The XML Files:
Using XML and XSL with IBM WebSphere 3.0

Luis Ennser, Christophe Chuvan, Paul Fremantle, Ramani Routray
Jouko Ruuskanen

International Technical Support Organization
www.redbooks.ibm.com
The XML Files:
Using XML and XSL with IBM WebSphere 3.0

March 2000
Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix D, “Special notices” on page 221.

First Edition (March 2000)

This edition applies to IBM WebSphere Application Server V 3.0.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2000. All rights reserved.
Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
## Contents

**Figures** ............................................................... ix

**Tables** ............................................................... xiii

**Preface** ............................................................... xv

The team that wrote this redbook ................................ xv

Comments welcome ................................................... xvii

### Part 1. Introduction to XML technology .......................... 1

**Chapter 1. Why XML?** .................................................. 3

1.1 Background .......................................................... 3

1.2 XML — a universal data format .................................... 4

1.3 XML business benefits .............................................. 5

1.3.1 Information Sharing ........................................... 6

1.3.2 XML inside a single application ................................. 6

1.3.3 Content delivery ................................................ 7

1.4 Technological benefits of XML ..................................... 7

1.4.1 An example of using XML ........................................ 7

1.4.2 Major benefits .................................................. 8

1.5 XML history .......................................................... 10

1.6 Real-life uses of XML ............................................... 11

1.6.1 SABRE and Wireless Markup Language ........................ 11

1.6.2 Chemical Markup Language ..................................... 12

**Chapter 2. XML overview** ........................................... 13

2.1 XML concepts ....................................................... 13

2.1.1 XML and SGML .................................................. 13

2.1.2 Document validity and well-formedness ........................ 14

2.1.3 Document Type Definition ..................................... 15

2.1.4 Namespaces .................................................... 20

2.1.5 DTD versus XML Schemas ...................................... 22

2.2 XML Linking .......................................................... 23

2.3 XPath ................................................................. 25

2.4 XPointer .............................................................. 26

2.4.1 Types of XPointer locators .................................... 27

2.5 eXtensible Stylesheet Language (XSL) ............................ 28

2.5.1 Cascading Style Sheets ......................................... 28

2.5.2 XSL = fo: + XSLT ................................................ 29

2.5.3 XSL Transformations ............................................ 31

2.5.4 Relationship between XSL and XPath .......................... 31
Chapter 3. Processing XML using Java ........................................... 35
  3.1 XML applications ....................................................... 35
  3.2 SAX ............................................................... 36
    3.2.1 SAX classes and interfaces .................................... 37
    3.2.2 SAX example .................................................. 38
  3.3 DOM ............................................................... 39
    3.3.1 DOM hierarchy ................................................. 40
    3.3.2 DOM example .................................................. 41
    3.3.3 DOM Level 2 ................................................... 41
  3.4 SAX or DOM? ......................................................... 42
    3.4.1 SAX advantages and disadvantages ......................... 42
    3.4.2 DOM advantages and disadvantages ......................... 42

Part 2. Applying XML technology in IBM WebSphere .................................... 43

Chapter 4. Introduction to IBM WebSphere Application Server V3 .................. 45
  4.1 XML for Java Parser ................................................ 45
  4.2 LotusXSL .......................................................... 46
  4.3 DTD Catalogs ....................................................... 46

Chapter 5. XML Parser for Java ................................................. 47
  5.1 How to parse an XML document using DOM ....................... 48
  5.2 How to create an XML document using DOM ..................... 53
  5.3 Reading and querying DTD information ........................... 53
  5.4 Using namespaces ................................................. 54

Chapter 6. LotusXSL ........................................................... 55
  6.1 So, what is an XSLT processor? ................................... 55
  6.2 XSLT basics ....................................................... 56
    6.2.1 Accessing attributes ......................................... 57
    6.2.2 Conditional processing ...................................... 58
    6.2.3 Repetition ...................................................... 59
    6.2.4 Creating attributes and elements .......................... 61
  6.3 LotusXSL API ....................................................... 62
    6.3.1 Integrating a parser with the XSL processor ............... 63
    6.3.2 DefaultApplyXSL ............................................... 63
    6.3.3 Configuring DefaultApplyXSL ................................ 63
    6.3.4 Running DefaultApplyXSL .................................... 64

Chapter 7. WebSphere and XML approaches ......................................... 65
  7.1 WebSphere programming model ...................................... 65
### Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.1</td>
<td>Servlets</td>
<td>65</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Java Server Pages</td>
<td>65</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Servlets, JSPs, and JavaBeans</td>
<td>66</td>
</tr>
<tr>
<td>7.2</td>
<td>Generating XML with WebSphere Application Server</td>
<td>66</td>
</tr>
<tr>
<td>7.2.1</td>
<td>The println method</td>
<td>67</td>
</tr>
<tr>
<td>7.2.2</td>
<td>The Document Object Model approach</td>
<td>68</td>
</tr>
<tr>
<td>7.2.3</td>
<td>The JSP method</td>
<td>70</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Comparison of methods — generating XML</td>
<td>71</td>
</tr>
<tr>
<td>7.3</td>
<td>Applying XSL to XML with WebSphere Application Server</td>
<td>72</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Using XSL within a servlet</td>
<td>72</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Using XSL automatically</td>
<td>78</td>
</tr>
<tr>
<td>7.3.3</td>
<td>XSLT Islands</td>
<td>79</td>
</tr>
<tr>
<td>7.4</td>
<td>Using XSL selectively</td>
<td>82</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Defining an alternative Web application</td>
<td>82</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Applying XSL based on browser type</td>
<td>83</td>
</tr>
<tr>
<td>7.5</td>
<td>Comparison of approaches</td>
<td>84</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Applying XSL</td>
<td>84</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Comparison of XML/XSL and Servlet/JSP</td>
<td>85</td>
</tr>
<tr>
<td><strong>Part 3.</strong></td>
<td>ITSO XML application</td>
<td>87</td>
</tr>
<tr>
<td><strong>Chapter 8.</strong></td>
<td>XML application scenario</td>
<td>89</td>
</tr>
<tr>
<td>8.1</td>
<td>Application overview</td>
<td>89</td>
</tr>
<tr>
<td>8.2</td>
<td>Database architecture</td>
<td>91</td>
</tr>
<tr>
<td>8.3</td>
<td>Document Type Definition</td>
<td>92</td>
</tr>
<tr>
<td>8.4</td>
<td>Application architecture</td>
<td>96</td>
</tr>
<tr>
<td><strong>Chapter 9.</strong></td>
<td>XML application implementation</td>
<td>97</td>
</tr>
<tr>
<td>9.1</td>
<td>Logon and user profile management</td>
<td>97</td>
</tr>
<tr>
<td>9.1.1</td>
<td>User profile database</td>
<td>97</td>
</tr>
<tr>
<td>9.1.2</td>
<td>User profile registration servlet</td>
<td>100</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Logon servlet</td>
<td>101</td>
</tr>
<tr>
<td>9.1.4</td>
<td>User interfaces</td>
<td>101</td>
</tr>
<tr>
<td>9.2</td>
<td>Alternative — XSLT Islands</td>
<td>109</td>
</tr>
<tr>
<td>9.3</td>
<td>Search engine</td>
<td>114</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Searching for products and adding to order</td>
<td>114</td>
</tr>
<tr>
<td>9.4</td>
<td>Implementation of shopping basket and ordering system</td>
<td>118</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Implementing the shopping basket as a JavaBean</td>
<td>119</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Program Flow</td>
<td>120</td>
</tr>
<tr>
<td>9.4.3</td>
<td>The orderbasket.jsp file</td>
<td>121</td>
</tr>
<tr>
<td>9.4.4</td>
<td>BasketBean implementation</td>
<td>123</td>
</tr>
<tr>
<td>9.4.5</td>
<td>XSL stylesheet implementation — basket.xsl</td>
<td>131</td>
</tr>
<tr>
<td>9.5</td>
<td>Order tracking</td>
<td>133</td>
</tr>
</tbody>
</table>
9.5.1 Servlet overview .............................................. 133
9.5.2 Displaying order reports ...................................... 134
9.5.3 Retrieving individual order information .................... 136
9.5.4 Generating the HTML output .................................. 139
9.6 Supplier .................................................................. 144

Chapter 10. Use and implementation of the DatabaseDom bean ...................................................................... 149
10.1 Use, limitations, and approach .................................. 149
  10.1.1 The template file ................................................ 149
  10.1.2 Definition of template file elements ......................... 152
  10.1.3 Example output .................................................. 154
  10.1.4 Bean methods ................................................... 155
  10.1.5 Updating the database .......................................... 155
10.2 The implementation of the DatabaseDom bean .......... 155
  10.2.1 Generating XML from the database ......................... 157
  10.2.2 Updating the database from an XML DOM tree ........ 157
  10.2.3 Creating the column to XML mapping .................... 158

Chapter 11. Conclusions .................................................. 161
11.1 Successes ............................................................ 161
11.2 Lessons Learned .................................................... 163
11.3 Areas for future exploration ...................................... 164

Appendix A. WebSphere Installation ......................... 165
  A.1 Prerequisites ......................................................... 165
  A.2 Create a user for WebSphere and DB2 UDB ................ 166
  A.3 Install the IBM JDK 1.1.7P ...................................... 169
  A.4 Install IBM HTTP Server 1.3.6 ................................. 170
  A.5 Install DB2 UDB 6.1 and Fix pack ............................ 172
    A.5.1 Installing DB2 UDB 6.1 .................................... 172
    A.5.2 Installing the Fix pack ..................................... 175
  A.6 Test the installation so far ...................................... 176
  A.7 Installing WebSphere Application Server .................. 179
  A.8 Testing your installation ......................................... 184
  A.9 Configuring a laptop/mobile machine with a constant IP address 188
  A.10 Configuring and testing an Enterprise Bean ............. 191
    A.10.1 Setting up a DataSource .................................. 192
    A.10.2 Deploying and installing the Enterprise Bean .... 197
    A.10.3 Summary ................................................... 206

Appendix B. ITSO XML Application demo installation .... 207
  B.1 Installing the databases ........................................ 207
  B.2 Configuring WebSphere Application Server 3.0 ........... 209
    B.2.1 Updates to common classes .............................. 209
## Figures

1. XLinks as directed labelled graphs ........................................ 24  
2. A simple CSS stylesheet .................................................. 29  
3. Result of applying the XSL stylesheet to the marketshare XML data ... 34  
4. SAX application components .............................................. 36  
5. DOM interface hierarchy .................................................. 40  
6. Sample DOM tree ........................................................... 41  
7. XSL with attributes ....................................................... 57  
8. Using for:each loops ....................................................... 60  
9. Results of running XMLTest servlet, viewed with Internet Explorer 5.0 ... 68  
10. Using JSP to create XML ............................................... 71  
11. XSL Stylesheet — book.xsl ............................................... 76  
12. Result of running XMLTestWithXSL servlet ........................... 77  
13. Performance comparison — using XML and XSL in a servlet .......... 77  
14. JSP with XSL Island .................................................... 81  
15. Result of the JSP using XSL Islands ................................... 82  
16. XML application workflow model ...................................... 90  
17. XML database model .................................................... 92  
18. Datasource for user profiles .......................................... 98  
19. Extended user profile ................................................... 99  
20. Command line arguments for WebSphere .............................. 100  
21. DTD for user interfaces ................................................. 102  
22. XSL LogonForm stylesheet ............................................. 104  
23. Logon user interface XML document .................................. 105  
24. Logon form ................................................................. 106  
25. LogonOk screen ........................................................... 107  
26. LogonError screen ....................................................... 107  
27. Registration ............................................................... 108  
28. XSLT Island template ................................................... 109  
29. RegisterChange.JSP ..................................................... 111  
30. Changing registration information ..................................... 113  
31. Result of the search for user input “printer” ......................... 118  
32. Overview of architecture for the order subsystem .................. 119  
33. Public Interface of itso.xmlapp.beans.BasketBean .................. 120  
34. Interaction between search results and orderbasket.jsp ............ 120  
35. A template, database and resulting XML ............................. 124  
36. Template file PRODUCTS.XML ......................................... 127  
37. Screen capture of the orderbasket view ................................ 132  
38. Order list screen ........................................................ 141  
39. Order detail screen ..................................................... 144  
40. Supplier update and viewing system ................................... 146
84. Bean deployment notice ........................................ 199
85. Deployment completed dialog .................................. 199
86. Deploy Bean dialog after selecting a Bean ................... 200
87. Viewing the Deployment Descriptor inside WebSphere Admin Console 201
88. Selecting a DataSource ......................................... 202
89. Setting the Advanced properties of a bean ................... 203
90. Successful creation of an Enterprise Bean ................. 203
91. WebSphere Increment sample ............................... 204
92. Results of using the Increment Bean sample .............. 205
93. Viewing Enterprise Bean table using DB2 Command Line Processor 206
94. Installing the DB2 Driver .................................... 208
95. Web application definition .................................... 210
96. DAXSL servlet definition .................................... 211
97. DAXSL properties ........................................... 211
98. MIME filter definition ....................................... 212
99. Installing the JSP enabler .................................... 212
100. WebSphere Standard Administrative Console ............. 213
101. Enabling serving servlets by classname .................... 214
## Tables

1. DTD symbols .............................................................. 18
2. Attribute types ............................................................ 19
3. Validating and non-validating parsers ............................. 49
4. Simple example of XML with XSL ................................. 56
5. DefaultApplyXSL .......................................................... 64
6. Application modules .................................................... 96
7. Defining servlets used in xmlapp .................................. 215
Preface

Extensible Markup Language (XML) has very quickly gathered a large number of industry supporters. Therefore, a significant number of XML-based conferences, books, Web sites, and training classes have sprung up. Very soon, a large number of new XML-based tools will be available.

In this book we try to show several different ways of how and where XML can be used in a B2B and B2C environment.

The book was written for those interested in designing and developing Web applications using XML and related technologies (XSL, XSLT). Project managers, architects, and developers will find this book particularly useful.

We start with an overview of XML technology. Then we explain how to apply XML technology in IBM WebSphere. Finally, we show a sample application written using the technologies described. Source code and installation steps are also available.

This redbook applies to IBM WebSphere Application Server V 3.0.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Luis Ennser is a consultant at the International Technical Support Organization, San Jose Center. He holds a Bachelor degree in Mechanical Engineering from IME-Instituto Militar de Engenharia, Rio de Janeiro, Brazil. Luis has 15 years of experience in Application Development and Networking. He writes extensively and teaches IBM classes worldwide on all areas of XML, WebSphere, and Java. Before joining the ITSO, he worked as an e-business Solution Architect in Brazil.

Christophe Chuvan is a Software Developer/Consultant working for Streamlink Pty Ltd, an IBM Premium Business Partner based in Sydney, Australia. He has 5 years of experience in dealing with Internet-related technologies and has been specializing in electronic commerce for more than 3 years. He holds a Bachelor of Computing Science from Bond University, majoring in Network and Communications. His areas of expertise include consulting and development of electronic procurement solutions on various platforms, which include Lotus Notes and Java servlets.
Paul Fremantle is an e-business Software Architect in the United Kingdom. He has 10 years of experience in Information Technology and Consulting, and 5 years of experience in the Web and Java. He holds an M.A. degree in Mathematics and Philosophy from Balliol College, Oxford University, and an M.Sc in Computation from the Oxford University Computing Laboratory. He has worked at IBM for 3 years. His areas of expertise include Java, Internet technologies and protocols, Enterprise JavaBeans, IBM WebSphere, performance and security. He has published articles on IBM WebSphere v3.0 and Enterprise JavaBean portability.

Ramani Routray is a Software Engineer in IBM Global Services in India. He has 2 years of experience in OOA&D and Java programming. He holds a Bachelor of Engineering Degree in Computer Science and Engineering from Sri Siddhartha Institute of Technology, Bangalore University. His areas of expertise include UNIX Internals, Java Beans/Application development and Networking.

Jouko Ruuskanen is a Software IT Architect in IBM Finland. He has had 15 years of experience in Information Technology, of which 5 are mainly in telecommunications, and the last 10 in OO related technologies and tools. He holds an M.Sc degree in Electrical and Communications Engineering from Helsinki University of Technology. His areas of expertise include Java, Smalltalk, IBM WebSphere and Internet application development. He has participated in the writing of several IBM redbooks.

Thanks to the following people for their invaluable contributions to this project:

Andrea Conzett, for supporting the project.
International Technical Support Organization, San Jose Center

Sanjiva Weerawarana, IBM TJ Watson Research Center, for providing technical guidance on XSLT Islands.

Elias Bayeh, IBM WebSphere development, Raleigh, for providing information on the future directions of XML inside WebSphere.

Ted Leung, IBM Center for Java Technology, for providing help with the XML Parser.

Adrian Walmsley, EMEA Northern Region Software Business, for reviewing the book and providing useful feedback.

Jane Fremantle, for the title.
Comments welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks evaluation” on page 237 to the fax number shown on the form.
- Use the online evaluation form found at http://www.redbooks.ibm.com/
- Send your comments in an Internet note to redbook@us.ibm.com
Part 1. Introduction to XML technology
The XML Files: Using XML and XSL with IBM WebSphere 3.0
Chapter 1. Why XML?

This chapter aims to put the eXtensible Markup Language (XML) into perspective. We outline the requirements for XML, the business and technological benefits, as well as the history of XML. We conclude with some real-life examples of how XML is being used.

1.1 Background

The Internet has had amazing growth during the last 5 years. This growth has been caused by a number of factors, including the lower cost of personal computers and better communications infrastructure. However, the biggest driver of Internet growth has been the World Wide Web (WWW), and the information, services and content which can be accessed by anyone using a browser and a phone line.

This growth can be linked to two technical standards which underpin the WWW, HyperText Markup Language (HTML) and HyperText Transfer Protocol (HTTP). Together these form a common language for the Internet: HTTP provides a universal communication protocol, and HTML together with a browser provides a universal user interface. Without these technologies, the Internet remained a useful but unexciting tool for twenty years. Unlike previous hypertext systems, HTML was text-based, which made it easy to work with, and an open standard, which meant that anyone could develop, build, and create HTML and HTML tools.

The inventor of HTML and HTTP, Tim Berners-Lee, certainly did not envisage everything that the Web would be used for. HTML has evolved considerably since it was first presented in 1992, and many advanced Web sites are now finding that they have evolved beyond the capabilities of HTML — that the dynamic, data-driven Web of 1999 has outgrown HTML. That is not surprising: HTML was originally designed to capture a particular sort of data — mainly textual documents with links to other mainly textual documents. It originated in the research environment of CERN, the European high-energy physics lab, and was designed to share linked information from scientific research.

HTML was originally designed to describe the format of a document. The presentation of that format depended on the browser. As the Web expanded into new uses, more and more presentation information was added to Web pages, and more extensions were added to the HTML specification to support this.
Today's Web contains many dynamic Web sites which provide “on-line services”. You can buy almost anything, reserve seats at a conference, check the weather, trade in stocks and shares, and numerous other activities.

In order to represent the sorts of data these applications use in HTML, Web designers are forced to mix together the data and the presentation in a single HTML file. Recent work on Cascading Style Sheets (CSS) has helped to improve it, but even with stylesheets, most dynamically generated HTML files are still a mix of presentation and data.

The desire to manage large amounts of complex dynamic data, and present it, have created the requirement for a universal data format — that is provided by the eXtensible Markup Language, XML.

1.2 XML — a universal data format

While HTML is a single markup language, designed for a particular application, XML is really a family of markup languages: in fact, you can define any number of markup languages in XML. This means that almost any type of data can be easily defined in XML. So, in addition to a universal communications medium (the Internet), a universal user interface (the browser), and a universal programming language (Java), we now have a universal data format — XML. IBM, together with many other companies, believes this is a solid foundation for e-business.

XML is universal not only by its range of applications but also by its ease of use: its text-based nature makes it easy to create tools, and it is also an open, license-free, cross-platform standard, which means anyone can create, develop, and use tools for XML. What also makes XML universal is its power. Data is transmitted and stored in computers in many different ways: originally it was stored in flat-files, with fixed-length or delimited formats, and then it moved into databases, and often into complex binary formats. XML is a structured data format, which allows it to store complex data, whether it is originally textual, binary, or object-oriented. To this day, very few data-driven technologies have managed to address all these different aspects in one package — except XML.

A short comparison of XML and HTML

XML and HTML both descend from common roots. However, because XML is still new, most data on the Web is stored in HTML format. The amount of data currently stored on the Web is hard to imagine: the latest survey (by OCLC Research; see the following Web site):

http://www.oclc.org/oclc/research/projects/webstats/statistics.htm
This survey indicates that there are 300 million pages on the Web. In fact, the amount of data available may be much higher, because counting pages ignores the fact that one dynamically generated page can act as a gateway to a large database.

The problem with data available in HTML format is that it is formatted for people to view, and not for computers to use. HTML consists of a pre-defined set of tags, the purpose of which are known. This makes it a language that is easy to learn and accessible, but makes it hard to re-use the data.

This is where XML enters the picture. As its name indicates, XML is extensible, which means that you can define your own set of tags and make it possible for other parties (people or programs) to know and understand these tags. This makes XML much more flexible than HTML. In fact, because XML tags represent the logical structure of the data, they can be interpreted and used in various ways by different applications.

Much of the value of the Web comes from re-using data. For example, one of the great success stories of the Web are the search engines. They work on the basis of a universal communications method (HTTP), and a universal markup language (HTML), to catalog Web pages. However, search engines work on very limited information, because only a tiny part of an HTML document is designed to be used by a search engine. Imagine how much more powerful search engines could be if the data that they search was stored in a simple, structured, re-usable format.

### 1.3 XML business benefits

XML has three main applications so far:

- Sharing of information between computer systems and between businesses
- Storage and transmission of information within a single application
- Content delivery — delivering information to users

The early usage of XML has been in the first two areas so far — the benefits of XML are easiest to achieve in these areas.
1.3.1 Information Sharing

The benefits of using XML between computer systems and businesses are probably the greatest and easiest to achieve: XML allows businesses to define data formats in XML, and easily build tools which read data, write data and transform data between XML and other formats. This has allowed a number of businesses and industry consortiums to build standard XML data formats. Areas such as Electronic Data Interchange (EDI), inter-bank payments, supply-chain, trading, and document management are all the subject of ongoing XML-based standardization by industry consortiums.

