

```
        "info:srw/schema/1/dc-v1.1</recordSchema>\n";
$response .= "<recordPacking>xml</recordPacking>\n";
$response .= "<recordData>\n";
$response .= "<dc>\n";
$response .= "<title>" .
            &escape_entities($record->Property('title')) .
            "</title>\n";

# check for and process uri
if ($record->Property ('url')) {

    $response .= "<identifier>" .
                &escape_entities($record->Property('url')) .
                "</identifier>\n"

}

# get and process holdings
my $holding = $record->Property ('holding');
my @holdings = split (/\\|/, $holding);
foreach my $h (@holdings) {

    $response .= '<coverage>' . &escape_entities($h) . "</coverage>\n"

}

# clean up
$response .= "</dc>\n";
$response .= "</recordData>\n";
$response .= "</record>\n";

# increment the pointer and check
$p++;
last if ($input->param('maximumRecords') == $p);

}

# close records
$response .= "</records>\n";

}

# close response
$response .= "</searchRetrieveResponse>\n";

# return it
return $response;

}
```

The result is an XML stream looking much like this:

```
<?xml version='1.0' ?>
<?xml-stylesheet type='text/xsl' href='etc/search.xsl' ?>
<searchRetrieveResponse>
  <version>1.1</version>
  <numberOfRecords>2</numberOfRecords>
  <records>
```

```
<record>
  <recordSchema>info:srw/schema/1/dc-v1.1</recordSchema>
  <recordPacking>xml</recordPacking>
  <recordData>
    <dc>
      <title>The bottom dog</title>
      <coverage>
        Microforms [Lower Level HESB] General Collection
        Microfilm 3639 v.1:no.1 (1917:Oct. 20)-v.1:no.5
        (1917:Nov. 17)
      </coverage>
    </dc>
  </recordData>
</record>
<record>
  <recordSchema>info:srw/schema/1/dc-v1.1</recordSchema>
  <recordPacking>xml</recordPacking>
  <recordData>
    <dc>
      <title>Dog world</title>
      <identifier>
        http://sfx.nd.edu:8889/ndu_local?genre=article&sid=ND:ejl_loc&issn=0012-4893
      </identifier>
      <coverage>
        EBSCO MasterFILE Premier:Full Text
        (Availability: from 1998)
      </coverage>
    </dc>
  </recordData>
</record>
</records>
</searchRetrieveResponse>
```

A nifty feature of SRW/U is the optional specification of an XSLT stylesheet by the user agent for transforming the XML output. If a stylesheet is not specified, then the server can specify a stylesheet. This is how the Journal Locator is implemented. The XML is returned and transformed using the stylesheet defined in the second line of the output, the XML processing instruction pointing to 'etc/search.xsl'. Here is the part of that stylesheet rendering the XML into an ordered list of titles, holdings, and locations:

```
<ol>
  <xsl:for-each select='//dc'>
    <li>
      <xsl:value-of select='title' />
      <ul>
        <xsl:for-each select='coverage'>
          <xsl:choose>
            <xsl:when test='../identifier'>
              <li class='holding'>
                <a>
                  <xsl:attribute name='href'>
                    <xsl:value-of select='../identifier' />
                  </xsl:attribute>
                  <xsl:value-of select='.' />
                </a>
              </li>
            </xsl:when>
            <xsl:otherwise>
```



```
<li class='holding'>
  <xsl:value-of select='.' />
</li>
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</ul>
</li>
</xsl:for-each>
</ol>
```

Another nifty feature of SRW/U is the use of 'extra data parameters'. These parameters, always prefixed with x- in an SRU URL, allow implementers to add additional functionality to their applications. The author has used this option to create an x-suggest feature. By turning 'x-suggest' on (x-suggest=1), the system will examine the number of hits returned from a query and attempt to suggest additional searches ready for execution. For example, if the number of hits returned from a search is zero (0), then the application will create alternative searches analogous to Google's popular Did You Mean? service by looking up the user's search terms in a dictionary. If the number of hits is greater than twenty-five (25), then the application will help users limit their search by suggesting alternative searches such as title searches or phrase searches.

The next steps for Journal Locator are ambiguous. On one hand the integrated library system may be able to support this functionality some time soon, but the solution will be expensive and quite likely not exactly what we desire. On the other hand, a locally written solution will cost less in terms of cash outlays, but ongoing support may be an issue. In any event, Journal Locator provided a suitable venue for SRU exploration.