By using XML, the standard can be published, extended, and easily used by new applications. Data transformation tools exist that can convert between existing formats and XML, and new tools are emerging. The ability to link enterprise applications (Enterprise Application Integration) is a key focus for many companies, and has produced cost savings and increased revenue for many enterprise customers. In particular, many businesses aim to improve Customer Relationship Management (CRM) by creating a single logical view of each customer across multiple existing systems. XML is an important technology to create this single customer view.

Furthermore, because XML makes it easy to relate structure to content, XML subsets can be defined with specific industries or applications in mind. For example, XML has been used to define standard data formats for the banking industry. In the same manner, a standard could be developed specifically for flight booking systems, thereby allowing airlines to easily exchange information.

1.3.2 XML inside a single application

The business benefits of using XML within a single application are less compelling, but they are very easy to achieve, and so we have seen a number of applications that use XML internally. The benefits of this approach are speed of development and future extensibility.

XML is a very powerful and flexible language for describing the complexities of the real world. Because XML can describe complex data, it means that it can be a powerful tool to creating new applications. The extensibility of XML means that the application can grow and develop without major changes. Finally, when the application needs to connect to other applications, XML makes an excellent way of linking the application with others.
1.3.3 Content delivery

XML has a number of key benefits for content delivery. The main benefits are the ability to support different users and channels, and to build more efficient applications. Channels are information delivery mechanisms — for example, Digital TV, phone, Web, and multimedia kiosk. Supporting different channels is an important step in delivering e-business applications to customers through their chosen medium. XML is a key technology for this.

For example, a customer and a supplier both need to access the same on-line product catalogue. Although the information is the same, the visual emphasis will differ, depending on who the user is: the customer will be more interested in looking for information on functionality, pricing, and availability, while the supplier will want to have easy access to catalog maintenance and inventory information. All this information might be stored in a single XML document and be displayed differently by the application.

Using XML for content delivery has been limited by the availability of XML-enabled browsers. Microsoft Internet Explorer 5.0 was the first browser to support XML directly, and until a large majority of users are XML ready, many Web masters will not adopt XML. As you will see later in this book, IBM WebSphere allows the server to generate HTML from XML, and therefore support users of standard browsers. As people become more familiar with this technology, this particular strength of XML is likely to be exploited more often.

1.4 Technological benefits of XML

In order to see the technological benefits of XML, let us consider a fictitious example.

1.4.1 An example of using XML

Many libraries offer their catalogs over the Web. Usually, there is a simple Web form where you enter a title, name or subject, and you are presented with some search results. If you wanted to search several libraries, you would need to go to each of their Web sites. It would be useful and convenient if there could be a single Web page which could search many libraries. To build that today would require extracting the data (title, author, ISBN, and so forth) from each search results page. However, each search results page is formatted differently, and the data is mixed in with presentation information. To collate the results of many searches would require complex programming for each libraries Web site, separating the data from the presentation.
Suppose, instead, that there were a single format for returning search results from libraries: let us call it *A Book Catalog Markup Language* or ABCML. ABCML would define tags for the author, title, and so on, thus making it easy for a computer to extract the data. Building the meta-catalog suddenly becomes easier.

ABCML would also help the libraries, because they could re-use each other’s software. Also, when a new book came in, the publisher could provide the book’s catalog information in ABCML, and save the librarian the effort of typing it into the catalog.

### 1.4.2 Major benefits

So ABCML would help us build a meta-catalog, and also help the libraries re-use existing data. Those arguments may not be enough to persuade a particular library’s IT director to rewrite his online catalog. However, there are a number of further technical benefits to using XML. As well as re-using data, these benefits include: separating data from display, extensibility, and adding semantic content to the data.

#### 1.4.2.1 Re-use of data

We have seen the benefit of re-using data: the librarian could re-use the publisher’s data, because it was in a common format, and we could re-use the data when we built our meta-catalog. Of course there are many common file formats in the world of computing that have allowed data re-use. These have usually been proprietary and application specific. XML is neither of those.

#### 1.4.2.2 Separation of data and display

What are the benefits of separating data and presentation? First, without this separation, we could not achieve the simple re-use of the data. Second, the presentation changes. If you look at the Web sites of 5 years ago, and the Web sites of today, they are radically different. If you look at any successful Web site, you will probably see at least one redesign a year. That is not simply because it is trendy — successful Web sites analyze and react to feedback from users, and redesign the site to be more productive and intuitive. Let us return to the library Web site — the Web site gets a redesign, but the underlying data remains in place — so it makes sense to separate the data output from the Web site design.

There is still another even more compelling reason to separate data and display: the rise of pervasive computing. Pervasive computing is where computing devices become integrated into common everyday appliances: mobile phones, televisions, printers, and palm computers. Each of these
appliances may have a different display technology, and require different instructions on how to display the data. The same search of the library catalog should be viewable on a mobile phone or a high-resolution PC.

1.4.2.3 Extensibility
HTML has been a constantly evolving standard since it emerged, and one of the problems it has faced is that it has often been extended by companies wishing to go beyond HTML. Browser suppliers regularly add non-standard extensions to HTML. Similarly, Web server manufacturers build “server-side” extensions to HTML: these include NCSA includes, Microsoft Active Server Pages, Java Server Pages, and many others. This has led to many confusing variants of the HTML standard, causing difficulties for Web developers, tool vendors, and ultimately for end-users.

As the name implies, eXtensible Markup Language was designed from the beginning to allow extensions. If we go back to our example of the library, when they first indexed books, the Web did not exist. Probably the library catalog has no references to Web sites in it. Nowadays, many books have a companion Web site, and the librarian may wish to reference it. If XML were used to develop the catalog, then this could easily be accomplished. Importantly, with XML, old software is not disrupted by the addition of new information.

1.4.2.4 Semantic information
The final major benefit of XML is that it builds semantic information into the document. Semantic information (or meaning) is what allows readers to decide whether the book is about the color Brown, or written by Brown. An HTML-based Web search engine cannot do that, because it cannot distinguish between the title and author in the document — there isn’t enough semantic information in the HTML page. With XML, the document includes self-descriptive terms that show the meaning of the document.

1.4.2.5 Other benefits
The other main benefits of XML are that it is human-readable, tree-based, and easy to program. As time goes on, a large number of XML tools are emerging from both existing software vendors and XML startup companies. It is human-readable, because it is based on text and simple tagging. This means that it can be understood by humans, and it can be written using nothing more than a text-editor or word-processor. This is important in the sense that programmers can interpret the data faster when writing new applications, but once they start running, no one reads the data (only the applications). The tree-based structure of XML is much more powerful than fixed-length data formats. Because objects are tree structures as well, XML is
ideally suited to working with object-oriented programming. In particular, many people believe that there is an excellent affinity between Java and XML.

Finally, XML is easy to program, because there are already standards for XML parsers. XML parsers are the subsystems that read the XML and allow programmers to work with XML. Because XML parsers are available to be re-used in new computer systems, many programmers are starting to use XML, even if they do not need any of the previously mentioned benefits.

We cover the affinity between XML and Java, and also the XML parser, in Chapter 3, “Processing XML using Java” on page 35.

1.5 XML history

The history of XML is really the history of another system: Standard Generalized Markup Language or SGML. XML is actually just a subset of SGML, and SGML has been around for many years. In fact SGML dates back to the late 60's and the work of an IBM employee Charles Goldfarb. Goldfarb was developing a system to share documents, and together with two of his colleagues, Edward Mosher and Raymond Lorie, he put together a markup language called Generalized Markup Language, GML. (Of course, GML really stands for Goldfarb, Mosher, and Lorie, who invented it.)

GML also drew on work by William Tunnicliffe and Stanley Rice on building a “Generic Coding” system, which was done under the auspices of the Graphic Communication Association.

GML formed the basis of many IBM documentation systems including Script and Bookmaster. Later developments led to SGML, which became an ISO standard in 1986. SGML has had a great deal of success, but unfortunately that has mainly been limited to large corporations and government departments. The reason was that SGML required a major investment, and so only large organizations had the resources to achieve the benefits of SGML. For more information on SGML and its history, read The SGML Handbook, 0198537379, by Charles Goldfarb.

Probably the most popular application of SGML was the Web: HTML is actually defined in SGML, and is a very limited example of SGML. XML emerged towards the end of 1996 as a presentation to World Wide Web Consortium (W3C). The complete text of the initial draft is available at:

http://www.w3.org/pub/WWW/TR/WD-xml-961114.html
The initial draft included ten key design goals, which are worth showing here:
1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness is of minimal importance.

Since the end of 1997, XML has grown quickly under the leadership of W3C, and in September 1999, W3C announced that XML activity was entering its third phase: phase one built the base technology, phase two created stylesheets and namespaces, and phase three will endeavor to finish the ongoing work and introduce new specifications for an XML query standard.

1.6 Real-life uses of XML

In this section we present two examples of how XML is being used in real-life to bring benefits to people, businesses, and organizations. For more examples, see the XML section in the IBM developerWorks site at:
http://www.ibm.com/developer/xml

or at the XML.org catalog site at:
http://www.xml.org/xmlorg_registry/index.shtml

1.6.1 SABRE and Wireless Markup Language

The SABRE Group is one of the major distributors of international travel services, offering electronic travel bookings through travel agents and on the Web worldwide. They are transforming their travel information into XML using a Java application, and then allowing mobile phone users worldwide to look up, reserve, and purchase travel from a mobile phone. The XML is automatically translated from XML into Wireless Markup Language, which is a standard for building applications on mobile phones. The benefits of XML to this application are its extensibility, the speed of development, and the ability
to build a standard repository in XML, and translate as needed into a particular environment, in this case the mobile phone.

1.6.2 Chemical Markup Language

Chemical compounds and molecules are complex combinations of atoms. There are around 20 million “known” molecules, and until recently, there has been no standard way of representing molecules that was machine-readable. Two British scientists, Prof. Peter Murray Rust of the University of Nottingham and Dr. Henry Rezpa of Imperial College, London have developed a standard way of describing molecules in XML. Chemical Markup Language (CML) is expected to bring huge savings to the chemical industry, and also aid communications between chemists, as well as with other related disciplines such as biology and medicine.

One of the key benefits of using XML in this application was the large quantity of tools available for XML, which have helped to create applications for CML quickly and productively.
Chapter 2. XML overview

In the introduction, we have looked at why there was a need for XML to be developed in the first place. The following section will discuss basic XML concepts such as tags, Document Type Definitions, and namespaces.

2.1 XML concepts

While this book is not intended as an XML reference manual, we feel it is useful to cover a few basic concepts, an understanding of which will be required in the second part of this book. Since HTML is a popular markup language with which a lot of people are familiar, you will notice that comparisons will often be made between HTML, XML, and their common ancestor, SGML. While people’s familiarity with HTML will hopefully simplify their task in understanding XML, it is also important to emphasize the differences between the two languages.

2.1.1 XML and SGML

We have already mentioned the fact that XML is a subset of SGML, which is more powerful but also much more complex. Like SGML, XML is a metalanguage which allows the author of a document to provide markup definitions or in other words to associate “tags” with data. The purpose of the tags is to provide information to an application about the data through their association with the document contents.

Accordingly, an XML document typically consists of markup and character data. The markup information conveys the meaning behind the document contents and is represented through tags and other XML elements, while character data represents the actual content.

In the following example, the <author> tag indicates that “Joe Bloggs” represents an author’s name.

<author>Joe Bloggs</author>

Note that, in the above example, although we know that “Joe Bloggs” is the name of an author, it does not necessarily imply that he is the author of the document. The meaning behind the tag will depend on the context in which it will be interpreted and therefore on the application reading the document.
2.1.2 Document validity and well-formedness

XML is a metalanguage, which means that the author can define a set of tags that will suit the purpose of the application using the XML documents. However, one of the main advantages of XML is that it can be read by various parties and interpreted in different ways by these parties. Therefore, there are a few rules that XML tagsets and documents should adhere to, in order to ensure that they are usable by any XML application.

As we mentioned before, XML has tighter constraints than HTML, which tolerate minor structural irregularities in the documents they are parsing, such as unclosed tags. XML parsers will not accept documents that contain start tags, such as <AUTHOR>, without their counterpart end tags, in this example </AUTHOR>. This differs from HTML and SGML, which can be parsed even without any explicit end tags. On the other hand, XML does accept empty tags such as <AUTHOR/>. Well-formedness constraints also deal with attribute names, which should be unique within an element, and attribute values, which must not contain the character “<”. A document is said to be well-formed when it conforms to these constraints, which are referred to as the well-formedness constraints (WFC) defined in the XML 1.0 Recommendation (refer to http://www.w3.org/XML/).

The notion of validity applies to XML documents which define a Document Type Definition (DTD) against which they must be validated. When a document is checked for validity, the XML parser checks whether the document’s contents match the logical structure as it is defined in the DTD. For example, for a document to be valid, all tags and attributes appearing in the document must have corresponding declarations in the DTD, and all elements appearing within other elements must respect the content model defined in the DTD.

It is worth noting that validity and well-formedness are two different aspects of an XML document. While well-formedness insures that XML parsers will be able to read the document, validity determines whether an XML document adheres to a DTD or schema. An XML application will check for and reject documents that are not well-formed before checking whether these documents comply with its validity constraints (VC). After a system is tested, validity checking can be turned off to improve performance.
2.1.3 Document Type Definition

2.1.3.1 What is a DTD?
Like SGML from which it derives, XML supports the use of Document Type Definitions (DTD). Since it is not the intent of this book to serve as an XML reference manual, we will not describe all the syntax elements of a DTD here. However, it is essential to understand the purpose and use of a DTD, and that is what we will focus on in this section.

The DTD specifies the grammatical structure of an XML document, thereby allowing XML parsers to understand and interpret the document's contents. The DTDs contain the list of tags which are allowed within the XML document and what their types and attributes are. More specifically, the DTD defines how elements relate to one another within the document's tree structure, and specifies which attributes may be used with which elements. Therefore, it also constrains the element types that can be included in the document and determines its conformance: an XML document which conforms to its DTD is said to be valid.

A DTD comes in the form of a simple text file, which can be either be stored in a separate file or embedded within the XML file. XML documents referencing a DTD will contain the <!DOCTYPE> declaration which either contains the DTD declaration, or specifies the location of an external DTD, as in the following example:

```
<!DOCTYPE LibraryCatalogue SYSTEM "library.dtd">
```

On the other hand, an XML document is not required to specify a DTD. However, with most applications, it will prove beneficial or even necessary to build a DTD which conveys efficiently the meaning behind the XML file's contents. DTDs provide parsers with clear instructions on what to check for when they are determining the validity of an XML document.

Having the logical definition of an XML file stored separately allows for the resulting DTD to be shared across organizations, industries or the Web. When building XML applications, it is probably a good idea to look for existing DTDs that might suit your purpose. At the time of writing, well-recognized, universal standards have yet to emerge as more current industry initiatives are still in the drafting stages. However, as XML becomes more popular, more commercially-oriented or industry-oriented applications will likely appear and standards will emerge.
For more information on the latest emerging XML standards, the following sites may prove a good starting point:

http://www.schema.net
http://www.oasis-open.org

2.1.3.2 DTD Example
The DTD has its own syntax, but is similar to XML in that it also uses markup tags. The following sample shows a simple internal DTD:

```xml
<?xml version = "1.0"?>
<!DOCTYPE authors [
  <!ELEMENT authors(author)+>
  <!ELEMENT author(firstname, lastname, title)>    
  <!ELEMENT firstname(#PCDATA)>                    
  <!ELEMENT lastname(#PCDATA)>                     
  <!ELEMENT title(#PCDATA)>                        
]>                                                  
...                                                
[ insert XML data here]                            
...
```

In the above example, the `DOCTYPE` statement represents the Document Type Declaration and the statements included within the square brackets make up the Document Type Definition. Both terms share the same acronym (DTD), which can be confusing but it is usually clear from the context which of the two meanings is being referred to.

A well-formed XML document must contain at least one root element, that is, a single element declaration. Additionally, the `DOCTYPE` name specified in the declaration must match that root element, in this case `authors`:

```xml
<!DOCTYPE authors [ 
  <!ELEMENT authors(author)*>* 
]> 
```

The second line constitutes an element declaration and the * indicates that the `authors` element can contain one or more `author` elements, which in turn are declared like this:

```xml
<!ELEMENT author(firstname, lastname, title)> 
```

Similarly, the `author` element contains several elements: `firstname`, `lastname`, `title`. However, only one instance of each is allowed in this case.

```xml
<!ELEMENT firstname(#PCDATA)>     
<!ELEMENT lastname(#PCDATA)>       
<!ELEMENT title(#PCDATA)>          
```
These last three elements contain only text and are therefore defined as parse character data or PCDATA. The adjunction of the # character marks the PCDATA type as a reserved word, and it cannot be used for names created by the author.

As mentioned earlier in this chapter, the DTD can either be stored within the XML document which it validates, or in an external file. The following is an example of a Document Type Declaration specifying an external DTD:

```xml
<?xml version = "1.0"?>
<!DOCTYPE authors SYSTEM "authors.dtd">
```

The use of the SYSTEM keyword indicates to the parser that this is an external declaration and that the set of rules for this XML document can be found in the specified file.

### 2.1.3.3 What's in a DTD?

A Document Type Definition can contain different types of declarations. A list of these different types follows:

- **Elements** constitute the basic building blocks of an XML file and are declared like this:

  ```xml
  <!ELEMENT elementName(allowed element contents)>
  ```

  Example:

  ```xml
  <!ELEMENT greeting (#PCDATA)>
  <greeting>Hello, World!</greeting>
  ```

  Table 1 lists all the declaration attributes allowed inside an element declaration.
Table 1. DTD symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>, (comma)</td>
<td>Means “and” in specified order</td>
<td>TITLE, AUTHOR</td>
<td>TITLE and AUTHOR in that order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Means “or”</td>
<td>TITLE</td>
<td>TITLE or AUTHOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AUTHOR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Means “optional”, but no more than one is allowed</td>
<td>ISBN?</td>
<td>ISBN does not have to be present, but if it is, there can be no more than one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(TITLE</td>
<td>Any number of TITLE or AUTHOR elements can be present</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AUTHOR) *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AUTHOR+</td>
<td>At least one or more AUTHOR elements must be present</td>
</tr>
<tr>
<td></td>
<td>Used to group elements</td>
<td>&lt;!ELEMENT BOOK (AUTHOR</td>
<td>An AUTHOR or a TITLE element must be present and must precede the YEAR-PUBLISHED and ISBN elements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TITLE), YEAR-PUBLISHED, ISBN)&gt;</td>
<td></td>
</tr>
</tbody>
</table>

• **Attributes**, as their name indicates, are attributes of an element which must be declared in the same DTD:

```xml
<!ATTLIST elementName attributeName attributeType attributeDefault>
```

Example:

```xml
<!ELEMENT BOOK (#PCDATA)>
<!ATTLIST BOOK
  ID ID #REQUIRED
  TYPE (Hardcover | Paperback) "Hardcover"
  STORELOC CDATA #FIXED "5th Avenue"
  COMMENT CDATA #IMPLIED>
```

Table 2 provides a description of the various attributes types which can be used in attribute declarations.
Table 2. Attribute types

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDATA</td>
<td>Can contain any kind of character data</td>
</tr>
</tbody>
</table>
| ID            | Must have unique values within the element. In the example below, TYPEID is of the ID type, and so requires unique values within the range of BOOK elements:  
  `<BOOK TYPEID="ch1">See Spot Run</BOOK>`  
  `<BOOK TYPEID="ch2">Jack and Jill</BOOK>`  
  (enumerated) Attributes can have a specified list of acceptable values. |
| #REQUIRED     | The attribute is required. |
| #IMPLIED      | The attribute is optional. |
| #FIXED        | The attribute is fixed (the syntax is of the type "#FIXED Value"). |

- **Entities** are used to represent one or more characters in an XML document and act as constants, the value of which can be changed without the need to edit corresponding XML documents:

  ```xml
  <!ENTITY entityName "character string represented">
  ```

  Example:

  ```xml
  <!ENTITY proiname "ACME Calendar">
  ```

  (XML file:)
  Thank you for choosing &proiname; as your primary scheduling program
  (rendered:)
  Thank you for choosing ACME Calendar as your primary scheduling program

- **Parameter Entities** are entities which are used within the DTD itself. Their declaration differs by the inclusion of the % character:

  ```xml
  <!ENTITY % entityName "character string represented">
  ```

  Example:

  ```xml
  <!ENTITY % commonAtts "ID ID #REQUIRED
  MAKE CDATS #IMPLIED
  MODEL CDATA #IMPLIED">
  ```

  ```xml
  <!ELEMENT CAR (#PCDATA)>
  <!ATTLIST CAR %commonAtts>
  ```

  ```xml
  <!ELEMENT COMPUTER (#PCDATA)>
  <!ATTLIST COMPUTER %commonAtts>
  ```
• **Notations** are used to refer to data from an outside (non-XML) source. They provide a basic means by which non-textual information can be handled within a document:

```xml
<!NOTATION name ExternalID>
```

Example:

```xml
<!NOTATION jpeg SYSTEM "jpeg.exe">
<!NOTATION gif SYSTEM "gif.exe">

<!ELEMENT person (#PCDATA)>
<!ATTLIST person
    picformat NOTATION (jpeg | gif) #REQUIRED>
```

(XML file:)

```xml
<person picformat="jpeg">Kelly Brown</person>
```

• **Comments** can be inserted inside the DTD by using the following notation:

```xml
<!-- insert comment text here -->
```

Example:

```xml
<!-- XML Comments Example -->
<!-- Author: Christophe Chuvan -->
<!-- Last Modified Date: 05/10/99 -->
```

### 2.1.4 Namespaces

Now that we know what DTDs are, we can take a look at another XML concept called **namespaces**. Namespaces are useful when there is a need for elements and attributes of the same name to take on a different meaning depending on the context in which they are used.

For instance, a tag called `<TITLE>` takes on a different meaning, depending on whether it is applied to a person or a book. If both entities (a person and a book) need to be defined in the same document, for example, in a library entry which associates a book with its author, we need some mechanism to distinguish between the two and apply the correct semantic description to the `<TITLE>` tag whenever it is used in the document. Namespaces provide the mechanism that allows us to write XML documents which contain information relevant to many software modules. Consider this example:
As we can see, the <TITLE> tag is used twice, but in a different context, once within the <AUTHOR> element and once within the <BOOK> element. Note the use of the xmlns keyword in the namespace declaration. Interestingly, the XML Recommendation does not specify whether a namespace declaration should point to a valid URI (Uniform Resource Identifier), only that it should be unique and persistent.

In the previous example, in order to illustrate the relationship of each element to a given namespace, we chose to specify the relevant namespace prefix before each element. However, it is assumed that once a prefix is applied to an element name, it applies to all descendants of that element unless it is over-ridden by another prefix. The extent to which a namespace prefix applies to elements in a document is defined as the namespace scope. If we were to use scoping, the above example would then look like this:

```xml
<?xml version="1.0"?>
<library-entry xmlns:authr="authors.dtd"
               xmlns:bk="books.dtd">
  <bk:book>
    <title>XML Sample</title>
    <pages>210</pages>
    <isbn>1-868640-34-2</isbn>
    <authr:author>
      <firstname>Joe</firstname>
      <lastname>Bloggs</lastname>
      <title>Mr</title>
    </authr:author>
  </bk:book>
</library-entry>
```
In this example, it is clear that all elements within the <BOOK> element are associated with the bk namespace, except for the elements within the <AUTHOR> element which belong to the authr namespace.

### 2.1.5 DTD versus XML Schemas

The DTD provides a relatively easy-to-use way to describe the syntax and semantics of XML documents. However, to achieve this simplicity, a compromise was made when porting DTD support over from SGML to XML, which resulted in the expected simplification, but also in limitations that prevented the DTD from performing a high degree of semantic checking.

For example, a DTD allows for limited conditional checking by specifying allowed values, but there is no support for more complex semantic rules. For instance, it is currently impossible to check that an element which should contain a date actually contains a date. There are also limitations when it comes to defining complex relationships between data elements and their usage, especially when XML documents also use namespaces which might define elements conflicting with DTD declarations.

Therefore, there is a need for a way to specify more complex semantic rules and provide type-checking within an XML document. XML Schemas, while still in the drafting stage, aim to provide such functionality and also introduce new semantic capabilities such as support for namespaces and type-checking.

Whether XML Schemas provide a viable alternative to the DTD or come in the form of extensions to the DTD's functionality, they will need to provide a high degree of compatibility with existing structures. Also, because DTD support is part of the current XML Recommendation, the schema designers cannot overlook the fact that many applications and data structures will have been developed with the DTD in mind. For more information on XML Schemas, refer to the specification documents from the W3C (currently both parts are still in the drafting stage):

**XML Schema Part 1: Structures**

http://www.w3.org/TR/xmlschema-1/

**XML Schema Part 2: Datatypes**

http://www.w3.org/TR/xmlschema-2/
2.2 XML Linking

XML Linking Language (XLink) defines a set of constructs which may be inserted into XML resources to describe links between objects. These constructs use XML syntax to create structures which can describe links starting from simple unidirectional hyperlinks to multi-ended and typed links. In case of HTML, linking means moving from one point in a document to another point in the same or another document. Using XLink, a new document can be embedded in an old document. A new document can be opened or a new document can replace an old document using XLink. XLink can be used to provide a list of links related to a resource and can lead to multiple destinations. Let us look at some terminologies associated with XML linking.

A locator for a resource can be provided in form of a URI. A URI contains (optional) a fragment identifier that is separated from the URI by a crosshatch symbol(#). Link is nothing but an explicit relationship between two or more data objects or portions of data objects. Linking Element describes the existence of the link. XLink has four different kinds of linking attributes:

- Locator attributes
- Arc end attributes
- Behavior attributes
- Semantics attributes

Locators define the exact source of the resource, and href is a locator attribute.

Arcs define the traversal of links and the information regarding the sequence of links are stored in arc end attributes. The attributes associated with arcs are from and to.

Behaviors define the activation of the link. The attributes associated with behaviors can be show or actuate. Behaviors with the show attribute can be new, parsed, or replace. Behaviors with actuate attributes can be auto or user.

Semantics define the relevant information, which may be used by the application. The attributes related to semantics are role and title. The role describes the function of the content of the link, and the title provides human-readable text describing the link.
XLinks describe a directed labelled graph of XML objects. Each XLink is a collection of locators and arcs. Locators identify the remote resources and use the “href” attribute to specify the resource. A link can have multiple locators. Locators are the nodes in the labelled graph connecting XML objects. Arcs specifies the traversal behavior and defines the edges of the labelled graph. Each arc has a “from” and a “to” locator. A link can also have multiple arcs. Let us look at this diagram, which describes XLinks as directed labelled graphs. This diagram (Figure 1) shows 3 links, which connect 4 documents.

Figure 1. XLinks as directed labelled graphs

Figure 1 shows three links, “LinkA”, “LinkB”, and “LinkC”. LinkA has a single arc (Arc0). LinkB has two arcs (Arc1 and Arc2). LinkC has two unlabeled arcs (bidirectional).

A simple link has a single locator and arc. Simple links can be used to approximate the use of the <A> tag of HTML. An example of a simple link follows:

```xml
<BOOK id="2.2" xmlm:xlink="http://examples.com/xlnk/0.2">
  <TITLE>
    xyz
  </TITLE>
  <COPIES>
    7
  </COPIES>
  <AUTHOR xlink:href="author1.xml#first" xlink:show="parsed" />
</BOOK>
```
In contrast to simple links, extended links can connect to any number of resources and can have multiple locators and arcs. An example of an extended link follows:

```xml
<xlink:extended xmlns:xlink="http://www.examples.com/xlnk/0.9">
  <xlink:locator href="#xyz" id="B5.5"/>
</xlink:extended>
```

2.3 XPath

The name *XPath* comes from its use as notations in URLs for navigating through XML documents. The aim of an XPath is to address parts of an XML document. XPath answers the needs common to both XPointer and XSL transformations. XPath uses a compact syntax, and it operates on the logical structure underlying XML to facilitate usage of XPath within URIs and XML attribute values. Xpath supports XML namespaces because XPath models an XML document as a tree of nodes (root nodes, element nodes, attribute nodes, text nodes, namespace nodes, processing instruction nodes, and comment nodes). The basic syntactic construct in XPath is the *expression*. An object is obtained by evaluating an expression, which has one of the following four basic types:

- Node-set (an unordered collection of nodes without duplicates)
- Boolean
- Number
- String

XPath uses path notation to define locations within a document. The paths starting with a “/” signifies an absolute path. A simple example of this is shown below.

Let us consider an XML document (Library.xml) which describes a Library System. This sample document will be used throughout XPath and XPointer for examples.

```xml
<?xml version="1.0"?>
<!DOCTYPE LIBRARY SYSTEM "library.dtd">
<LIBRARY>
  <BOOK ID="B1.1">
    <TITLE>xml</TITLE>
    <COPIES>5</COPIES>
  </BOOK>
  <BOOK ID="B2.1">
```

Chapter 2: XML overview 25
The path \texttt{/child::book/child::copies} selects all \texttt{copies} element children of \texttt{book} which are defined under the document's root. The above path can also be written as \texttt{/book/copies}.

The XPath location step makes the selection of document part based on the basis and the predicate. The basis performs a selection based on Axis Name and Node Test. Then the predicate performs additional selections based on the outcome of the selection from the basis. A simple example of this is as follows:

The path \texttt{/child::book[position()-1]} selects the first \texttt{book} element under root. This location step can also be written as \texttt{/book[1]}

For example, the path \texttt{/book/author/@*} would have selected all the \texttt{author} elements' attributes

The path \texttt{/book/author[@type='old']} would have selected all the \texttt{author} elements with type attribute equal to "old".

2.4 XPointer

XPointer (eXtended Markup Language Pointer) is a reference to a fragment of a document. The syntax of XPointer was derived from the Test Encoding Initiative (TEI), modified to support the need of the HTTP protocol for encoding URIs.

Let us consider a situation where your document points to a paragraph in another document on the Web. At certain point of time, if the paragraph of the document on the Web changes and you still want to get the latest version of the paragraph in your document, then XPointer is the answer. XPointer
defines the meaning of the identifier portion of the URIs that locate the resources of MIME media types “text/XML” and “application/XML”.

After all, each such “XPointer” can point at anything in an XML document. XPointers make selection of parts of documents easy, but all the processing takes place on the client side, and the server sends the files to clients that are used by the client. XPointer also provides syntax, using which, a fragment of data in a document can be specified and can be passed to the server as a URL. Enhanced interpretation of the XPointer by the server helps in ignoring a great deal of unnecessary information.

XPointers operate on the XML information set (the XML information set is derived from the elements and other markup constructs of XML document) by selecting them by their structural relationship to other identified nodes. The errors that can arise using XPointers are: syntax errors, resource errors due to incorrect URIs, and sub-resource errors.

For example, in the URI: http://www.ibm.com/xptr.xml#id(info.5), the string id(info.5) is a location term. In terms of XPointers, in the URI: http://www.ibm.com/xptr.xml#root(), root is the location term. Here, root() points to the root of the document. The keywords which are generally used by XPointer are: CHILD, ANCESTOR, PSIBLING, DESCENDANT, PRECEDING, FOLLOWING and FSIBLING.

### 2.4.1 Types of XPointer locators

Location terms of the locators may be one of the three categories. They can be absolute or relative or string matching.

Let us assume that the XML document (library.xml) is available at the URL http://www.examples.com/library.xml. The second book in the document can be referred as http://www.examples.com/library.xml#id(B2.1). Here, the reference to the id with exact value is an example of absolute location term.

We could refer to the second sub-element of the root element in the document (library.xml) using the URI: http://www.examples.com/library.xml#root().child(2,BOOK). This is an example of relative location term because it depends upon a starting location identified by root().

Finally, let us consider the following for string matching location terms:
http://www.examples.com/Library.xml#id(B5.5).parent().preceeding(2,COPIES). This states the COPIES preceding twice the parent of the element named B5.5. Location pointers can avoid using keywords repeated between steps. For
example, `ID(B2.2)CHILD(3,TITLE)CHILD(2,COPIES)` can also be written as `ID(B2.2)CHILD(3,TITLE)(2,COPIES)`.

The best effect of XPointers can be felt when file structures avoid making the processing application load the entire document rather than the desired chunk. The best way to do this is to make the file system itself an XML processor storing XML documents as elements, rather than as a single file, which has to be parsed in the traditional way.

### 2.5 eXtensible Stylesheet Language (XSL)

XSL is the language defined by the W3C to add formatting information to XML data. Stylesheets allow data to be formatted based on the structure of the data, so one stylesheet can be used with many similar XML documents.

XSL is based on two existing standards, Cascading Style Sheets (CSS) and Document Style Semantics and Specification Language (DSSSL).

CSS is the stylesheet language for HTML 4.0, and as such, is well supported in Web design tools, such as IBM WebSphere Studio. XSL is mainly based on CSS, and so a short description of CSS is provided below. CSS can also be used as a formatting language for XML, but it is less powerful than XSL. Because CSS was designed for the Web, it is excellent for defining the presentation of data for Web browsers.

XML aims to support any possible display, and printed output has a number of challenges that browsers do not face. DSSSL is the stylesheet language of SGML and has mainly been used for printed output. Therefore, elements of DSSSL that go beyond CSS have been incorporated into XSL. More information on DSSSL can be found at the following Web address:

http://www.jclark.com/dsssl/

#### 2.5.1 Cascading Style Sheets

Cascading Style Sheets were designed to help separate presentation from data with HTML. The reason that they are called Cascading Style Sheets is because HTML, like XML, has a tree structure, and styles which are applied to the root of a tree cascade down to the branches and leaves. CSS allows the Web developer to define styles that apply:

- To any given type of element (for example, all paragraphs)
- To a class of elements (for example, all paragraphs which contain code samples)
• To a particular element (for example, the third paragraph)

This is achieved by specifying classes and ids in the HTML, and applying styles to them.

A very simple stylesheet is presented in Figure 2. This stylesheet defines a standard font and colors for all text in the BODY of the HTML file. It defines a specific class of text which is twice the normal size, bold and capitalized (.largeClass), and finally it specifies that a particular element labelled THISID should be displayed in fuschia-colored cursive text.

The benefits of CSS are well-understood: Web developers can easily change the layout and presentation of a whole site by editing a single stylesheet. CSS can be used with XML if the display engine supports it. So far, the only shipping browser that supports this feature is Microsoft Internet Explorer 5.0.

```
BODY{
  font-family : Verdana,sans-serif;
  font-weight : normal;
  color : black;
  background-color : white;
  text-decoration : none;
}
.largeClass{
  font-size : 200%;
  font-weight : bolder;
  text-transform : capitalize;
}
#THISID{
  font-family : cursive;
  color : fuchsia;
}
```

*Figure 2. A simple CSS stylesheet*

CSS can be used within a document, or referenced in a separate stylesheet, which is the more common approach. For more information on CSS, see:

http://www.w3.org/Style/CSS

### 2.5.2 XSL = fo: + XSLT

Although XSL has derived much from CSS, the approach of XSL is much more powerful, and has major differences from CSS. XSL is defined by a working draft of the W3C at the following URL:
XSL actually consists of two different standards, the Transformation language, and the formatting objects.

The transformation language is called XSLT, and is defined as a working draft of the W3C at the following URL:

http://www.w3.org/TR/WD-xslt

XSLT defines a common language for transforming one XML document into another. It defines how to create a result tree from a source tree.

The formatting objects (FO) define how to display the result tree. This is the part of XSL which is most closely related to CSS. Formatting objects are referred to in the XSL code as \texttt{fo}.

The main difference between CSS and XSL FO is that XSL is based on an XML format, properly defined with a DTD, and CSS is not.

Because XSL defines an extra step in presenting data, it can do much more powerful presentation tasks than CSS can. CSS always retains the order of the source tree, whereas XSL can re-order the data. A simple example might be where two stylesheets can be used to display the same data, one ordered by name, and the other ordered by location.

At the time of writing, there are only experimental implementations of XSL formatting objects, although as XML browsers become more advanced, this functionality is expected to become mainstream. Because XSL formatting objects are still mainly theoretical, we chose not to cover them in detail in this book.

So, if XSL FO is not implemented, how can you actually display XML? The most common way of doing this is to use an XSL stylesheet that transforms the XML source document into an HTML result document. Then, any HTML display engine can be used to display the result tree. Similarly, alternative stylesheets can be built that transform the source tree into other directly displayable markup languages: for example, Wireless Markup Language for mobile phones, and Precision Graphics Markup Language for printed output.

There is also a method of specifying alternate stylesheets for one document, and with the correct software these can be chosen based on the display engine, so an Internet Explorer user will see a different result than a Nokia mobile phone user might see.
Most XSL implementations that target HTML result in a combination of HTML and CSS. This breaks up the presentation of XML into two stages: structure and layout are controlled by the XSLT and resulting HTML, and the visual style and formatting are controlled by the CSS stylesheet.

An important point is that XSL does not require a DTD for either the source or result document. This means that it can be flexibly applied in many situations which would be impossible otherwise.

2.5.3 XSL Transformations

XSLT works on two principles: *pattern matching* and *templates*. Remember that XSLT transforms a source tree into a result tree. The XSLT processor takes two inputs: the XSL stylesheet and the source tree, and produces one output — the result tree. The stylesheet associate patterns with templates. The patterns identify parts of the source tree. When a match is made, the associated template is applied.

XSL stylesheets can also *filter* the input data, based on logical tests, and *re-order* the data. The templates can also contain arbitrary tags, formatting instructions or data. The result of this is that an XSL stylesheet can perform very powerful transformations on the input data. For example, a stylesheet could be used to create an HTML display of a list of bank accounts, sorted by balance, with overdrawn accounts colored red, and large balances colored green. The same data could be used with another stylesheet which graphically represented the data by transforming it into *Structured Vector Graphics* (SVG), which is an XML format for drawing graphics.

2.5.4 Relationship between XSL and XPath

As we mentioned above, XPath was defined to be a common format between XSL and XPointer. You will recall that XPath allows a particular part of an XML document to be identified. In fact it also allows a set of nodes to be identified. HTML paths must be defined by the author, but XPaths can be defined without modifying the original XML document.

In the following stylesheet example, you will see XPath references to the XML document, which allow the XSL stylesheet to extract data from the XML document.
2.5.5 An XML example

The following example shows a simple XML data structure which represents the market share of the major Web servers over time. This data is derived from the Netcraft Web Server Survey available at:

http://www.netcraft.com/survey

Here is the example:

```xml
<?xml version="1.0" ?>
<marketshare>
  <marketname>Web Server</marketname>
  <year value="1998">
    <product pname="Apache">50.6%</product>
    <product pname="Netscape">8.2%</product>
    <product pname="Microsoft">22.7%</product>
  </year>
  <year value="1997">
    <product pname="Netscape">11.8%</product>
    <product pname="Microsoft">17.7%</product>
    <product pname="Apache">43.7%</product>
  </year>
  <year value="1999">
    <product pname="Apache">55.5%</product>
    <product pname="Microsoft">22.0%</product>
    <product pname="Netscape">7.5%</product>
  </year>
</marketshare>
```

For the purpose of creating an interesting example, the data is ordered differently in each section, and the stylesheet uses selection and sorting to produce a table order by year and product name. A sample stylesheet for this is presented here:

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <HTML>
      <HEAD>
        <META http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
        <META http-equiv="Expires" content="0"/>
        <STYLE TYPE="text/css">
          .tableText{
            font-weight : bold;
            font-family : monospace;
            background-color : silver;
            text-align : right;}
          </STYLE>
      </HEAD>
      <body>
        <!-- Your XSL output goes here -->
      </body>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```
The main elements of the stylesheet are as follows:

- The namespace is defined correctly.
- The stylesheet is defined to output HTML, which is necessary, because HTML does not obey XML syntax, and so requires special handling.
- The root of the source tree is matched creating the main HTML output. A particular branch, the *marketname*, is selected to produce the heading.
- A CSS style is defined for the table text.
- A *for-each* loop pulls out the product names for titles.
- Another *for-each* loop pulls out the years, while a subsidiary loop selects the marketshare values, to create the main table data.

The results of applying this stylesheet to the data are presented below in Figure 3.
As you have seen, XSL is a very powerful way of transforming XML documents. In fact, it is nearly as powerful as a general purpose programming language. It can also be extended by using scripting in languages such as Java and JavaScript. Many real-life tasks such as transforming data between formats can be done using XSL. In many ways, XSL is the primary programming language for XML, as well as being the primary style language.
Chapter 3. Processing XML using Java

In a perfect world, computer applications would just exchange XML documents. In real life, applications often have to be able to support multiple client types, all with different capabilities. The dominant client type for Web application servers is currently a browser (usually Netscape or Internet Explorer), but it will not be like that forever. We might have cellular phones and other front-end devices, all with different XML capabilities.

The most recent edition of any browser that might have XML support cannot be a prerequisite for using an XML based Web application. We also do not want to send the same XML document to every client, because some users of the application might be authorized to see more data than others. We must have the ability to process XML documents and generate the kind of response to the client that is adequate for the client type.

On the server side, the Web application server usually connects to a back-end data store like a relational database that does not natively support data interchange using XML. We need to be able to extract the necessary information from an XML document and pass that information to the database, as well as transform the information coming from the database to XML. To fulfill both the client and the server requirements, we need an XML processor.

While the XML document format is the most natural form of data exchange in the internet, Java is the most natural language to be used in Internet applications and application servers. This is because of Java’s object-oriented and distributed nature. One technical advantage of Java over other languages is its built-in support for Unicode. With other languages, XML processing has to be done using tricks or by developing additional libraries to support Unicode in that language environment. IBM does have a C++ implementation of an XML parser, as well as supporting Unicode libraries, but, to summarize, Java is an excellent language in implementing XML processors and other XML related tools and applications.

3.1 XML applications

At the heart of every XML application is an XML processor that parses the well-formed XML document, so that the document elements can be retrieved and transformed into data that can be understood by the application and task in hand. The other responsibility of the parser is to check the syntax and structure (validity and well-formedness) of the document.
Anyone has the freedom to implement a parser that can read and print an XML document. The XML 1.0 Recommendation defines how an XML processor should behave when reading and printing a document, but the API to be used is not defined. However, there are standards that define how XML documents should be accessed and manipulated. Currently, the following two APIs used are widely used:

- Simple API for XML
- Document Object Model

### 3.2 SAX

Simple API for XML (SAX) was developed by David Megginson and a number of people on the xml-dev mailing list on the Web, because a need was recognized for simple, common way of processing XML documents. As such, SAX 1.0 is not a W3C recommendation, but it is the de-facto standard for interfacing with an XML parser, with many commonly available Java parsers supporting it.

SAX is an event-driven lightweight API for accessing XML documents and extracting information from them. It cannot be used to manipulate the internal structures of XML documents. As the document is parsed, the application using SAX receives information about the various parsing events. The logical structure of an application using SAX API with the parser is shown in Figure 4.

![Figure 4. SAX application components](image-url)
The SAX driver can be implemented by the XML parser vendor, or as an add-on to the parser. That makes the application using the parser via SAX independent of the parser.

### 3.2.1 SAX classes and interfaces

The SAX 1.0 API defines two sets of interfaces, one of which is meant to be implemented by XML parsers, and one by XML applications. The interfaces that parsers have to implement are:

- Parser
- AttributeList
- Locator (optional)

The first thing an XML application has to do is to register SAX event handlers to a parser object that implements the `Parser` interface. As the XML document is processed by a parser, SAX notifies the application whenever an event occurs. The events that can be captured depend on the registered event handlers, the interfaces of which are:

- DocumentHandler
- DTDHandler
- ErrorHandler

The most important and commonly used interface is DocumentHandler, because it can be used to track basic document-related events like the start and end of elements and character data. The events occur in the order that is directly related to the order of elements that are found in the tree-formed XML document that is being parsed. DTDHandler notifies the application about unparsed external entity declarations or when a notation declaration is encountered. ErrorHandler notifies the application whenever an error occurs while parsing the XML document.

The SAX specification also provides `HandlerBase` class which implements all interfaces and provides default behavior. Instead of implementing the appropriate interfaces, an XML application can extend the `HandlerBase` class and override just the methods that need to be customized.

The Java implementation of SAX is organized in two Java packages:

- `org.xml.sax`
- `org.xml.sax.helpers`
The first of the above-mentioned packages contains the SAX core implementation classes, interfaces and exceptions. The second one contains convenience classes and a Java-specific class (ParserFactory) for dynamically loading SAX parsers.