SRW/U and OAI-PMH

SRW/U and OAI-PMH are complementary protocols. They have similar goals, namely, the retrieval of metadata from remote hosts, but each provides functionality that the other does not. Both protocols have similar 'about' functions. SRW/U's explain operation and OAI-PMH's identify verb both return characteristics describing the properties of the remote service.

Both protocols have a sort of "browse" functionality. SRW/U has its scan function and OAI-PMH has ListSets. Scan is like browsing a book's back-of-the-book index. ListSets is similar to reading a book's table of contents.

SRW/U and OAI differ the most when it comes to retrieval. SRW/U provides a much more granular approach (precision) at the expense of constructing complex CQL queries. OAI-PMH is stronger on recall allowing a person to harvest the sum total of data a repository has to offer using a combination of the ListRecords and GetRecords verbs. This is implemented at the expense of gathering unwanted information.

If a set of data were exposed via SRW/U as well as OAI-PMH, then SRW/U would be the tool to use if a person wanted to extract only data crossing predefined sets. OAI-PMH would be more apropos if the person wanted to get everything or predefined subsets of the data.

OCKHAM

There is a plan to use SRU as a means to implement an alerting service in a sponsored NSF grant called OCKHAM [6]. The OCKHAM Project is lead by Martin Halbert (Emory University), Ed Fox (Virginia Tech), Jeremy Frumkin (Oregon State), and the author. The goals of OCKHAM are three-fold:

- To articulate and draft a reference model describing digital library services
- To propose a number of light-weight protocols along the lines of OAI-PMH as a means to facilitate digital library services
- To implement a select number of digital library services exemplifying the use of the protocols

OCKHAM proposes a number of initial services to be implemented:

- a registry service
- an alerting service
- a browsing service
- a pathfinder service
- a search service
- a metadata conversion service
- a cataloguing service

Notre Dame is taking the lead in developing the alerting service -- a sort of current awareness application allowing people to be kept abreast of newly available materials on the Internet. This is how the service will work:

1. An institution (read 'library'), will create a list of OAI data repositories (URLs) containing information useful to its clientele.
2. These URLs will be fed to an OAI harvester and the harvested information will be centrally stored. Only a limited amount of information will be retained, namely information that is no older than what the hosting institution defines as 'new.'
3. One or more sets of MARC records will be harvested from library catalogues and saved to the central store as well. Again, only lists of 'new' items will be retained. As additional data is harvested the older data is removed.
4. Users will be given the opportunity to create searches against the centralised store. These searches will be saved on behalf of the user and executed on a regular basis with the results returned via email, a Web page, an RSS feed, and/or some other format.
5. Repeat.

SRU URL's will be the format of the saved searches outlined in Step 4 above. These URLs will be constructed through an interface allowing the user to qualify their searches with things like author names, titles, subject terms, free text terms or phrases, locations, and/or physical formats. By saving user profiles in the form of SRU URLs, patrons will be able to apply their profiles to other SRU-accessible indexes simply by changing the host and path specifications.

The goal is to promote the use of SRU URLs as a way of interfacing with alerting services as unambiguously and as openly as possible.

Summary

SRW and SRU are "brother and sister" standardised Web Service-based protocols for accomplishing the task of querying Internet-accessible databases and returning search results. If index providers were to expose their services via SRW and/or SRU, then the content of the 'hidden Web' would become more accessible and there would be less of a need to constantly re-invent the interfaces to these indexes.

Acknowledgements

The author would like to thank the people on the ZNG mailing list for their assistance. They were invaluable during the learning process. Special thanks goes to Ralph Levan of OCLC who helped clarify the meaning and purpose of XML packing.

References

1. The home page for SRW/U is <http://www.loc.gov/z3950/agency/zing/srw/>.
2. CQL is fully described at <http://www.loc.gov/z3950/agency/zing/cql/>.
3. The home page of MARC::Record is <http://marc.sourceforge.net/>.
4. Xsltproc is an application written against two C libraries called libxml and libxslt described at <http://xmlsoft.org/>.
5. Swish-e is an indexer/search engine application with a C as well as a Perl interface. See: <http://swish-e.org/>.
6. The canonical URL of OCKHAM is <http://ockham.org/>.

Appendix C. Selected readings

This is a short list of selected books and websites that can be used to supplement 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 if 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 canonical 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 canonical homepage for XSLT
3. XSL Tutorial [<http://www.w3schools.com/xsl/>] - Test your XSLT skill here.

DocBook

1. DocBook, 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 cononical home page for OAI-PMH.