The implementation can be downloaded from http://www.megginson.com/SAX/. The same location also contains full descriptions (in JavaDoc format) of all classes and interfaces defined in SAX 1.0.

3.2.2 SAX example

For a Java application to be able to use SAX, we need a class that implements an interface most suitable for the job. The following code fragment shows the relevant methods of DocumentHandler that are implemented to track start and end of elements and the whole document. It also prints out the actual data within the elements:

```java
public class MyDocHandler implements org.xml.sax.DocumentHandler {

    @Override
    public void characters(char[] arg1, int start, int length) throws org.xml.sax.SAXException {
        System.out.println(new String(arg1, arg2, arg3));
    }

    @Override
    public void startDocument() throws org.xml.sax.SAXException {
        System.out.println("Start of document");
    }

    @Override
    public void endDocument() throws org.xml.sax.SAXException {
        System.out.println("End of document");
    }

    @Override
    public void startElement(String name, org.xml.sax.AttributeList arg2) throws org.xml.sax.SAXException {
        System.out.println("Start of element "+ name);
    }

    @Override
    public void endElement(String name) throws org.xml.sax.SAXException {
        System.out.println("End of element "+ name);
    }
}
```

The application that uses the DocumentHandler implementation above is simple. IBM’s XML for Java implements the Parser interface in SAXParser class, which the following example uses:

```java
Parser parser = ParserFactory.makeParser("com.ibm.xml.parsers.SAXParser");
SampleDocumentHandler hndlr = new SampleDocumentHandler();
parser.setDocumentHandler(hndlr);
parser.parse(anXMLFileURL);
```

38 The XML Files: Using XML and XSL with IBM WebSphere 3.0
Given the following XML document as input:

```xml
<?xml version="1.0"?>
<personnel>
    <person id="jedi1">
        <name>
            <lastname>Skywalker</lastname>
            <firstname>Luke</firstname>
        </name>
    </person>
</personnel>
```

The output of the SAX application looks like this:

```
Start of document  
Start of element personnel  
Start of element person  
Start of element name  
Start of element lastname  
Skywalker  
End of element lastname  
Start of element firstname  
Luke  
End of element firstname  
End of element name  
End of element person  
End of element personnel  
End of document  
```

### 3.3 DOM

While XML is a language to describe tree-structured data, the Document Object Model (DOM) defines a set of interfaces to access tree-structured XML documents. DOM specifies how XML and HTML documents can be represented as objects. Unlike SAX, DOM also allows creating and manipulating the contents of XML documents. Basically, the DOM interfaces are platform and language neutral. The current W3C DOM Level 1 Specification provides bindings for Java, OMG IDL, and ECMAScript (formerly JavaScript).

The origins of DOM are in the need to dynamically render HTML content (DHTML). The current DOM Level 1 Recommendation has two parts: Core and HTML. Core contains fourteen interfaces, seven of which are applicable to both HTML and XML documents. Six remaining interfaces are specific to XML. DOM HTML defines additional convenience methods that are useful for client side scripting.
3.3.1 DOM hierarchy

The DOM API is a set of interfaces that must be implemented by a DOM implementation such as IBM’s XML for Java. The interfaces, being originally described in IDL, form a hierarchy (see Figure 5).

The root of the inheritance tree is Node, that defines the necessary methods to navigate and manipulate the tree-structure of XML documents. The methods include getting, deleting, and modifying the children of a node, as well as inserting new children to it. Document represents the whole documents, and the interface define methods for creating elements, attributes, comments, and so on. Attributes of a Node are manipulated using the methods of the Element interface. DocumentFragment allows extracting parts of a document. It should be noticed that while a DOM application reads an XML document and an object representation if formed, that representation remains only in memory. Changing a DOM object in memory does not automatically modify the original file. That is something an application program has to do for itself.

Figure 5. DOM interface hierarchy

The W3C DOM Level 1 Recommendation can be found at http://www.w3.org/DOM/.
3.3.2 DOM example

When the simple XML document we used in our SAX example (see listing on page 39) is processed using DOM, the resulting object tree will look like the one in Figure 6. The shaded rectangles represent character data, and the others represent elements.

```
personnel

person id="jedi1"

name

  lastname
  Skywalker

  firstname
  Luke
```

Figure 6. Sample DOM tree

Reading an XML document using DOM is relatively easy, provided that a good parser is available. Among other things, IBM's XML Parser for Java provides a robust and very complete implementation of the W3C DOM API. The following code fragment shows a simplified example of how to read and manipulate an XML document using the DOMParser class:

```java
DOMParser parser = new DOMParser();
parser.parse(uri);
Document document = parser.getDocument();
print(document); // implemented in our own code
Node n = document.getLastChild().getFirstChild();
n.setNodeValue("ZAP! You're history!");
print(document);
```

3.3.3 DOM Level 2

Some important facilities that are missing from the DOM Level 1 Recommendation are being defined in DOM Level 2, which is currently a W3C Specification working draft (July 19, 1999). The added functionality in Level 2 contains interfaces for creating a document, importing a node from one document to another, supporting XML Namespaces, associating stylesheets with a document, the Cascading Style Sheets object model, the Range object model, filters and iterators, and the Events object model.
3.4 SAX or DOM?

There are certainly applications that could use either SAX or DOM to get the necessary functionality needed when processing XML documents. However, these two approaches to XML processing each have their strengths and weaknesses.

3.4.1 SAX advantages and disadvantages

SAX provides a standardized and commonly used interface to XML parsers. It is ideal for processing large documents whose content and structure does not need to be changed. Because the parser only tells about the events that the application is interested in, the application is typically small, and has a small memory footprint. This also means that SAX is fast and efficient, and a good choice for application areas such as filtering and searching, where only certain elements are extracted from a possibly very large document.

Because the events must be handled as they occur, it is impossible for a SAX application, for example, to traverse backwards in the document that is under processing. It is also beyond SAX’s capabilities to create or modify the contents and internal structure of an XML document.

3.4.2 DOM advantages and disadvantages

Because every element of an XML document is represented as a DOM object to the application using the DOM API, it is possible to make modifications to the original XML document. Deleting a DOM node means deleting the corresponding XML element and so on. This makes DOM a good choice for XML applications that want to manipulate XML documents, or to create new ones.

DOM is not originally an event driven API like SAX, even though the DOM Level 2 draft specifies events. To extract even a small piece of data from an XML document, the whole DOM tree has to be created. There is no way of creating lightweight applications using DOM. If the original XML document is large, the DOM application that manipulates the document requires a lot of memory. In practice, DOM is mostly used only when creating or manipulating XML documents is a requirement.
Part 2. Applying XML technology in IBM WebSphere
WebSphere Application Server is IBM's offering to the e-business marketplace. WebSphere Standard Edition is capable of running servlets, Java Beans and Java Server Pages, all of which can connect to back-end resources such as databases. In addition to that, WebSphere Advanced Edition gives you the ability to run Enterprise JavaBeans. Both editions provide a set of XML related tools and utilities that are packaged into what is called XML Document Structure Services. These services consist of document parsers, document validators, and document generators for server-side XML. The Document Structure Services support:

- W3C Extensible Markup Language (XML) 1.0
- W3C Namespaces in XML (Recommendation January 14, 1999)
- W3C Level 1 Document Object Model Specification (DOM) 1.0 (Recommendation October 1, 1998)
- XSL Transformations Version 1.0
- XML Path Language Version 1.0
- Microstar SAX 1.0

The XML Document Structure Services in WebSphere is a package of three components:

- XML for Java Parser
- LotusXSL
- DTD Catalogs

### 4.1 XML for Java Parser

XML Parser for Java (shipped in XML4J.JAR) is a validating XML parser written in 100% pure Java. The package (com.ibm.xml.parser) contains classes and methods for parsing, generating, manipulating, and validating XML documents. It conforms to XML Spec 1.0, and supports the latest DOM (Level 1.0 currently), SAX, and namespace specifications.

The latest version of XML Parser for Java is available at the IBM Alphaworks Web site:

http://www.alphaworks.ibm.com
4.2 LotusXSL

LotusXSL (shipped in LOTUSXSL.JAR) contains the APIs for the Lotus XSL processor. It supports the August 1999 W3C XSLT and XPath Working Drafts, and it can be used whenever XSLT transformations are needed. This is the case, for example, when an XML document needs to be converted into an HTML document using a stylesheet. LotusXSL also includes a separate XPath package and an XML Query interface.

The latest version of LotusXSL is also available at the IBM Alphaworks Web site:

http://www.alphaworks.ibm.com

4.3 DTD Catalogs

To make it easier to use industry-standard DTDs and other grammars, local copies are installed with the Application Server. Some of the grammars in the library are:

- Channel Definition Format (CDF)
- Mathematics Markup Language (MathML)
- Wireless Markup Language (WML)

Developed by Microsoft Corporation, CDF enables subscription to Web-based channels (a Web site or a portion of a Web site). The subscription can be implemented to have the Web site automatically send the subscriber updates to the channel (using push technology) or transmit the updates at the request of the subscriber (using pull technology). In either case, the subscriber must be able to link to a CDF file on the channel site. The CDF file is an XML document that conforms to the CDF DTD.

Developed by the W3C, MathML is a grammar for documents that contain math formulas, notations, and other data.

WML is a grammar for creating documents so that they can be efficiently transmitted and displayed on narrowband devices, such as cellular phones, personal digital assistants (PDAs), and pagers.
Chapter 5. XML Parser for Java

An XML parser is a piece of software that can read a stream of XML data, process it, notify errors, generate a structured tree, and expose all data elements as objects using the Document Object Model (DOM). IBM’s XML Parser for Java (XML4J) is a robust validating XML processor that is written in 100% pure Java.

An alternative to using DOM APIs is to use event-driven APIs. The difference between the processing of DOM and event driven APIs is that the former method is recursive and the latter method is single-pass for the entire process. One of the event-driven APIs is SAX (Simple API for XML). In contrast to DOM, SAX is a set of lightweight APIs, and it does not generate any internal structures. Mainly, we will look at the parsing of XML document using a DOM compliant XML processor. Parsing using SAX has been covered in brief in previous chapters.

Let us consider this XML document (dir.xml) and its corresponding DTD (directory.dtd) for parsing.

```xml
<?xml version="1.0" ?>
<!DOCTYPE directory SYSTEM "directory.dtd">
<directory>
  <entry>
    <name>Ramani</name>
    <address>
      <street>Harry Road</street>
      <city>San Jose</city>
      <state>California</state>
      <country>USA</country>
    </address>
    <tel>4089273679</tel>
  </entry>
  <entry>
    <name>Routray</name>
    <address>
      <street>B K Road</street>
      <city>Cuttack</city>
      <state>Orissa</state>
      <country>India</country>
    </address>
    <tel>671643135</tel>
  </entry>
</directory>
```
The DTD corresponding to the above XML document is as follows:

```xml
<!ELEMENT directory (entry*)>
<!ELEMENT entry (name,address,tel*)>
<!ELEMENT name (#PCDATA)
<!ELEMENT address (street,city,state,country)>
<!ELEMENT street (#PCDATA)
<!ELEMENT city (#PCDATA)
<!ELEMENT state (#PCDATA)
<!ELEMENT country (#PCDATA)
<!ELEMENT tel (#PCDATA)
```

5.1 How to parse an XML document using DOM

Before we start parsing the XML document, we should have the following environment set up complete.

- JDK installed
- XML parser for Java installed
- set the CLASSPATH to contain the parser’s “.jar” file.

The examples and the descriptions regarding the parsing are described with respect to IBM’s XML4J Ver2.0.15. Let us take a overview of this parser. This contains the following packages which suggest their content by their names.

- com.ibm.xml.dom
- com.ibm.xml.framework
- com.ibm.xml.parsers
- org.w3c.dom
- org.xml.sax
- org.xml.sax.helpers

The `com.ibm.xml.parsers` package contains the different types of parsers such as validating, non-validating, revalidating, and SAX validating. To be exact, the classes contained by this package are `DOMParser`, `NonValidatingDOMParser`, `NonValidatingParser`, `NonValidatingTXDOMParser`, `RevalidatingDOMParser`, `SAXParser`, `TXDOMParser`, `TXRevalidatingDOMParser`, `ValidatingParser`, `ValidatingSAXParser`. Table 3 highlights the difference between validating and non-validating parsers.
Now, we will write a Java program to parse an XML document. First we will try to get the root of the document, and then we will print the name, value, and type of the nodes of the document.

To write the Java program, we need to import two classes from the parser. These are org.w3c.dom and com.ibm.xml.parsers. The interface Document in the package org.w3c.dom represents the entire XML document according to the Document Object Model (DOM). It is the root of the document tree. Here is a simple coding example to get the root of an XML document and to print the node name and value.

```java
import org.w3c.dom.*;
import com.ibm.xml.parsers.*;
import java.io.*;

public class ParseXML {
    public static void main(String argv[]){
        if(argv.length != 1){
            System.out.println("Usage: java ParseXML filename.xml");
            System.exit(1);
        }
        try{
            Document doc = null;
            // trying to create instance of a validating DOM parser
            DOMParser parser = new DOMParser();
            // parses the file specified by command line parameter
            parser.parse(argv[0]);
            doc = parser.getDocument();
        }
```
The XML Files: Using XML and XSL with IBM WebSphere 3.0

```java
import org.w3c.dom.*;
import com.ibm.xml.parsers.*;
import java.io.*;

public class ParseXML {
    public static void main(String argv[]){
        if(argv.length != 1){
            System.out.println("Usage: java ParseXML filename.xml");
            System.exit(1);
        }
        try{
            Document doc = null;
            DOMParser parser = new DOMParser();
            parser.parse(argv[0]);
            doc = parser.getDocument();
            Element root = (Element)doc.getDocumentElement();
            traverse(root);
        }
        catch(FileNotFoundException e1){
            e1.printStackTrace();
        }
        catch(Exception e2){
            e2.printStackTrace();
        }
    }
}
```

We will now see, how to traverse the parse tree using the DOM interfaces. First, let us take a look at how to get the name and value of the nodes.

```
Element rootNode = (Element)doc.getDocumentElement();
System.out.println("Node Name: "+rootNode.getNodeName();
System.out.println("Node Value: "+rootNode.getNodeValue();
```

Element is an abstract interface and extends the Node interface. The API 
getDocumentElement returns the root element (if exists else returns null). The APIs getDocumentName() and getDocumentValue() return the node name and node value respectively. We can get different related nodes directly by using the APIs. For example, the getFirstChild() returns the first child of the node. The getLastChild() API returns the last child of the node. The getNextSibling() API returns the next immediate node following the node. If any such node does not exist, then it returns null. For a detailed list of APIs, you can take a look at the XML4j documentation.

Let us look at the way to traverse the DOM tree recursively, and to print the type, name, and value of each node in this XML document.

```
import org.w3c.dom.*;
import com.ibm.xml.parsers.*;
import java.io.*;

public class ParseXML {
    public static void main(String argv[]){
        if(argv.length != 1){
            System.out.println("Usage: java ParseXML filename.xml");
            System.exit(1);
        }
        try{
            Document doc = null;
            DOMParser parser = new DOMParser();
            parser.parse(argv[0]);
            doc = parser.getDocument();
            Element root = (Element)doc.getDocumentElement();
            traverse(root);
        }
```
public static void traverse(Node rn){
    System.out.println("Node Name : "+rn.getNodeName());
    System.out.println("Node Value : "+rn.getNodeValue());
    int type = rn.getNodeType();
    switch(type){
        case Node.DOCUMENT_NODE:
            System.out.println("Type: Document Node");
            break;
        case Node.ELEMENT_NODE:
            System.out.println("Type: Element Node");
            break;
        case Node.ENTITY_REFERENCE_NODE:
            System.out.println("Type: Entity Reference Node");
            break;
        case Node.CDATA_SECTION_NODE:
        case Node.TEXT_NODE:
            System.out.println("Type: CDATA Section/Text Node");
            break;
    }
    if(rn.hasChildNodes()){
        NodeList nList = rn.getChildNodes();
        int childrenCount = nList.getLength();
        for(int i=0;i<childrenCount;i++){
            traverse(nList.item(i));
        }
    }
}

In our examples above, we used DOMParser. You can instantiate the Parser in another way too. This second way of instantiating a Parser can be helpful when the application needs to switch between different parser configurations.

import org.xml.sax.Parser;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.DOMParser;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
import java.io.IOException;

public class ParseXML{

    public static void main(String argv[]){
        String parserClass = "com.ibm.xml.parsers.RevalidatingDOMParser";
        String xmlFile = "dir.xml";
        Parser parser = ParserFactory.makeParser(parserClass);
        try{
            parser.parse(xmlFile);
        }catch( Exception e ) {
            e.printStackTrace();
        }
    }//end method main
}//end class ParseXML

The Parser leaves the white-space processing up to the choice of the application. APIs like `ignorableWhiteSpace(parameters...)` give the application the liberty to choose whether white spaces are to be ignored or not.

The validation of the document and error reporting is also left to the application. For example, the APIs `setWarningOnDuplicateAttDef (flag)` and `getWarningOnDuplicateAttDef ()` give the application the choice of setting the error reporting off or on, and they also allow you to enquire about the previous setting.

In another example, if the application needs to get a warning regarding the missing of DTD declaration, then:

- The API `com.ibm.xml.parser.Parser.setWarningNoDoctypeDecl(boolean isWarningNoDoctypeDecl)` can set the option.
- The API `com.ibm.xml.parser.Parser.setWarningRedefinedEntity(boolean isWarningRedefinedEntity)` sets whether the parser treats redefined entities as a warning.
- The API `com.ibm.xml.parser.Parser.setEndBy1stError(boolean isEndBy1stError)` sets whether the parser terminates or continues processing when an initial error or exception occurs.

Regarding error handling, when an instance of a parser is created, the default error handler does nothing. You should register an error handler with the
parser by supplying a class which implements the `org.xml.sax.ErrorHandler` interface. This is true regardless of whether the parser is DOM based or SAX based.

### 5.2 How to create an XML document using DOM

Let us discuss how to generate a DOM tree. The set of DOM APIs allows you to build a well-formed DOM tree. To create an internal structure from scratch, you need to create the `Document`, regarding which we have already discussed. This `Document` interface represents the XML document. Conceptually, it is the root of the document tree, and provides the primary access to the document's data. Here is a small example:

```java
import org.w3c.dom.*;
import com.ibm.xml.dom.*;

............... //get the document
//DocumentImpl doc = new DocumentImpl();
//create an element
Element root = doc.createElement("NODE");
//append the element
doc.appendChild(root);
```

The version of the document, encoding parameter, and so on, can also be set using the set of APIs provided. These DOM structures can also be manipulated using the DOM APIs. Operations such as inserting a child node, deleting a child node, replacing a child node, removing all child nodes, and obtaining next and previous siblings, are all functions of the APIs that help in manipulating DOM structures. For detailed listings of these APIs, see the API documentation that comes with the XML4J package.

### 5.3 Reading and querying DTD information

In the previous section, we saw how to add elements and construct a DOM structure. There has to be a DTD, so that the validity of the structure to be created can be checked by the XML processor. The XML processor gets the information regarding the DTD from the contents of the `<!DOCTYPE ...>` tag in the XML document. The DTD can be read, and information from the DTD can also be queried. Here is an example of coding to read the DTD:

```java
String fileName = "directory.dtd"
FileInputStream fis = new FileInputStream(fileName);
```
Parser parser = new Parser(fileName);
DTD dtd = parser.readDTDSStream(fis);

In case the DTD file is available at a particular URL, then only the second line of the code needs to be modified to look like the following:

String fileURL = "http://........./directory.dtd";
InputStream is = (new URL(fileName)).openStream();

The class com.ibm.xml.parser.DTD contains a set of APIs which helps in retrieving relevant information regarding the DTD, and it also queries on the validity of extending a structure based upon this DTD. This DTD class implements the DocumentType interface defined by DOM.

5.4 Using namespaces

There are no standard APIs for namespace manipulation in the standard DOM and SAX packages. To get the support of namespaces, we can use TX compatibility classes. These TX compatibility classes provide additional, non-standard APIs to work with namespaces. Currently, the behavior of validating parsers is not defined when namespaces are in use.
Chapter 6. LotusXSL

LotusXSL is an XSLT processor, and as such, is an essential component in WebSphere Application Server, whenever XML documents need to be transformed into another form, such as other XML documents or HTML documents. At the time of writing, LotusXSL (version 0.18.2, September 7, 1999) implements the XSL Transformations (XSLT) Version 1.0 (3C Working Draft, August 13, 1999). In addition to XSLT, XSL includes an XML vocabulary for specifying formatting semantics called Formatting Objects, which are not supported by the current version of LotusXSL.

LotusXSL is not a product (yet), and it implements a standard that is a draft, so changes both in the standard and LotusXSL implementation are likely to happen. The current version uses Version 2.0.15 (preferred) or 1.1.16 of IBM's XML for Java, but it can interface to any Java DOM-tree-producing processor.

6.1 So, what is an XSLT processor?

When every piece of data that is exchanged between computers is expressed in terms of XML documents, and when we use XML processors to evaluate and modify that data, it should be enough, right? Well, not exactly. Even in case we only want to transform an XML document into another XML document, it is often desirable to have a mechanism that would produce the right output without writing a program that traverses the whole DOM tree and makes the modifications. This is exactly where XML stylesheets can be utilized. A stylesheet is a transformation expressed in XSLT, and XSL transformation describes the rules of how a result tree is built based on the source tree.

LotusXSL is an XSLT processor that implements an API for associating XML documents with stylesheets and producing the required output. The API (described in more detail in “LotusXSL API” on page 62) makes it relatively easy to write a program that does the translations. Moreover, LotusXSL ships with a servlet (DefaultApplyXSL), that can be used directly in the server to do the XSL transformations. This servlet is especially handy when HTML transformations are required. LotusXSL also includes an applet capable of hosting the XSL processor, in case client-side XSL processing is needed. Also, a batch file is provided for processing XML/XSL files using a command prompt.

The complete and current documentation of LotusXSL can be found at the IBM Alphaworks Web site (http://www.alphaworks.ibm.com).
6.2 XSLT basics

The basic principle of how XSLT works is based on associating patterns of the source tree against templates, which describe how the result tree should be built. While processing the source tree, whenever a pattern is found that matches an element in the source tree, the respective template is activated. Each template defines how the elements and attributes in the result tree are generated.

Note

If you wish to try the samples in this section yourself, please refer to 6.3, “LotusXSL API” on page 62 for instructions on how to use LotusXSL.

A simple example is shown in Table 4:

Table 4. Simple example of XML with XSL

<table>
<thead>
<tr>
<th>XML document</th>
<th>XSL template</th>
</tr>
</thead>
</table>

The output of the XML with the XSL applied should be like the following:

The document heading is:
Intro
Basics

*******************************************************************************
This is very simple
*******************************************************************************
As you can see, each XSL document is also a valid and well-formed XML document. XSLT processors use the XML namespaces mechanism to recognize elements defined by XSLT. The namespace is defined with the URI http://www.w3.org/XSL/Transform/1.0, and all elements defined by XSLT (starting with the prefix xsl:) have to belong in that namespace in order to be recognized by the XSLT processor.

The XSL document includes two templates, one for matching DOCUMENT elements and one for TEXT elements. If a DOCUMENT element is found in the source tree, the string “The document heading is:” is appended, followed by the value of the source element. In case a TEXT element is found, the value is output to the result tree, with a sequence of ** characters above and under the value. So who is responsible for the *Intro* text in the result tree? That is the responsibility of the default template rule, which takes the form:

```xml
<xsl:template match="/|
"/>>
<xsl:apply-templates/>
</xsl:template>
```

The template above matches with the root element or any other element. This means that if a more specific rule does not exist, the value of the element is output to the result tree as-is.

### 6.2.1 Accessing attributes

Notice that the attribute values are not matched by using the default template. The example in Figure 7 shows how to retrieve attribute values.

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
  <xsl:template match="DOCUMENT | PARA">
    The heading is: <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="TEXT">
    ***********************
    The following text is <xsl:value-of select="@TYPE"/>
    <xsl:apply-templates/>
    ***********************
  </xsl:template>

</xsl:stylesheet>
```

*Figure 7. XSL with attributes*
Given the XML document in Table 4 on page 56, the output should be:

The heading is: Intro
The heading is: Basics
*******************************
The following text is CONFIDENTIAL
This is very simple
*******************************

### 6.2.2 Conditional processing

The next introductory example shows conditional processing. Suppose we wish to output the text only if the TEXT TYPE attribute in our example XML document is not marked CONFIDENTIAL. To implement this, you have a few options, one of which is shown here:

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
    <xsl:template match="DOCUMENT">
        <xsl:apply-templates select="PARA"/>
    </xsl:template>

    <xsl:template match="PARA">
        The paragraph heading is: <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="TEXT">
        <xsl:choose>
            <xsl:when test='@TYPE="CONFIDENTIAL"'>
                *******************************
                This text is CONFIDENTIAL so it is not displayed, sorry
                *******************************
            </xsl:when>
            <xsl:otherwise>
                This following text is CONFIDENTIAL, the <xsl:apply-templates/> tag is not applied.
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
</xsl:stylesheet>
```

In the above stylesheet, the value of the TYPE attribute is tested, and if it is found to be CONFIDENTIAL, the <xsl:apply-templates/> tag is not applied.
When the above XSL is applied to the XML document in Table 4 on page 56, the output should be the following:

The paragraph heading is: Basics
*******************************
This text is CONFIDENTIAL
so it is not displayed, sorry
*******************************

If, in your XML document, you change the TYPE attribute to something other than CONFIDENTIAL, the results should change accordingly:

The paragraph heading is: Basics
The following text is SOOO CONFIDENTIAL ...
This is very simple

Also notice the template for DOCUMENT elements: the element itself is not displayed at all. Only the PARA child elements are selected for processing.

6.2.3 Repetition

By now, you should be familiar with the basic idea of how to build a result tree based on the source tree using templates. The examples given in 6.2, “XSLT basics” on page 56 all used the approach of multiple, relatively small templates, each of which is only applied to a certain element. This approach is familiar to functional programmers and results in easily understandable and easily maintainable stylesheets.

XSL is a language (remember: eXtensible Stylesheet Language), and as with any language, in some cases you can reach the same end result using alternative approaches, such as building your stylesheet as just one big template, with the details embedded within the main matching template. One possible case for using this approach is when you have a number of similar elements that you want to process in a loop. For that purpose, you can use the <xsl:for-each> tag, as shown in Figure 8:

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
  <xsl:template match="/">
    <H1>Simple &lt;xsl:for-each&gt; template</H1>
    <TABLE BORDER="1"><TBODY>
      <xsl:for-each select="DOCUMENT/PARA">
        <TR>
          <TH>Paragraph</TH>
        </TR>
      </xsl:for-each>
    </TBODY>
  </xsl:template>
</xsl:stylesheet>
```
This example builds an HTML table with each PARA element presented as one row, each TEXT element as a cell in that row. This approach is suitable for situations where the number of elements is known, and you want to transform these elements into form that has a known regular structure. It is worth noticing, however, that the same result can be achieved with the ‘small template’ approach as well. It is up to you to take the approach that suits you best. The stylesheet below builds the same result tree as the previous one:

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

  <xsl:template match="DOCUMENT">
    <H1>Simple &lt;xsl:for-each&gt; template</H1>
    <TABLE BORDER="1">
      <xsl:apply-templates select="PARA"/>
    </TABLE>
  </xsl:template>

  <xsl:template match="PARA">
    <TR>
      <TD>Paragraph</TD>
      <xsl:apply-templates select="TEXT"/>
    </TR>
  </xsl:template>

  <xsl:template match="TEXT">
    <TD>
      <xsl:apply-templates/>
    </TD>
  </xsl:template>

</xsl:stylesheet>
```
6.2.4 Creating attributes and elements

If your result document is to be a valid XML document, you have to be very careful in inserting your own text elements into the result tree. In the previous example we added the necessary HTML tags by simply entering them inside the templates. It is also possible to create attributes and elements with a computed name, by using xsl:element and xsl:attribute tags. The following example shows you how:

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
    <xsl:template match="DOCUMENT">
        <xsl:element name="H1">Simple &lt;xsl:for-each&gt; template</xsl:element>
        <xsl:element name="TABLE">
            <xsl:attribute name="BORDER">1</xsl:attribute>
            <xsl:apply-templates select="PARA"/>
        </xsl:element>
    </xsl:template>

    <xsl:template match="PARA">
        <xsl:element name="TR">
            <xsl:element name="TD">Paragraph</xsl:element>
            <xsl:apply-templates select="TEXT"/>
        </xsl:element>
    </xsl:template>

    <xsl:template match="TEXT">
        <xsl:element name="TD">
            <xsl:apply-templates/>
        </xsl:element>
    </xsl:template>

</xsl:stylesheet>

This stylesheet applied to an XML document gives the same result as the for:each template shown in Figure 8 on page 60.
6.3 LotusXSL API

For basic XSL processing, LotusXSL provides an easy-to-use Java API, with the primary public entry points implemented as the process(...) methods of the class XSLProcessor, which is responsible for processing an input source tree through a stylesheet. Because stylesheet and XML documents can come from various sources (URL, DOM, stream...), and the output can take many forms as well, there are currently over 30 process-methods to choose from. If you wish to write a Java program to process the examples shown in 6.2, “XSLT basics” on page 56, your program could look like the following:

```java
import org.xml.sax.SAXException;
import com.lotus.xpath.XPathException;
import com.lotus.xsl.XSLProcessor;
import java.io.PrintWriter;
import java.io.FileWriter;
public class Transformer {
    public static void main(String[] args)
    throws java.io.IOException, java.net.MalformedURLException,
          org.xml.sax.SAXException {
        XSLProcessor processor = new XSLProcessor();
        PrintWriter pw = new PrintWriter( new FileWriter("test.out") );
        processor.process("mydoc.xml", "myxsl.xsl", pw);
    }
}
```

The above example uses a variation of the process-method that takes the file names as arguments and writes the output to a PrintWriter instance. The XML and XSL documents could just as well be referenced by a URL (for example, http://localhost/mydoc.xml). Because there are over thirty other process methods, they are not covered here. Please refer to the LotusXSL documentation for a description of all available methods.

---

**Tip!**

If you feel that your tight schedule does not allow writing Java programs, and you still want to test how the LotusXSL processor works with the samples, you can use the RUN-utility that is included in LotusXSL. Just make sure that LOTUSXSL.JAR and XML4J.JAR files are included in your CLASSPATH, and the directory where LotusXSL is installed is in your PATH. RUN command without arguments tells you what it expects, but here is the most basic syntax:

```
RUN -IN mydoc.xml -XSL myxsl.xsl
```
6.3.1 Integrating a parser with the XSL processor

Although LotusXSL uses the IBM XML Parser for Java, the main XSLT and XPath engines are meant to be independent from any given DOM implementation or XML implementation. XMLParserLiaison defines the interface which should be implemented in order to enable any given XML parser to interact with the XSL processor. The implementations shipped with LotusXSL include ProcessXSL, XML4JLiaison, and XML4JLiaison4dom classes, which act as an interface to the IBM XML Parser for Java.

6.3.2 DefaultApplyXSL

The LotusXSL package that is available on the IBM Alphaworks site includes a servlet called DefaultApplyXSL, which lets you apply an XSL stylesheet to XML documents on the server. Using this servlet, the clients — usually browsers — do not need any native XML support, since the XML documents can be converted to HTML or other mark-up languages on-the-fly. Your applications can be still developed with pure XML in mind, and when native XML browsers are commonly available and a viable alternative, you can just take this conversion servlet out of business. Of course it could still be used to convert XML documents into other XML documents, which is often required; you could let this servlet act as a filter to send different XML documents to different classes of clients, such as customers, business partners, quests, and so on.

For performance reasons, DefaultApplyXSL does not try to validate XSL or XML documents, so you should validate the documents using other services, such as XML Parser for Java. In case you apply XSL stylesheets to dynamic XML documents, the validation does not necessarily ever happen, so you might get run-time errors. Also, for performance reasons, only HTTP/GET requests are processed.

6.3.3 Configuring DefaultApplyXSL

In order to be able to run the DefaultApplyXSL servlet, you need version 2.x XML Parser for Java, and a servlet engine which implements a 2.x-level servlet API, such as WebSphere Application Server 3.0. The JAR files of both LotusXSL (lotusxsl.jar) and XML Parser for Java need to be placed in location accessible by the servlet engine. In case of WebSphere Application Server, the jar files are already in the right location. This version of the LotusXSL jar file does not, however, contain the DefaultApplyXSL class, so you have to download a full version of LotusXSL from Alphaworks and replace the jar file.
Because the servlet specification does not yet specify how to configure a servlet, the actual configuration depends on the servlet engine that you are using. In case of WebSphere Application Server 3.0, you can select an existing application, or create a new one, and add a new servlet to the application using the values shown in Table 5.

Table 5. DefaultApplyXSL

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Value</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servlet name</td>
<td>DefaultApplyXSL</td>
<td>Yes</td>
</tr>
<tr>
<td>Description</td>
<td>Apply XSL to XML</td>
<td>No</td>
</tr>
<tr>
<td>Servlet class</td>
<td>com.lotus.xsl.server.DefaultApplyXSL</td>
<td>Yes</td>
</tr>
</tbody>
</table>

If you wish to configure the DefaultApplyXSL servlet to process all requests for XML documents automatically, you should configure your server accordingly. Usually this is accomplished by defining a mime-filter or an alias. In the ITSO XML Application, we used a mime-filter. The configuration process is described in detail in the B.2, “Configuring WebSphere Application Server 3.0” on page 209.

DefaultApplyXSL is also capable of recognizing the user agent. Using this capability, it is possible to send different data to different classes of browsers, such as MicroSoft and Netscape. The configuration involves using the media.properties file that is shipped with LotusXSL.

6.3.4 Running DefaultApplyXSL

Once the servlet is installed and the servlet engine is running, you can apply a stylesheet on your XML documents on the server. If the application path in WebSphere Application Server 3.0 is /xmltest, your XML file is called test.xml, and your XSL stylesheet is called test.xsl, with both the XSL and the XML files located in the root document directory of the HTTP server, you can send requests to the server using the following HTTP requests:

http://localhost/xmltest/servlet/DefaultApplyXSL/test.xml?xslURL=/test.xsl
http://localhost/xmltest/servlet/DefaultApplyXSL/test.xml

The last example in the above list assumes that test.xml file has an associated stylesheet that is referred to in the XML file.

If you set up a mime filter for XML documents, you will be able to address your XML files directly. For example, the request http://localhost/test.xml could be directed to DefaultApplyXSL servlet for processing.
Chapter 7. WebSphere and XML approaches

This chapter describes a number of approaches to using XML and XSL inside WebSphere Application Server. It includes a short overview of the WebSphere programming model, approaches to generating XML, and approaches to using XSL inside WebSphere. The chapter concludes with a comparison of approaches.

7.1 WebSphere programming model

In Chapter 5, “XML Parser for Java” on page 47, we covered the underlying technology for parsing and generating XML in Java. WebSphere Application Server supports three kinds of Java programming objects: Java Servlets, Java Server Pages (JSPs), and Enterprise Beans. Using XML inside Enterprise Beans is really no different from using XML inside any Java program, and so we have chosen not to cover Enterprise Beans in this book.

7.1.1 Servlets

Servlets are Java objects which generate an output stream based on input parameters. Generally these are used to generate HTML, which is then sent to a client browser over HTTP. However, servlets can generate any kind of output stream, including XML.

Servlets do not need to be called by a browser — other servers, Java programs or Java Applets can call servlets over HTTP and receive XML data in this way. Because XML is easy to parse, and can form a publishable interface between systems, this is an easy mechanism for system-to-system communications. We will present an example of this in Chapter 9, “XML application implementation” on page 97, where we use XML over HTTP as a business-to-business communications mechanism. For more information on servlets, see the JavaSoft specification at:


7.1.2 Java Server Pages

Java Server Pages are textual documents that contain special “dynamic” tags. These tags serve as templates for dynamic data which is inserted at runtime by the server. JSPs are requested by the user from the Web server, and WebSphere evaluates them, then converts them into a static output stream, which is sent to the client.
The actual implementation of JSPs is that they are compiled into servlets before running. This means that programmatically they are equivalent to servlets. However, syntactically they are like HTML or XML documents. Generally, JSPs have been used to generate HTML. However, the JSP 1.0 specification (see [http://java.sun.com/products/jsp/index.html](http://java.sun.com/products/jsp/index.html)) has been extended with an optional XML-compatible syntax to make it easier to create XML documents with JSPs. WebSphere Application Server 3.0 supports this extension.

### 7.1.3 Servlets, JSPs, and JavaBeans

The servlet and JSP programming model also includes the use of JavaBean components.JavaBeans are the component model for Java, and JavaBeans can be used within JSPs. This allows JSPs to easily access Java data. A JSP can create a JavaBean and call methods on it to access properties of the bean. This allows the programming logic to be coded separately from the display logic. Alternatively, a servlet can create and manipulate a bean, and then pass it on to a JSP. This second architecture is more flexible, as it allows a single request to result in one of several display pages, because the servlet can choose which JSP to call.

For more information on the WebSphere programming model, see the ITSO Redbook *WebSphere Application Servers: Standard and Advanced Editions*, SG24-5460.

### 7.2 Generating XML with WebSphere Application Server

The first step in using XML inside WebSphere Application Server is to generate and output XML. Later in this chapter we will see how to present this XML using XSL stylesheets. However, even raw XML is useful, either with an XML-enabled browser, such as Microsoft Internet Explorer 5.0, or in a business-to-business example.

There are three main approaches to generating XML inside WebSphere:

- Using `println` statements and string manipulation to generate and output text strings from servlets.

- Using the Document Object Model (DOM) API to create XML objects in memory, within servlets or JavaBeans, and outputting the results in a servlet or JSP.

- Using XML format JSPs to set elements of the XML document based on Java data.
We will present simple examples of each of these in this chapter, followed by more extensive examples in Chapter 9, “XML application implementation” on page 97.

7.2.1 The println method

This is the simplest method of generating XML output. A servlet can generate XML as easily as it can generate HTML, by simply outputting XML strings. For example, the following servlet code generates simple XML output to a browser:

```java
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;

public class XMLTest extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    {
        response.setContentType("text/xml");
        try {
            PrintWriter out = response.getWriter();
            out.println("<?xml version="1.0" ?>");
            out.println("<RESULTS>");
            out.println("<BOOK ISBN="0198537379">");
            out.println("<BOOKTITLE>The SGML Handbook</BOOKTITLE>");
            out.println("<AUTHOR>Charles Goldfarb</AUTHOR>");
            out.println("</BOOK>");
            out.println("</RESULTS>");
            out.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Here are a few remarks on the example. First, the code must define the content type as "text/xml", otherwise the browser will not be able to interpret it. Second, it is essential that the first output be the 

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

processing instruction. No white space is allowed between the start of the document and this tag.
The result, viewed in Microsoft Internet Explorer 5.0, looks as shown in Figure 9.

![Figure 9. Results of running XMLTest servlet, viewed with Internet Explorer 5.0](image)

Of course, this example only generated static XML, and could have been easily done with a static file. However, the same principals could be used to generate XML from any Java data available to the servlet. Since servlets can access databases, transaction systems, other Web sites, and many legacy applications, this approach could be used to present an XML file over an HTTP interface to many existing applications.

### 7.2.2 The Document Object Model approach

The Document Object Model (DOM) allows Java to manipulate XML documents as in-memory objects. WebSphere Application Server contains the DOM API as part of the XML4J class library, and allows XML documents to be created, manipulated, and output. The following servlet has the same result as the previous servlet, but uses the DOM to create the XML structure.

Some points to note about this example are:

- We have added two imports to the import list: `org.w3c.dom.*` which contains the DOM API; and `com.ibm.xml.parser.*` which contains XML4J support classes which convert the DOM tree to a string.
• The tree is put together in reverse order. This makes sense, because we create the branches and leaves and then put them together. However, this is not necessary. DOM objects are "live", and a new branch can be added to an existing tree without problems.

• WebSphere and XML4J provide the tools to easily print XML from a Document object:

```java
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;
import org.w3c.dom.*;
import com.ibm.xml.parser.*;

public class XMLDOMTest extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    {
        response.setContentType("text/xml");
        try {
            PrintWriter out = response.getWriter();
            String domImplementation = "com.ibm.xml.parser.TXDocument";
            Document d = (Document)Class.forName(domImplementation).newInstance();
            Element results = d.createElement("RESULTS");
            Element book = d.createElement("BOOK");
            book.setAttribute("ISBN","0198537379");
            Element booktitle = d.createElement("BOOKTITLE");
            Text booktitleValue = d.createTextNode("The SGML Handbook");
            Element author = d.createElement("AUTHOR");
            Text authorValue = d.createTextNode("Charles Goldfarb");
            book.appendChild(booktitle);
            book.appendChild(author);
            results.appendChild(book);
            d.appendChild(results);
            ((TXDocument)d).setVersion("1.0");
            new NonRecursivePreorderTreeTraversal(visitor).traverse(d);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```
7.2.3 The JSP method

Java Server Pages can be XML documents, and in this case, creating the XML is even simpler. Some simple directives define the content type as "text/xml", and special tags can be used to insert dynamically generated values. The following JSP 1.0 example generates an XML document with some dynamically generated values containing data about the server process:

```xml
<?xml version="1.0"?>
<jsp:directive.page contentType="text/xml"/>

<PROPERTIES>
  <DATE>
    <%= new java.util.Date() %>
  </DATE>
  <TOTALMEM>
    <%= java.lang.Runtime.getRuntime().totalMemory() %>
  </TOTALMEM>
  <FREEMEM>
    <%= java.lang.Runtime.getRuntime().freeMemory() %>
  </FREEMEM>
  <JAVA>
    <VERSION>
      <%= System.getProperty("java.version") %>
    </VERSION>
    <VENDOR>
      <%= System.getProperty("java.vendor") %>
    </VENDOR>
  </JAVA>
</PROPERTIES>
```

The resulting XML page, viewed in Internet Explorer 5.0, looks as shown in Figure 10.
7.2.4 Comparison of methods — generating XML

The servlet and println method is the most basic method — it generates XML using simple `println` statements, and it is compiled and will run quickly. The benefits of this approach are that it is simple, straightforward, and efficient. It requires a Java programmer who understands the Servlet programming model.

In effect, the JSP model is very similar, except that less Java programming experience is required, and the XML structure of the document is much easier to see. In terms of performance, the JSP should be of similar speed as the servlet, because it will be compiled into a servlet.

Both models rely on the developer to create well-formed XML. The JSP method will be quicker to create and to change if the XML format changes, and the developer is likely to see mistakes in the XML more easily, because the format is visually obvious.

The DOM approach is quite different from the other two. The benefits of the DOM approach are that it allows manipulation of the XML structure, and that the XML4J API takes care of creating well-formed XML. The disadvantage is that it lacks the simplicity of the JSP approach, and for very large documents will be less efficient. The other two approaches will send the document out over the network to the browser as it is created. With the DOM approach, it will be created in memory, and only sent when it is complete.
7.3 Applying XSL to XML with WebSphere Application Server

The Web still runs on HTML, and will do for the foreseeable future. Some users have Internet Explorer version 5.0, which has limited support for XSL stylesheets. However, many users have older browsers that do not support XML. Also, while XML/XSL are new technology, there are going to be differences between releases of XSL and therefore in browser technology. This all means that for today, it makes sense to apply the stylesheet on the server and to send HTML back to the browser. If you do that, then you are guaranteed that the HTML the user views will be the same regardless of browser, and HTML is a well-defined standard.

If you are going to use HTML on the browser anyway, you might ask: “why bother with XML and XSL?”. There are two reasons: (1) XML/XSL will stabilize, and downloading the XML to the browser will become a practical alternative, which will bring greater efficiency to the Web. (2) If you allow the XML to be downloaded as well as the transformed HTML, then your customers can use the raw XML to create new applications with your data.

Once again, there are three approaches to using XSL inside WebSphere:
1. Programmatically, within a servlet
2. Automatically, applied to any XML
3. With XSLT Islands, a new technology available within JSPs

Each of these methods will be described, including some simple examples.

7.3.1 Using XSL within a servlet

WebSphere Application Server provides XSL support through a package called LotusXSL. LotusXSL is IBM’s implementation of the XSLT specification. LotusXSL is installed as a JAR file in the lib\ directory of the WebSphere installation. The latest version of LotusXSL is available at:

http://www.alphaworks.ibm.com/tech/LotusXSL.

LotusXSL can be applied programmatically in a number of ways:
- With an XML document passed: as a URL; as a DOM tree; or as a Java reader or input stream
- With the XSL style specified: within the XML document using a processing instruction; as a URL; as a Java reader or input stream; or as an XML DOM tree
With the output returned: to a Java writer; to an output stream; or as a DOM tree
Almost any combination of the above.

Within a servlet, the method you choose will depend on which method of generating XML that you are using. If you use the DOM tree method to generate XML, then it will be more efficient to pass this directly to the XSL processor, because the processor will not need to parse the incoming XML.

Here are two examples of applying XSL stylesheets to XML within a servlet, one using the println method to generate the XML, and the other using the DOM tree method. Both examples are based on the previous examples of using println and DOM trees, and have been extended to use a XSL stylesheet book.xsl.

The first example takes the XMLTest servlet and extends it to use the XSL stylesheet. The main changes made were these:

- LotusXSL (com.lotus.xsl.*) was imported to add support for the XSL processor.
- The response type will now be HTML, because the XSL stylesheet will create HTML.
- We create a ByteArrayOutputStream to store the output XML, and then create a Reader from the byte array to read the XML into the processor.
- We created an Reader for the XSL stylesheet using the URL.

The code follows:

```java
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;
import com.lotus.xsl.*;

public class XMLTestWithXSL extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    {
        response.setContentType("text/html");
        try {
            PrintWriter out = response.getWriter();
            ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
            PrintWriter xmlWriter = new PrintWriter(byteStream);
            xmlWriter.println("<?xml version="1.0" ?>");
            xmlWriter.println("<RESULTS>");
            xmlWriter.println("<BOOK ISBN="0198537379">");
            xmlWriter.println("<BOOKTITLE>The SGML Handbook</BOOKTITLE>");
            xmlWriter.println("<AUTHOR>Charles Goldfarb</AUTHOR>");
```

```java
Chapter 7. WebSphere and XML approaches   73```
The XML Files: Using XML and XSL with IBM WebSphere 3.0

public class XMLDOMTestWithXSL extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        response.setContentType("text/html");
        try {
            PrintWriter out = response.getWriter();
            // document object type - specific to XML4J
            String domImplementation = "com.ibm.xml.parser.TXDocument";
            // create a document object

            A similar approach can be taken with the DOM tree method of generating XSL. In fact, in some ways this is more straightforward, because there is no need to create the intermediary streams.

            In this case, we modified the XMLDOMTest servlet to call the XSL processor. The changes were small:

            • LotusXSL (com.lotus.xsl.*) was imported to add support for the XSL processor
            • The response type will now be HTML, because the XSL stylesheet will create HTML
            • We called the process method with the DOM tree, the XSL URL, and the output writer.

            import javax.servlet.*;
            import javax.servlet.http.*;
            import java.util.*;
            import java.io.*;
            import org.w3c.dom.*;
            import com.ibm.xml.parser.*;
            import com.lotus.xsl.*;

            public class XMLDOMTestWithXSL extends HttpServlet {
                public void doGet(HttpServletRequest request, HttpServletResponse response) {
                    response.setContentType("text/html");
                    try {
                        PrintWriter out = response.getWriter();
                        // document object type - specific to XML4J
                        String domImplementation = "com.ibm.xml.parser.TXDocument";
                        // create a document object

                        xmlWriter.println("</BOOK>");
                        xmlWriter.println("</RESULTS>");
                        xmlWriter.close();

                        // Create a reader of the byte array.
                        byte[] bytes = byteStream.toByteArray();
                        ByteArrayInputStream is = new ByteArrayInputStream(bytes);
                        Reader xmlReader = new BufferedReader(new InputStreamReader(is));
                        java.net.URL xslURL = new java.net.URL("http://localhost/book.xsl");
                        Reader xslReader = new BufferedReader(new InputStreamReader(xslURL.openStream()));
                        XSLProcessor pr = new XSLProcessor();
                        pr.process(xmlReader, xslReader, null, out);
                        out.close();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
Document d = (Document) Class.forName(domImplementation).newInstance();
    // use the DOM factories from the document object to create parts of the XML document
    Element results = d.createElement("RESULTS");
    Element book = d.createElement("BOOK");
    book.setAttribute("ISBN", "0198537379");
    Element booktitle = d.createElement("BOOKTITLE");
    Text booktitleValue = d.createTextNode("The SGML Handbook");
    Element author = d.createElement("AUTHOR");
    Text authorValue = d.createTextNode("Charles Goldfarb");
    // put the objects together
    booktitle.appendChild(booktitleValue);
    author.appendChild(authorValue);
    book.appendChild(booktitle);
    book.appendChild(author);
    results.appendChild(book);
    d.appendChild(results);

    XSLProcessor pr = new XSLProcessor();
    pr.process(d, "http://localhost/book.xsl", out);
    out.close();
} catch (Exception e)
{
    e.printStackTrace();
}

The XSL stylesheet book.xsl is fairly simple. It creates a table based on the output XML, as shown in Figure 11.
The output from both servlets is the same, and is shown in Figure 12.
7.3.1.1 Comparison of performance

We did some simple benchmarking of the different servlets. The benchmark consisted of accessing the servlets from twenty simulated clients and timing how many connections per second the server produced. The results are not indicative of how real world XML applications will perform, because the XML fragments were small. However, the data does fit with the theory. The results are shown in Figure 13.

![Performance of different approaches](image.png)

Figure 13. Performance comparison — using XML and XSL in a servlet
The results show that using the `println` method is about 7% faster than using the `DOM tree` method. This is because the DOM tree uses a generalized library to generate the output XML, which is less “tuned” than simply outputting the XML directly. However, the tables are turned when the XSL stylesheet is applied — the `DOM tree` method is about 7% faster than the `println` method. This is because the XSL processor does not need to parse the incoming XML. It is also worth noting that the server slowed by a factor of four when it was performing XSL processing.

### 7.3.2 Using XSL automatically

WebSphere can be configured to automatically apply XSL stylesheets to XML. At the time of writing this book, there is a method of doing this that requires a separate download. This functionality is expected to be built into WebSphere in the 3.02 release. Please read the release notes for information. We used a servlet called `DefaultApplyXSL`, which is available from the LotusXSL download site, either as part of LotusXSL or as a separate download file. `DefaultApplyXSL` has three modes of operation:

1. It can be called with request parameters which specify the XML input and output: for example — `http://server/servlet/DefaultApplyXSL?URL=book.xml`. This is probably the least useful mode, because it “hard codes” the use of this servlet.

2. It can be called as an `alias`, which means that it will automatically present static XML files stored in the Web server’s path using an XSL stylesheet. For example, if `http://server/book.xml` is the URL of a static XML file, then you can configure `http://server/xmlapp/book.xml` to be the same file with the XSL stylesheet applied.

3. It can be called as a `mime-filter`, which means that it will intercept any dynamically generated output of a given mime-type, and apply the relevant XSL stylesheet. For example, you can configure all `text/xml` output to have an XSL stylesheet applied to it.

To use aliases and mime-filters, you need to define a `processing instruction` (PI) which associates a stylesheet with the XML. This is defined by the W3C Recommendation “Associating Style Sheets with XML documents” (see [http://www.w3.org/TR/xml-stylesheet/](http://www.w3.org/TR/xml-stylesheet/)). For example the following PI would associate the sample XSL stylesheet with the XML document:

```xml
<?xml-stylesheet type="text/xsl" href="sample.xsl" ?>
```
Using aliases and mime-filters, you can develop a completely XML-based application, using static XML files, XML based JSPs, and servlets which generate XML. These can be used together with multiple XSL stylesheets to create a front-end. Other front-ends can be built for different client types, for example, Wireless Markup Language for mobile phone browsers. In our sample application (see Chapter 8, “XML application scenario” on page 89), we demonstrate such an application.

For details of how to configure DefaultApplyXSL as an alias and a mime-filter, please see Appendix B, “ITSO XML Application demo installation” on page 207.

7.3.3 XSLT Islands

*XSLT Islands* are a new technology available inside WebSphere Application Server version 3.0, which allows XSL to be used directly inside JSPs. In effect, this technology allows JSPs to contain HTML, Java and XSL in one file. The JSP can give the XSL an XML source which it applies the XSL to, and the results are incorporated inside the resulting output. This is a very powerful functionality, and it allows the XSL stylesheet to directly access Java data as well as XML data.

XSLT Islands are based on an IBM technology called *Bean Scripting Framework* (BSF). BSF is an IBM project which is available free-of-charge with source for commercial use. The BSF team have publicly stated that they intend to submit BSF as a Java standard. BSF allows scripting languages such as NetRexx, Java Script, and Python to be used within a Java framework. IBM has incorporated BSF into WebSphere Application Server JSPs, and also included XSL as a scripting language for BSF. This means that JSP pages can include JSP expressions written in XSL.

To enable XSLT Islands inside WebSphere Application Server, you need to download a support file from the BSF homepage, which is http://www.alphaworks.ibm.com/tech/bsf. Details on how to configure WebSphere for XSLT Islands are in Appendix B, “ITSO XML Application demo installation” on page 207.
An example of using XSLT Islands is presented in Figure 14 on page 81.

The main points of this are:

1. The initial scriptlet `<jsp:scriptlet ...>` is there because of a bug in WebSphere 3.0.

2. The next code `<% String src = ..... %>` defines the XML — in this case it is hard coded, but it could be based on Java or referenced from a URL.

3. The whole XSL document is referenced between the tags `<jsp:expr language="lotusxsl">` and `</jsp:expr>`.

4. By defining the `java:` namespace we can directly reference Java data from the XSL stylesheet. The tag `<xsl:value-of select="java:java.util.Date.new()"/>` gets the current date and time from the Java Virtual Machine.

The result of this JSP is displayed in Figure 15 on page 82.
Figure 14. JSP with XSL Island
7.4 Using XSL selectively

There are two reasons not to use XSL on the server-side. First, as browsers become more XML/XSL aware, you might wish to disable server-side applications of XSL for performance reasons. Tests show that using XSL on the server-side can slow down performance by a factor of 4. Second, the great benefits of XML — re-usability of data, are lost to the users when the user cannot access the original XML data.

There are two approaches to selectively applying XSL stylesheets to XML with WebSphere.

7.4.1 Defining an alternative Web application

WebSphere Application Server defines a concept of Web applications. Each Web application is defined by a unique URI prefix, such as /xmlapp.

Each Web application has a defined document directory for JSPs, and classpath for servlets. For the ITSO XML application, we define the xmlapp Web application, and define the DefaultApplyXSL servlet as a mime-filter and alias. For details see Appendix B, “ITSO XML Application demo installation” on page 207. One of the clever features of WebSphere is that another Web application can point to the same JSPs and servlets, and yet have different application definitions. For example, if we define a new Web application using the URI /xmlapp_noxsl, but do not define DefaultApplyXSL for this
application, then the XML is directly accessible by the browser. The URL `/xmlapp/orderbasket.jsp` will return a formatted HTML response, while the `/xmlapp_noxsl/orderbasket.jsp` will return the unformatted XML.

This can be used in two ways. First, users with XML/XSL browsers could be pointed at the non-XSL version, either with a link or automatically using a servlet or some Java Script. Second, there could be pointers to the non-formatted XML output for saving and importing into other applications.

### 7.4.2 Applying XSL based on browser type

The W3C recommendation on associating stylesheets (see Part 7.3.2, “Using XSL automatically” on page 78) defines the ability to use separate stylesheets for different browsers. This is done through a media attribute in the PI. For example, the following two PIs together would define one stylesheet as the default, and an alternate stylesheet to use for Microsoft Internet Explorer:

```xml
<? xml-stylesheet type="text/xsl" href="default.xsl" ?>
<? xml-stylesheet type="text/xsl" href="special.xsl" media="explorer" alternate="yes" ?>
```

WebSphere Application Server and DefaultApplyXSL handle this by defining a mapping between the attributes in the media attribute, and the strings which the browsers send to identify themselves to the server. The mapping is defined in the `media.properties` file in the `properties` directory of the WebSphere installation. Using these PIs, you could, for example, define an XML to HTML stylesheet as a default, and an XML to Wireless Markup Language stylesheet to use with browsers which identify themselves as mobile phones.

---

**Note**

It is tempting to try to define an empty stylesheet that doesn’t modify the XML to use with an XSL-enabled browser such as Microsoft Internet Explorer v5.0. However, if you try to do this with a text/xml mime-filter, you will end up with an infinite loop, because each time it is run, it will output text/xml, firing up the filter once again.

You could solve this by defining a new mime-type, for example text/xml-auto. The servlets and JSPs would output text/xml-auto, which would be setup as the mime-type for filtering.
7.5 Comparison of approaches

We have already compared the three ways of generating XML earlier in the chapter. In this section we compare the three ways of using XSL. We also offer a brief comparison of using XML and XSL with using pure Java Servlets and JSPs without XML.

7.5.1 Applying XSL

Applying XSL programmatically is very powerful. XSLT is a general purpose transformation language, and it can be used in many ways, including doing general purpose XML to XML transformations. When you call the XSL processor in a servlet, you can use any input and output that you wish, making this the most flexible way of using XSL. It is also probably the fastest way of generating HTML from XML.

Using the WebSphere facility to automatically apply XSL to XML based on the <?xml-stylesheet?> tags has two main benefits.

First, it uses code based purely on the standards XML, XSLT, and how to apply XSL to XML. This means that applications generated using these technologies will support future browsers that support these standards, as well as working with vendor-independent technology.

Second, this approach automatically supports multiple channels through the use of browser-specific XSL stylesheets. This is the only one of these approaches which does this. Unfortunately this approach is not fast.

XSLT Islands offer a very powerful approach to using XSL. At the expense of extending your files with some extra tags, you get the ability to mix Java data and XML data.

For example, you may wish to enhance the presentation of XML based on data stored in a user’s session object. To do this with the other two approaches would require you to extend the XML format to include the extra data. In many applications this would not be acceptable, because the XML format is carefully defined, possibly by an external body. Using the XSL Islands, the session data is directly accessible in the XSL stylesheet.

Another benefit is that the JSP and XSL, which together create the output, can be physically stored in the same file, making it easier to manage.

The XSLT Islands technology performs better than aliasing and filtering, but not as well as the programmatic approach.
7.5.2 Comparison of XML/XSL and Servlet/JSP

At a very high level, XML/XSL and Servlet/JSP technologies both offer separation of data and presentation. However, the two technologies are not mutually exclusive — as we have shown, they can be used together to create powerful approaches to create applications.

The selection of technologies is really based on a number of factors, including corporate strategy, tactical requirements, and skills. Without going into those, we present a short list of the advantages of each technology.

7.5.2.1 Advantages of Servlets/JSPs

Here are the advantages of using Servlets/JSPs:

- Servlets/JSPs are fast. They are compiled technology.
- JSPs are the standard way of scripting application built to the IBM e-business Application Framework, and as such they fit into a well developed programming model.
- JSPs have excellent design tools. Most HTML tools can be used to edit JSPs. IBM WebSphere Studio includes the WebSphere Page Designer which is a fully graphical editor for JSPs as well as HTML.
- JSPs are much simpler to develop and debug than XSL scripts.

7.5.2.2 Advantages of XML/XSL

Here are the advantages of using XML/XSL:

- XSL can sort and generate complex data.
- XSL can be used on some browsers allowing off loading of server work.
- XSL has uses outside of presentation, and may become a very widespread technology.
- XSL is optimized to generate multiple XML output types — including HTML, WML, SMIL, and SVG.
- XSL is very powerful, at the cost of being harder to develop than JSPs.
Chapter 8. XML application scenario

In this chapter, we define a sample application that uses XML in various fashions, thereby allowing us to demonstrate how to use XML in conjunction with servlets, Java Server Pages (JSPs) and WebSphere in general.

The design approach outlined in the following sections is not as thorough as it would be in real life. However, it suits our purpose, which is to provide a reasonable understanding of the application’s requirements and functionality.

8.1 Application overview

The sample application that was implemented for this book is a simplified variation on the shopping-cart approach most common on the Web nowadays. Navigating through the application with a Web Browser, users can logon or register, perform a search on a product catalogue, place purchase orders to suppliers and track their current and past orders. From a lower level perspective, we can outline the list of components which provide the aforementioned functionality:

- A main application menu, which lists the options available to the user.
- A logon screen where the user authenticates himself/herself in order to get access to the rest of the system’s functionality.
- A registration process where new users enter their details before they can access the system using the above logon screen.
- A search mechanism which provides shoppers with a fast means to find items of interest, based on a keyword description.
- A shopping basket which lists the items currently selected for purchase by the user, and also provides editing functionality for the user to modify their choices.
- An ordering mechanism which creates purchase orders based on the user’s selection and sends them to the appropriate supplier.
- An order completion mechanism which simulates supplier input and automatically updates pending orders.

In addition to this list, refer to Figure 16 for a visual description of how these components fit inside the overall application workflow.
Figure 16. XML application workflow model
8.2 Database architecture

Persistent data from our sample application is maintained in database tables. In keeping with our intent to keep things simple, the tables were designed with a basic approach in mind, with columns clearly labeled and constraints kept to a minimum. One may notice that, in some places, the use of constraints and triggers might have been justified. However, in cases where implementing these features might have complicated the issue and distracted us from our initial purpose, we took a conscious decision to do away with them.

User information was not stored directly in our application database. Since WebSphere provides functionality to store user profiles in a pre-designed PROFILES database, we decided to make use of this feature to simplify the implementation of an authentication mechanism and storage of user-related data. We look at the implementation of user profile management in greater detail in 9.1, “Logon and user profile management” on page 97.

Other types of data are stored in the following tables in a separate custom-designed database: SUPPLIERS, PRODUCTS, ORDER_HEADERS and ORDER_ITEMS. For more information regarding the table design, refer to the Data Definition Language (DDL) documents listed in Appendix C, “Data Definition Languages (DDLS)” on page 217. The model depicted in Figure 17 shows the relationships between the tables in our database, along with the tables’ primary keys.
8.3 Document Type Definition

Since our application will generate and use XML documents, we need to define a Document Type Definition (DTD). This document has two purposes:

- To serve as a framework and provide a set of elements and attributes which will be used in the XML “templates” used by the application to generate XML documents from raw data retrieved from the back-end.

- To validate incoming XML documents sent by the supplier in reply to outgoing purchase orders generated by the customer application.

To simplify the mapping process, we have provided element definitions which match as closely as possible to the existing tables in the database. Similarly, relationships which were explicitly implemented in the database design are
declared explicitly in the DTD, as in the following example, where the <ORDER_ITEM> element mirrors the ORDER_ITEMS table by including in its definition a <PRODUCT> element, which in turns, contains a <SUPPLIER> element:

```xml
<!ELEMENT Product (
    Supplier,
    Product_Name,
    Description,
    Price,
    Minimum_Quantity,
    Stock_Quantity)
...
```
```xml
<!ELEMENT Order_Item (
    Product,
    Order_Quantity >
...
```

Several approaches could have been used in writing a DTD for this application. The main motivation behind our approach was to illustrate a varied array of DTD concepts, such as attributes and entities. The following example shows how we made use of parameter entities and attributes:

```xml
<!ENTITY % User_Attributes
    "Username ID #REQUIRED
    Password CDATA #REQUIRED">
...
```
```xml
<!ENTITY % Item_Attributes
    "Product_ID ID #REQUIRED
    Supplier_Part_No CDATA #IMPLIED">
```

Consequently, the following DTD sets out to use lots of DTD design elements, such as attributes and entities, and although it suits our purpose, it may not represent the best possible approach to implement XML support for this type of application:

```xml
<!-- Entities List & Attributes Definitions -->
<!-- XML & WebSphere Document Type Definition Subset -->
<!-- Author: Christophe Chuvan -->
<!-- Last Modified Date: 05/10/99 -->
<!ENTITY % User_Attributes
    "Username ID #REQUIRED
    Password CDATA #REQUIRED">

<!ENTITY % Item_Attributes
    "Product_ID ID #REQUIRED
    Supplier_Part_No CDATA #IMPLIED">
```
<!ELEMENT First_name (#PCDATA)>
<!ELEMENT Last_name (#PCDATA)>
<!ELEMENT Customer_Name (First_name, Last_name)>
<!ELEMENT Supplier_name (#PCDATA)>
<!ELEMENT Product_name (#PCDATA)>
<!ELEMENT Phone (#PCDATA)>
<!ELEMENT Email (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Country (#PCDATA)>
<!ELEMENT Zip (#PCDATA)>
<!ELEMENT Address (Phone, Email, Street, City, State, Country, Zip)>
<!ELEMENT B2B_URL (#PCDATA)>
<!ELEMENT XSL_URL (#PCDATA)>
<!ELEMENT Customer (Customer_Name, Address?)>
<!ATTLIST Customer %User_Attributes;>
<!ELEMENT Supplier (Supplier_Name, Address, B2B_URL, XSL_URL)>
<!ATTLIST Supplier %User_Attributes;>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ELEMENT Minimum_Quantity (#PCDATA)>
<!ELEMENT Stock_Quantity (#PCDATA)>
<!ELEMENT Order_Quantity (#PCDATA)>
<!ELEMENT Product ( Supplier, 
Product_Name, 
Description, 
Price, 
Minimum_Quantity, 
Stock_Quantity)> 

<!ATTLIST Product %Item_Attributes;> 

<!ELEMENT Order_Item ( Product, 
Order_Quantity )> 

<!ATTLIST Order_Item 
STATUS (New|Unsubmitted|Pending|Shipped|Completed) "New" 
SHIP_DATE CDATA #IMPLIED 
ITEM_NO CDATA #REQUIRED> 

<!ELEMENT Order_Item_List (Order_Item+)> 

<!ELEMENT Order (Order_Item_List)> 

<!ATTLIST Order 
Order_No ID #REQUIRED 
Customer_Order_No CDATA #IMPLIED 
ORDER_DATE CDATA #REQUIRED> 

<!-- NB: Dates should be in ISO format (yyyy-mm-dd) --> 
<!-- --> 
<!-- --> 
<!-- -->
8.4 Application architecture

This section provides a quick reference to each module in the sample application in Table 6. A description of each module is provided along with a page reference to the corresponding section in the next chapter, which looks in detail at the design and implementation of the module.

Table 6. Application modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logon Sub-System</td>
<td>This section describes the implementation of the authentication mechanism and how user-related information is stored.</td>
<td>Section 9.1, on page 97</td>
</tr>
<tr>
<td>Search Engine</td>
<td>This section looks at the search mechanism that allows users to retrieve products from the application database based on keyword searches.</td>
<td>Section 9.3, on page 114</td>
</tr>
<tr>
<td>Shopping Basket</td>
<td>The implementation of the shopping basket and ordering process is discussed in detail in this section.</td>
<td>Section 9.4, on page 118</td>
</tr>
<tr>
<td>Order Tracking</td>
<td>This module looks at how the user can track the status of his/her current orders and generate reports.</td>
<td>Section 9.5, on page 133</td>
</tr>
<tr>
<td>Supplier Updates</td>
<td>The mechanism to allow suppliers to update the status of pending orders is discussed here.</td>
<td>Section 9.6, on page 144</td>
</tr>
</tbody>
</table>
Chapter 9. XML application implementation

9.1 Logon and user profile management

Our Web application needs to register user information in a safe place. We decided to use the user profile management services of WebSphere to store the necessary data in a relational database. The database also maps nicely to the customer DTD we use in our application scenario.

9.1.1 User profile database

WebSphere implements an object oriented way of accessing user profile data. The following fields are included in the implementation class (com.ibm.servlet.personalization.userprofile.UserProfile) and the corresponding database table: (USERPROFILE):

- Address(1)
- Address(2)
- First Name
- Surname
- Day phone number
- Night phone number
- City
- Nation
- Employer
- Fax number
- Language
- Email address
- State/Province
- Postal code

Because we need our users to logon to our Web site, a password field is also needed.

9.1.1.1 Creating the database

WebSphere Application Server creates a database table for holding the user profile information when user profile services are enabled for the first time. The database to be used must exist before activating the services. In DB2, you can create the database by opening the Command Line Processor, and entering the following command:

```bash
create db profiles
```
After the database has been created, you can define the newly created database as a datasource for WebSphere Application Server. On the Types page of WebSphere Administration Console, select Datasources->create. In the dialog window, specify the name of the database to be used (see Figure 18).

![Create a DataSource](image)

**Figure 18. Datasource for user profiles**

After having defined the database, you need to enable the user profile services. In the WebSphere Administration Console, select the Topology page, expand your host, Default Server, and Servlet Engine, to modify the settings for user profiles. On the Enable page of the User Profile Management settings, set the Enable User Profile radio button state to true. The database table for user profiles is created in the selected database when the first application tries to use the user profile services. WebSphere Application Server samples contain a sample that you can use for this purpose.

### 9.1.1.2 Modifying the user profile database

The extra field for the password is created automatically in the database, if you extend the existing UserProfile class with your own extensions. The following listing shows the complete listing of PwUserProfile class that implements the necessary functionality to store the password:

```java
package itso.xmlapp.util;

public class PwUserProfile extends 
    com.ibm.servlet.personalization.userprofile.UserProfile 
  implements com.ibm.websphere.userprofile.UserProfileExtender {
  public String passwd;
  //** PwUserProfile constructor.
```
public PwUserProfile() {
    super();
}

/** Return the new columns of User Profile table. */
public java.lang.String[] getNewColumns() {
    String newCol[] = {"passwd"};
    return newCol;
}

/** Get the value of passwd field. */
public String getPasswd() {
    return (String) getByType("passwd");
}

/** Set the passwd field value. */
public void setPasswd(String aPw) {
    passwd = aPw;
    setByType("passwd", aPw);
}

The extended PwUserProfile class has to be introduced to User Profile Management as the new Data Wrapper class, as shown in Figure 19.

![Extended user profile](image)

**Figure 19. Extended user profile**

The Data Wrapper class can potentially be used by all WebSphere applications, and therefore the class must be put in a place where WebSphere can find it. The Java class file can be placed in the classes directory under WebSphere, or the Java archive (jar) file can be included in WebSphere Application Server runtime. The latter can be achieved by
specifying the -classpath flag in the Command line arguments field of the Default Server (see Figure 20).

Figure 20. Command line arguments for WebSphere

9.1.2 User profile registration servlet

When registering to the Web site, users enter the necessary data in a form, and the data is sent to a servlet. The servlet extracts the data from the request and checks for a duplicate username. If a username already exists, a response is sent to the user telling that the chosen userid is already in use. Otherwise, a new user profile is created, and an HttpSession is established, with username in the session data. The following listing shows how a servlet establishes a user profile:

```java
UserProfileManager manager = UserProfileManager.getUserProfileManager();
com.ibm.websphere.userprofile.UserProfile userProfile = null;
String username = (String) request.getParameter("id");
String passwd = (String) request.getParameter("pw");
....
try { // try to create the user profile
    userProfile = manager.addUserProfile(username);
    userProfile.setFirstName((String) request.getParameter("firstname"));
    userProfile.setSurName((String) request.getParameter("lastname"));
    ....
    ((PwUserProfile) userProfile).setPasswd(passwd);
    javax.servlet.http.HttpSession session = request.getSession(true);
    // put the user name in the current session
    session.putValue("username", username);
}
```

100 The XML Files: Using XML and XSL with IBM WebSphere 3.0
catch (UserProfileCreateException e) {
    // if user profile for the username already exists
...}

9.1.3 Logon servlet

Whenever a user tries to access a servlet resource, the servlets checks for an existing session. If a session does not exist, the user is redirected to the logon page. Logon information is sent to a servlet that checks if the userid/password combination exists in the user profile database. The logon servlet uses the user profile management services to retrieve the user information from the database and to check the supplied credentials. The following listing shows how this can be achieved:

```java
UserProfileManager manager = UserProfileManager.getUserProfileManager();
com.ibm.websphere.userprofile.UserProfile userProfile = null;
username = (String)request.getParameter("id");
pwd = (String)request.getParameter("pw");
if ((pwd != null) && (username != null)) {
    try {
        // try to retrieve the user profile
        userProfile = manager.getUserProfile(username);
        if (pwd.equals(((PwUserProfile) userProfile).getPasswd())) {
            javax.servlet.http.HttpSession session = request.getSession(true);
            // establish a session
            // put the user name in the current session
            session.putValue("username", username);
            page = "LogonOk.xml";
        }
    } catch (UserProfileFinderException e) {
        // try to find the existing user profile in read-only mode
        page = "LogonError.xml";
    }
    // Call the output page. If the output page is not passed
response.sendRedirect(pagePrefix + page);
```

9.1.4 User interfaces

All user interface elements for both logon and registration can be implemented using static XML documents and XSL stylesheets. The functionality that associates the stylesheet with XML and produces the output for the user interface can be implemented by using the DefaultApplyXSL servlet, JSP Islands, or writing a servlet of your own. The simplest way of
doing this is to use the DefaultApplyXSL servlet, which can be configured to be activated whenever an XML document is requested from the server.

9.1.4.1 DTD for static user interfaces

Because user interfaces consist of a common set of elements, such as headings, labels, text areas, and controls (for example, buttons and text fields), we decided to create a simple DTD (shown in Figure 21) describing these elements.

```
<!ELEMENT SCREEN (HEADER, TITLE, PARAGRAPH*, FORM?)>
<!ELEMENT HEADER (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT TEXT (#PCDATA)>
<!ELEMENT PROMPT (#PCDATA)>
<!ELEMENT PHEADER (#PCDATA)>
<!ELEMENT PARAGRAPH (PHEADER?, TEXT+, HREF*)>
<!ELEMENT INPUT (#PCDATA)>
<!ATTLIST INPUT
  TYPE CDATA #REQUIRED
  ID CDATA #REQUIRED
  NAME CDATA #REQUIRED
  MAXLENGTH CDATA #IMPLIED
  SIZE CDATA #IMPLIED>
<!ELEMENT HREF (#PCDATA)>
<!ATTLIST HREF
  TARGET CDATA #IMPLIED>
<!ELEMENT TABLE (CONTROL*, TEXT*)>
<!ATTLIST TABLE
  ALIGN (LEFT|RIGHT|CENTER) "CENTER"
  WIDTH CDATA #IMPLIED
  BORDER CDATA #IMPLIED>
<!ELEMENT FORM (TABLE)>
<!ATTLIST FORM
  METHOD (GET|POST) "POST"
  ACTION CDATA #REQUIRED>
<!ELEMENT CONTROL (PROMPT?, INPUT+)>
```

Figure 21. DTD for user interfaces
The DTD in Figure 21 is far from being a complete, all-purpose DTD for user interfaces. It is just an example of how XML can be used to describe a structure of them, and it fulfills the needs of our little XML application well enough.

9.1.4.2 XSL for static user interfaces
The XSL stylesheet is needed to convert the XML document to a form appropriate for the client type that does not natively support XML. In our XML application, we use only one stylesheet for each client type. This approach has the advantage of concentrating all conversions in one place. For example, if we wish to include a background image in every HTML user interface, that can be defined in the stylesheet. After all, that is exactly the reason for using stylesheets: they control the representation, not the actual data, which is the responsibility of the XML document. The stylesheet implementation is shown in Figure 22.

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

  <xsl:template match="SCREEN">
    <HTML>
      <BODY background="/xsl/world.jpg">
        <CENTER>
          <IMG ALIGN="Middle" ALT="XMLapp" BORDER="0" HSPACE="0"
           SRC="/xsl/xmlapp.gif" VSPACE="0"/>
          <xsl:apply-templates/>
        </CENTER>
      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="HEADER">
    <HEAD>
      <TITLE><xsl:apply-templates/></TITLE>
      <META http-equiv="Content-Type" content="text/html"
           charset="iso-8859-1"/>
      <META http-equiv="Expires" content="0"/>
    </HEAD>
  </xsl:template>

  <xsl:template match="TITLE">
    <H1><xsl:apply-templates/></H1>
  </xsl:template>
</xsl:stylesheet>
```
Figure 22. XSL LogonForm stylesheet
9.1.4.3 Logon user interface

The logon user interface prompts the user for a username and a password, which are sent to the LogonServlet. If the logon is successful, the LogonOk.xml file is presented to the user. In case the logon fails, LogonError.xml is shown. The whole XML file describing the user interface for logon is presented in Figure 23.

```xml
<?xml version="1.0"?>
<!DOCTYPE LogonScreen SYSTEM "http://localhost/dtd/xmlgui.dtd">
<?xml-stylesheet type="text/xsl" href="http://localhost/xsl/LogonForm.xsl"?>
<!--DOCTYPE LogonScreen SYSTEM ../../../dtd/xmlgui.dtd-->
<SCREEN>
  <HEADER>User Logon</HEADER>
  <TITLE>Logon required</TITLE>
  <PARAGRAPH>
    <PHEADER>Please fill in the form</PHEADER>
    <TEXT>The userid and password are needed before you can do business with us</TEXT>
  </PARAGRAPH>
  <FORM METHOD="post" ACTION="/xmlapp/servlet/itso.xmlapp.servlets.LogonServlet">
    <TABLE BORDER="2" WIDTH="50%" ALIGN="CENTER">
      <CONTROL>
        <PROMPT>User name</PROMPT>
        <INPUT TYPE="TEXT" NAME="id" ID="id" MAXLENGTH="20" SIZE="20"/>
      </CONTROL>
      <CONTROL>
        <PROMPT>Password</PROMPT>
        <INPUT TYPE="PASSWORD" NAME="pw" ID="pw" MAXLENGTH="20" SIZE="20"/>
      </CONTROL>
      <CONTROL>
        <INPUT TYPE="SUBMIT" NAME="Logon" ID="Logon">Logon</INPUT>
        <INPUT TYPE="RESET" NAME="Reset" ID="Reset">Reset</INPUT>
      </CONTROL>
    </TABLE>
  </FORM>
</SCREEN>
```

Figure 23. Logon user interface XML document

When the XSL stylesheet is applied to the XML document shown in Figure 23, the output is an HTML document, the visual representation of which is shown in Figure 24.
LogonOk.xml is a simple user interface, just informing the user that logon was successful. The user interface includes a link that can be used to request the search form, which is the main dynamic user interface for the ITSO XML application. The following listing shows how LogonOk.xml is implemented:

```xml
<SCREEN>
  <HEADER>Logon</HEADER>
  <TITLE>Logon Successfull</TITLE>
  <PARAGRAPH>
    <PHEADER>We are pleased to do business with You!</PHEADER>
    <![CDATA[Please follow the link below to continue]]>
    <HREF TARGET="/xmlapp/xml/SearchStart.xml">*** Search our product catalog ***</HREF>
  </PARAGRAPH>
</SCREEN>
```

The above document structure is also used for other informative static user interfaces (LogonError.xml, RegisterError.xml, RegisterOk.xml and
RegisterInvalid.xml). With the stylesheet in Figure 22 applied, the LogonOk user interface looks like the one Figure 25.

Figure 25. LogonOk screen

The LogonError screen informs the user that the supplier credentials did not match with any entries in the user profiles database. The screen is shown in Figure 26.

Figure 26. LogonError screen
9.1.4.4 Registration user interface
The registration user interface (RegisterForm.xml) is very much like the logon interface, only more fields are presented to the user to be filled in. The user interface is shown in Figure 27.

RegisterServlet processes the data and sends the response to the user. If not enough data was sent to the servlet, RegisterInvalid.xml is shown, and if other errors occur (such as a duplicate username), RegisterError.xml is shown. RegisterOk.xml is shown if registration was successful. Their visual representation is not shown here, since they look very much like the LogonOk end LogonError screens.

The complete listings of all XML documents that describe the user interfaces, the LogonForm.xsl stylesheet, and the two servlets can be found in the ITSO redbook Web site:

ftp://www.redbooks.ibm.com/SG245479/
9.2 Alternative — XSLT Islands

As an alternative to DefaultApplyServlet, applying stylesheets to XML documents can be done using the XSLT Islands technology. When using XSLT Islands, the mime-filter for text/XML mime type does not need to be configured. Instead, a JSP containing the XSL stylesheet is addressed by an HTTP request, and the JSP loads and processes the XML document. Provided that all HTML tags are generated by XSLT processing, the JSP implementation can be written as follows:

```jsp
<jsp:scriptlet language="javascript"></jsp:scriptlet>
<jsp:directive.page import="java.io.*, java.net.*" />
<%
    String url = null;
    Object o = request.getParameter("xmlURL");
<%   String url = null;
    Object o = request.getParameter("xmlURL");
    if (o == null) throw new Exception("xmlURL parameter missing");
else {
    url = (String)o;
    URL u = new URL("http://localhost" + url);
    URLConnection uc = u.openConnection();
    uc.connect();
    InputStream is = uc.getInputStream();
    Reader r = new InputStreamReader(is);
    bsf.registerBean("lotusxsl:src", r);

    <jsp:expr language="lotusxsl">
        <xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
                         xmlns:java="http://xsl.lotus.com/java">
            <!-- INSERT YOUR STYLESHEET TEMPLATES BELOW THIS LINE-->
            <xsl:template match="SCREEN">
                <HTML>
                    <BODY background="/xsl/world.jpg">
                        <xsl:apply-templates/>
                    </BODY>
                </HTML>
            </xsl:template>
            .............
            <!-- INSERT YOUR STYLESHEET TEMPLATES ABOVE THIS LINE-->
        </xsl:stylesheet>
    </jsp:expr>
```

Figure 28. XSLT Island template
In the JSP in Figure 28, the XML document to be processed is passed to the JSP as a request parameter xmlURL. The URL reference is converted into an input stream and then registered to the Bean Scripting Framework. After that, the LotusXSL JSP extensions invoke the LotusXSL processor. The stylesheet templates that we implemented in the static XSL files can be copied as-is between the commented lines in the JSP file. In Figure 28, only the first template is shown.

Using the XSLT Islands approach, the HTTP request for XSL/XML processing takes the form:

```
http://localhost/xmlapp/XSL2JSP.jsp?xmlURL=/xml/LogonForm.xml
```

The example above applies the stylesheet defined inside the XSL2JSP.jsp to the static XML file LogonForm.xml, the URL of which is /xml/LogonForm.xml. Because for every static XML document in the ITSO XML application, we use a common XSL file, the same JSP file that implements the stylesheet using XSLT Islands can also be used to process the static XML documents.

It is important that you do not refer to the static XML file using a path that includes the Web application (/xmlapp/xml/LogonForm.xml), if you have set up the DefaultApplyXSL servlet to handle the XML content type, because the XML file would be processed by the servlet before invoking the JSP. The results would not be the ones you are after.

We implemented the functionality needed to change the user registration information using the XSLT Islands technology. The XML document is created by a JSP called RegisterChange.jsp that accesses the userprofile database using the value of the user parameter of the HTTP request. This JSP is shown in Figure 29.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE LogonScreen SYSTEM "http://localhost/dtd/xmlgui.dtd">
<jsp:directive.page contentType="text/xml" session="true"
    errorPage="showError.jsp" />

<% String user = ((String) request.getParameter("user")); %>

<tsx:dbconnect id="profiles" userid="xmlguru" passwd="xmlguru"
    url="jdbc:db2:profiles" driver="COM.ibm.db2.jdbc.app.DB2Driver">
</tsx:dbconnect>

<tsx:dbquery id="logon" connection="profiles">
    select * from xmlguru.userprofile where username='<%=user%>'
</tsx:dbquery>
</SCREEN>
```
Figure 29. RegisterChange.JSP

The XSLT Islands implementation differs slightly from that shown in Figure 28 on page 109, because we wanted to take the username from the session object and pass that as a parameter to the JSP that does the database query. The portion of the Java code that shows the differences is given below:

```java
<% HttpSession hs = request.getSession(true); String user = ((String) hs.getValue("username")); if (user == null) throw new Exception("Sorry, could not get username from session, please logon again"); else {
    String url = null;
    Object o = request.getParameter("xmlURL");
}
```
if (o == null) throw new Exception("xmlURL parameter missing");
}

try {
    manager.remove(username); // Remove old values
    // try to add a new user profile
    userProfile = manager.addUserProfile(username);
    userProfile.setFirstName(((String) request.getParameter("firstname")).trim());
    userProfile.setSurname(((String) request.getParameter("lastname")).trim());
    userProfile.setAddress1(((String) request.getParameter("address")).trim());
    userProfile.setDayPhone(((String) request.getParameter("phone")).trim());
    userProfile.setCity(((String) request.getParameter("city")).trim());
    userProfile.setStateOrProvince(((String) request.getParameter("state")).trim());
    userProfile.setPostalCode(((String) request.getParameter("zipcode")).trim());
    userProfile.setEmail(((String) request.getParameter("email")).trim());
    ((PwUserProfile) userProfile).setPasswd(passwd);
    page = "RegisterChangeOk.xml";
} catch (UserProfileCreateException e) {
    page = "RegisterChangeError.xml";
}

User profile information is inserted in a dynamically created XML document, which looks very much like the registration servlet (see Figure 30), except that the fields are pre-filled with existing values. The data is sent to a servlet (RegisterChangeServlet) that uses the UserProfile classes of WebSphere to update the information in the database. The essential part of the performTask method body is shown below.
The HTTP request for accessing the registration update screen takes the form:

http://localhost/xmlapp/XSL2JSP.jsp?xmlURL=/xmlapp/RegisterChange.jsp

As you can see, the XSLT Islands implementation processes the request, retrieves the XML document passed as a parameter, and applies the stylesheet. The end result is shown in Figure 30.

Figure 30. Changing registration information

Note that XSLT Islands implementation does not work for dynamically created XML pages, if the Web application has the DefaultApplyXSL servlet configured as a mime-filter, because the dynamic XML document would be interpreted by the servlet before it would be parsed by the JSP.
9.3 Search engine

First, the user needs to register to our Web application. Then, the user can order products. To select the products from a wide variety, the user needs an efficient search engine. The search engine should allow the user to browse through the range of products of his/her interest, check the price, check a brief description, and order the item.

9.3.1 Searching for products and adding to order

Upon selecting to search for products, the user is asked for an input string (keyword) that will be used to concentrate on that user's interests. This user input is requested through a form that is based on a static XML document displayed using XSL. The user interface contains a text field to receive the input string from the user. This input string is received by a JSP, which is the heart of the search engine. The JSP checks for the session to avoid searches by users who are not registered. If the search is invoked by an unregistered user, then the JSP redirects the user to the logon form. Otherwise, the JSP establishes the database connection. The search string is sought in the database, looking for a match within product id, product name, and product description. To be exact, the search string is matched to the values of fields (PRODUCT_ID, PRODUCT_NAME, and DESCRIPTION) of the table "products". The result of the search is an XML file. The JSP file for the search is as follows:

```xml
<?xml version="1.0"?>
<?xml-stylesheet href="http://localhost/xsl/result.xsl" type="text/xsl"?>

<!-- FileName: search.jsp -->
<!-- Author: Ramani Ranjan Routray -->
<!-- Created 30th Sept. 99 -->
<!-- Last Modified XXXX -->
<jsp:directive.page contentType="text/xml" />

<tsx:dbconnect id="a118" userid="xmlguru" passwd="xmlguru" url="jdbc:db2:A118" driver="COM.ibm.db2.jdbc.app.DB2Driver">
</tsx:dbconnect>

<% String search = request.getParameter("searchstr");%>
<% search = search.trim();%>
<% search = search.toUpperCase();%>
<% HttpSession hs = request.getSession(false);
if(hs == null){
    //send back to logon screen
```
response.setContentType("text/html");
    response.sendRedirect("/xml/LogonForm.xml");
}
try{
    //check if username exists in session, else back to logon sceeen
    Object un = hs.getValue("username");
    if (un == null ){
        response.setContentType("text/html");
        response.sendRedirect("/xml/LogonForm.xml");
    }
} catch(Exception e){
    response.sendRedirect("/xml/LogonForm.xml");
}

<tsx:dbquery id="qry" connection="a118">
    select * from products where UCASE(PRODUCT_ID) LIKE '%%%<%= search %>%%%'
    OR UCASE(PRODUCT_NAME) LIKE '%%%<%= search %>%%%'
    OR UCASE(DESCRIPTION) LIKE '%%%<%= search %>%%%'
</tsx:dbquery>

<Product>
<tsx:repeat>
    <Product PRODUCT_ID="<tsx:getProperty name="qry" property="PRODUCT_ID"/>">
        <Product_NAME><tsx:getProperty name="qry" property="PRODUCT_NAME" /></Product_NAME>
        <DESCRIPTION><tsx:getProperty name="qry" property="DESCRIPTION" /></DESCRIPTION>
        <SUPPLIER_ID><tsx:getProperty name="qry" property="SUPPLIER_ID" /></SUPPLIER_ID>
        <SUPPLIER_PART_NO><tsx:getProperty name="qry" property="SUPPLIER_PART_NO" /></SUPPLIER_PART_NO>
        <UNIT_PRICE><tsx:getProperty name="qry" property="UNIT_PRICE" /></UNIT_PRICE>
        <UNITS_IN_STOCK><tsx:getProperty name="qry" property="UNITS_IN_STOCK" /></UNITS_IN_STOCK>
        <MINIMUM_STOCK_LEVEL><tsx:getProperty name="qry" property="MINIMUM_STOCK_LEVEL" /></MINIMUM_STOCK_LEVEL>
    </Product>
</tsx:repeat>
<SEARCH_STRING><%= search %></SEARCH_STRING>
</PRODUCT_LIST>

Chapter 9. XML application implementation  115
The output XML document contains all the information regarding the products matching that user’s interests. This can be used by any other connecting applications. The XSL which renders this XML document uses the product id, product name, description, and price. The user has the option to specify a quantity of the item and order it. The user can also start searching again. The last user input string is also displayed to enhance the search from the user’s point of view. The XSL file which transforms the XML document (output of JSP) is as follows:

```xml
```
Let us suppose that the user provides an input string “printer”. The result of the combined effort of the JSP and the XSL, which we described before, looks as shown in Figure 31.

```xml
<td><input type="text" name="_PNAME_{@PRODUCT_ID}" size="40" value=""/></td>
</tr>
</xsl:for-each>
</TBODY>
</TABLE>
<br></br><br></br>
<CENTER>
<INPUT TYPE="submit" name="addtocorder" VALUE="ADD TO ORDER"></INPUT>
<br></br>
</FORM>

<FORM action="search.jsp">
<br></br><br></br>
<hr width="100%"></hr>
<br></br>
<b>Search String : </b>
<br>
<xsl:apply-templates select="PRODUCT_LIST//SEARCH_STRING"/>
<br></br><br></br>
<CENTER>
<INPUT TYPE="submit" name="searchagain" VALUE="SEARCH AGAIN"></INPUT>
<br>
<INPUT TYPE="reset" name="reset" VALUE="CLEAR"></INPUT>
<br></br>
</FORM>
</BODY>
</HTML>
</xsl:template>
<xsl:template match="SEARCH_STRING">
<input type="text" name="searchstr" size="40" value="{.}"/>
</xsl:template>
</xsl:stylesheet>

Let us suppose that the user provides an input string “printer”. The result of the combined effort of the JSP and the XSL, which we described before, looks as shown in Figure 31.
9.4 Implementation of shopping basket and ordering system

When a user selects “Add to Order” on the search screen, the system needs to add the selected items into a “shopping basket”. The shopping basket is an object that must live in the HttpSession for the user — so that it can remain between different pages.

We chose to implement the shopping basket as a DOM object. This had two main benefits:

1. Displaying the current order involves simply applying a stylesheet to the current basket XML document.
2. When the customer places an order, the XML tree detailing the order already exists.

9.4.1 Implementing the shopping basket as a JavaBean

The shopping basket is implemented as a JavaBean — BasketBean. This bean has a very simple interface, which means that it is suitable to be used by a JSP page. The bean allows a product quantity to be set, the order to be placed or cleared, and finally the XML to be retrieved as a string. Figure 32 shows the architecture of the order subsystem.

![Figure 32: Overview of architecture for the order subsystem](image)

The core of this subsystem is the BasketBean. The BasketBean has the interface shown in Figure 33.
9.4.2 Program Flow

The BasketBean is always owned and managed by a JSP — orderbasket.jsp. This JSP is called by the search results page, and it also calls itself with updated data. The interaction is shown in Figure 34.

Figure 33. Public Interface of itso.xmlapp.beans.BasketBean

```java
public BasketBean();
public void clearOrder();
public void setCustomerOrderID(String custid);
public void setUsername(String uname);
public void setOrderStatus(String stat);
public void setProductQuantity(String pcode, String quant);
public void placeOrder();
public Node getOrderDOM();
public String getOrderXMLString();
```

Figure 34. Interaction between search results and orderbasket.jsp
9.4.3 The orderbasket.jsp file

The program flow is as follows:

1. The user first requests the owning JSP, which has a `<jsp:useBean>` tag. This causes the constructor `BasketBean()` to be called, which creates a unique order number, and a DOM tree consisting of an order with no items. The order status defaults to `NEW`.

2. The JSP then extracts the Username from the session, and calls the `setUsername()` method. If the username is not available in the session, the JSP will redirect the user to the logon page.

3. The JSP looks for parameters like `_PNAME_PRODID=QUANT`, where `PRODID` is a valid product code, and `QUANT` is a number. For each of these, the JSP will call `setProductQuantity(PRODID, QUANT)` on the BasketBean.

4. The JSP looks for a parameter `CUSTID=` and if this is present, it calls `setCustomerOrderID()` on the BasketBean, and also updates the status to `UNSUBMITTED`.

5. The JSP looks for a parameter `PLACEORDER`, which corresponds to the Place Order button. If this is present, it will update the status to `PENDING`, call `placeOrder()`, display the order, and then clear the order using `clearOrder()`. This is linked to the XSL stylesheet which displays `PENDING` orders differently than `NEW` or `UNSUBMITTED` orders. Details of the XSL stylesheet appear later in this chapter.

6. If there is a parameter `CLEARORDER`, which corresponds to the Clear Order button, the JSP calls the `clearOrder()` method, which re-initializes the BasketBean with a new empty order.

7. Finally, the JSP returns the updated DOM tree as a string as the output of the JSP.

The JSP code follows:

```jsp
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="http://localhost/xsl/basket.xsl"?>
<jsp:directive.page contentType="text/xml"/>
<jsp:useBean id="basket" class="itso.xmlapp.beans.BasketBean" scope="session">
</jsp:useBean>

String redirectpage = "/xmlapp/XML/OrderMenu.xml";
String username = (String)request.getSession(true).getValue("username");
boolean orderplaced=false;
String orderno=null;
if (username == null)
{
    response.setContentType("text/html");
```
try {
    basket.setUsername(username);
    java.util.Enumeration en = request.getParameterNames();
    while (en.hasMoreElements()) {
        String paramName = (String)en.nextElement();
        if (paramName.startsWith("_PNAME_")) {
            String prodid = paramName.substring(7); // strip off _PNAME_
            String quantity = request.getParameterValues(paramName)[0];
            if (quantity != null && !quantity.equals("")) {
                int q = 1;
                try {
                    q = Integer.parseInt(quantity);
                } catch (Exception e) {} 
                basket.setProductQuantity(prodid, (new Integer(q)).toString());
            }
        }
    }
    en = request.getParameterNames();
    while (en.hasMoreElements()) {
        String paramName = (String)en.nextElement();
        if (paramName.equals("CUSTID")) {
            basket.setCustomerOrderID(request.getParameterValues(paramName)[0]);
            basket.setOrderStatus("UNSUBMITTED");
        }
        if (paramName.equals("PLACEORDER")) {
            basket.placeOrder();
            basket.setOrderStatus("PENDING");
            orderplaced = true;
        }
        if (paramName.equals("CANCELORDER")) {
            basket.clearOrder();
        }
    }
} catch (Exception e) {
    out.println("<ERROR>");e.printStackTrace();out.println("</ERROR>");
}%

<%= basket.getOrderXMLString() %>
<% if (orderplaced) basket.clearOrder(); %>

This JSP generates XML, and includes a reference to an XSL stylesheet — basket.xsl. This stylesheet formats the XML as HTML so that the browser can view the order. The stylesheet and JSP are very closely linked, because the stylesheet defines the form and buttons which call the JSP again, for example, to cancel or place an order. This is an example where combining the JSP with the XSL using the XSLT Islands technology could be a good idea, because then the form and the handler would be in the same file, and could be more closely linked in code.
9.4.4 BasketBean implementation

Before we look at the XSL code, we will examine the implementation of the BasketBean. The BasketBean relies on a helper class. The helper class is called DatabaseDom, and it is used to map into the underlying database. In effect, DatabaseDom is a very simple persistence library for XML. DatabaseDom is described in more detail in Chapter 10, “Use and implementation of the DatabaseDom bean” on page 149.

9.4.4.1 Overview of DatabaseDom

DatabaseDom has a very simple interface. Most of the aspects of DatabaseDom are configured using a template XML file, which contains details of the database table and the mapping between columns and XML elements and attributes. Although there are some pre-existing Java classes which map databases into XML and back, we did not find any that mapped to arbitrary XML structures. Since we were working from defined DTDs, we would have had to write a lot of DOM manipulation code to use a class or bean which generated a row or column based XML structure. We therefore chose to write a support class which would map from almost arbitrary XML structures into databases and back again.

Once a DatabaseDom object has been initialized with an XML template file, the programmer can modify the WHERE clause of the select statement, and then call getDomTree() to access a DOM tree containing dynamically generated XML based on the database contents. This implements database to XML.

Figure 35 shows a fragment of the PRODUCTS template. This is an XML tree, with a top level element PRODUCTSET and a second level element PRODUCT, which has a number of sub-elements and attributes. The elements and attributes contain column names from the PRODUCTS table. Underneath the template is a sample PRODUCTS table, showing the column names and values for two rows. Finally, the result XML shows the XML which DatabaseDom will provide, based on this template and table.

This greatly simplifies building XML-based Web applications. Almost all Web applications are based on relational databases, and the ability to create complex XML structures from relational data. Although DatabaseDom cannot support complex mappings with multiple tables, simple DOM manipulation can be used to implement more complex mappings.
The DOM tree can also be updated, or a new DOM tree can be created. This can be then be “written” to the database by calling the `updateDB(DomTree)` method. The class identifies for each branch of the XML tree whether this is a new row (insert) or an existing row (update) based on the primary keys. This implements XML to database.

We will now briefly cover the logic of the main public methods of BasketBean.

### 9.4.4.2 BasketBean() constructor

The constructor creates a new order. The order is stored as a DOM tree in memory. A new ORDER element is created, and it is populated with the following attributes:

- ORDER_DATE — the current date in valid SQL date format
- STATUS — NEW
- ORDER_NO — a unique order id
Each order requires a unique order number, and we chose to create this by mapping a function of DB2 which generates a unique number 26 digits long based on the current time and date. It also could have been created using Java functionality.

The code fragment that does this is:

```java
root = currOrder.createElement("ORDER");

// DatabaseDom to create new order number
DatabaseDom oh = new DatabaseDom("/xmlapp/unique.xml");
String ORDER_ID = ((Element)oh.getDomTree()).getElementsByTagName("ORDER_NO").item(0).getFirstChild().getNodeValue();

oh = null; // allow Garbage Collection on unused item
root.setAttribute("ORDER_DATE", (new java.sql.Date(java.lang.System.currentTimeMillis())).toString());
root.setAttribute("ORDER_NO", ORDER_ID);
this.setOrderStatus("NEW");
currOrder.appendChild(root);
```

9.4.4.3 getOrderXMLString()

This calls the XML4J-specific interfaces which can generate XML from a DOM object:

```java
StringWriter sw = new StringWriter();
Visitor visitor = new FormatPrintVisitor(sw);
new NonRecursivePreorderTreeTraversal(visitor).traverse(root);
rs = sw.toString();
```

This code returns the current DOM order object as XML

9.4.4.4 setProductQuantity()

This method has two parameters, a product code and a product quantity.

The basic approach is to cycle through the existing order, and compare the existing ORDER_ITEMS to the given product:

```java
NodeList elements = currOrder.getElementsByTagName("ORDER_ITEM");
boolean foundExisting = false;
for (int i = 0; i < elements.getLength(); i++) {
    Element orderItem = (Element)elements.item(i);
    // compare orderItem with the given product
    if (orderItem.equals(givenProduct)) {
        foundExisting = true;
        // update existing orderItem with the new quantity
        break;
    }
}
```
String pID =
((Element)orderItem.getElementsByTagName("PRODUCT").item(0)).getAttribute("PRODUCT_ID");
if (prodId.trim().equals(pID.trim())) {
    If the product is part of the existing order, and the updated quantity is 0, the product is deleted from the current order DOM object.
    orderItem.getParentNode().removeChild(orderItem);

    Otherwise it modifies the quantity:
    orderItem.getElementsByTagName("ORDER_QUANTITY").item(0).getFirstChild().setNodeValue(q);

    Finally, if the product does not exist in the current order, and the quantity is not zero, we need to create a new XML branch for this product. This is more complex, because the branch must contain data about the product from the database. This is achieved using a DatabaseDom object that maps into the PRODUCTS table in the database.

    if (!foundExisting && !q.equals("0")) {
        // get product details - this implementation is from database
        DatabaseDom db = new DatabaseDom("/xmlapp/products.xml");
        db.setWhereClause("WHERE PRODUCT_ID='"+prodId+"'");
        Node result = db.getDomTree();
        Element prod = null;
        NodeList prods = ((Element)result).getElementsByTagName("PRODUCT");
        if (prods.getLength() == 0) { // prod not found
            // don't create element
        } else {
            Element newOrderItem = currOrder.createElement("ORDER_ITEM");
            newOrderItem.setAttribute("STATUS","NEW");
            newOrderItem.setAttribute("SHIP_DATE","");
            int numExistingItems = ((Element)root).getElementsByTagName("ORDER_ITEM").getLength();
            newOrderItem.setAttribute("ITEM_NO",Integer.toString(numExistingItems+1));

            newOrderItem.setAttribute("ORDER_NO",root.getAttribute("ORDER_NO"));
            prod = (Element)prods.item(0);
            newOrderItem.appendChild(prod);
            Element quant = currOrder.createElement("ORDER_QUANTITY");
            Text qt = currOrder.createTextNode(q);
            quant.appendChild(qt);
        }
    }
newOrderItem.appendChild(quant);
root.appendChild(newOrderItem);
}
}

Finally, in Figure 36, we show the template file /xmlapp/products.xml:

```xml
<?xml version="1.0"?>
<XMLDATABASEMAP>
  <TEMPLATE>
    <PRODUCTSET>
      <PRODUCT PRODUCT_ID="PRODUCT_ID"
SUPPLIER_PART_NO="SUPPLIER_PART_NO">
        <SUPPLIER>SUPPLIER_ID</SUPPLIER>
        <PRODUCT_NAME>PRODUCT_NAME</PRODUCT_NAME>
        <DESCRIPTION>DESCRIPTION</DESCRIPTION>
        <STOCK_LEVEL>UNITS_IN_STOCK</STOCK_LEVEL>
        <UNIT_PRICE>UNIT_PRICE</UNIT_PRICE>
        <MINIMUM_QUANTITY>MINIMUM_STOCK_LEVEL</MINIMUM_QUANTITY>
      </PRODUCT>
    </PRODUCTSET>
  </TEMPLATE>
</XMLDATABASEMAP>
```

Figure 36. Template file PRODUCTS.XML

The template file has two main elements: TEMPLATE and DATABASE.

TEMPLATE contains an XML tree which will be used as a template for the final dynamically generated XML tree. The top level element of TEMPLATE is assumed to be a holder for a set of row elements. In this case it is a PRODUCTSET element. The first child of that element is the template for each row. Every text node and attribute in the template is matched against column names in the database table. If it matches, then it will be replaced in
the final XML generated by the value of that column. This means that the structure of the XML tree can be quite complex, and the XML element names need not match the database column names. It also allows some columns to match to attributes and others to elements.

The DATABASE element contains the details of the table required for Java to access the table.

9.4.4.5 placeOrder()

The final method we will describe is the placeOrder() method. To place an order requires two main steps. Firstly, we need to update the database with the order data. Secondly, we need to send an XML message to each of the suppliers containing only their XML data.

Writing the order to the database

Writing the order data to the database ought to be easy, because we have an XML-to-database mapping class. Unfortunately, this class only supports a mapping from a DOM tree to a single table. The order data is stored in two linked tables: ORDER_HEADERS and ORDER_ITEMS. However, with some very simple manipulation of the XML tree, we can use the DatabaseDom class to update these tables with the correct data.

The first step is to write the header to the database. Because there is a foreign key relationship between the headers and items in the database, the header must exist before we can write the items.

The whole order DOM tree corresponds to a single row in the ORDER_HEADERS database. In theory, we need an <ORDERSET> tag to hold this row, because the DatabaseDom class is built around working with a set of rows. However, DatabaseDom is implemented to search for rows in the DOM tree using getElementsByTagName(), which searches at any depth. Therefore, by creating a template with the row element the same as the ORDER element in the database, we can bypass this step. Furthermore, the ORDER element in our DOM tree also contains a number of ORDER_ITEMS. However, these will be ignored by DatabaseDom, because it doesn’t contain mappings from these branches.

Here is the code that writes the header to the database:

```java
DatabaseDom oh = new DatabaseDom("/xmlapp/order_header.xml");
oh.updateDB(this.getOrderDOM());
```
The ORDER_HEADERS map file looks like this:

```xml
<?xml version="1.0"?>
<XMLDATABASEMAP>
  <TEMPLATE>
    <ORDERSET>
      <ORDER ORDER_DATE="ORDER_DATE"
             ORDER_NO="ORDER_NO"
             CUSTOMER_ORDER_ID="CUSTOMER_ORDER_NO"
             USERNAME="USERNAME" />
    </ORDERSET>
  </TEMPLATE>
</XMLDATABASEMAP>
```

Writing the ORDER_ITEM element to the ORDER_ITEMS table is just as easy. The ORDER element corresponds to a set of rows, and so by using a corresponding template, we can update the ORDER_ITEMS table. Note that not all the elements of the tree correspond to columns in the database table. By creating a template which is a subset of the DOM tree, we only map the subset into the table.

The template is:

```xml
<?xml version="1.0"?>
<XMLDATABASEMAP>
  <TEMPLATE>
    <ORDER>
      <ORDER_ITEM STATUS="ITEM_STATUS"
                   SHIP_DATE="SHIP_DATE"
                   ITEM_NO="ITEM_NO"
                   ORDER_NO="ORDER_NO">
        <PRODUCT PRODUCT_ID="PRODUCT_ID">
          <UNIT_PRICE>ITEM_PRICE</UNIT_PRICE>
        </PRODUCT>
        <ORDER_QUANTITY>ORDER_QUANTITY</ORDER_QUANTITY>
      </ORDER_ITEM>
    </ORDER>
  </TEMPLATE>
</XMLDATABASEMAP>
```
The code to update the ORDER_ITEMS table is:

```java
databaseDom oi = new DatabaseDom("/xmlapp/order_items.xml");
oi.updateDB(this.getOrderDOM());
```

**Sending the XML message to each supplier**

The XML message is sent to each supplier using the standard application DTD that we initially wrote. However, we do not want to send the whole order to each supplier, because that would give one supplier competitive information about what orders the other suppliers were fulfilling. To deal with this, we simply filter the order DOM tree for each supplier, and send each supplier only the items in the XML tree which relate to that supplier.

The code required to do this involves simple manipulation of the DOM tree.

The first step is to create a list of the suppliers referenced in this order:

```java
NodeList supplnl =
    ((Element)this.getOrderDOM()).getElementsByTagName("SUPPLIER");

Vector supps = new Vector();
for (int i = 0; i < supplnl.getLength(); i++) {
    String sn = supplnl.item(i).getFirstChild().getNodeValue();
    if (!supps.contains(sn)) supps.addElement(sn);
}
```

There are a few ways we could filter for each supplier. We chose to create a new copy of the root element (the order), without any children (order items).
Then, for a given supplier, we added in the order_items that were destined for that supplier. The code to do this follows:

```java
Node sXML = this.getOrderDOM().cloneNode(false); // childless copy

NodeList orderitems =
((Element)this.getOrderDOM()).getElementsByTagName("ORDER_ITEM");

for (int i = 0; i<orderitems.getLength();i++)
{
    // get supplier of this particular item
    String oiSupplier =
    ((Element)orderitems.item(i)).getElementsByTagName("SUPPLIER").item(0).
    getFirstChild().getNodeValue();
    if (supplier.equals(oiSupplier)) {
        sXML.appendChild(orderitems.item(i).cloneNode(true));
    }
}
```

Once again, the power of the DOM interface makes it very easy to do this kind of operation. The last step is to use the Java net classes to POST the XML to the supplier as an HTTP message.

We have now described the orderbasket JSP page, and the BasketBean. The final aspect of this subsystem is the XSL stylesheet: `basket.xsl`.

### 9.4.5 XSL stylesheet implementation — basket.xsl

The stylesheet for the order page is implemented as a set of templates. We chose this style because of the clarity and flexibility it gives compared to using the iterative approach (`<xsl:for-each>`).

In general, the XSL stylesheet is fairly straightforward. The XSL generates a form which both displays the current order, and allows the user to modify it, place an order, cancel the order, or jump to the search page. The interesting aspects of the XSL stylesheet are:

- The stylesheet gives different responses based on whether the order has been submitted or not. If the order STATUS is `NEW` or `UNSUBMITTED`, then the form is enabled, and users can modify the order. Once the order is `PENDING`, the user can only view the data, and a link is shown to view the status of this order in the order tracking system. This is implemented using the `<xsl:if>` tags.
- The stylesheet calculates the total cost of each item by using simple arithmetic and XSL variables.
The results of applying the stylesheet are shown in Figure 37.

Figure 37. Screen capture of the orderbasket view
9.5 Order tracking

This section looks at how users can track their current and past orders and get updates on their shipping statuses. This functionality is provided for the most part by a Java servlet (itso.xmlapp.servlets.BrowseOrderServlet) which makes use of the itso.xmlapp.beans.DatabaseDOM class mentioned in the previous section.

9.5.1 Servlet overview

The BrowseOrderServlet class provides two different types of functionality:

- Given an order number as an HTTP parameter, the servlet retrieves from the database the corresponding order information, including the order header and related order items. This information is then built into a DOM tree and displayed to the user through an XSL stylesheet, applied using LotusXSL.

- If no order number is provided, the servlet will, by default, retrieve all the order information for the current user, based on the information contained in the active session. Following a process similar to the one mentioned in the previous point, an XSL stylesheet will be applied to a DOM tree comprising the list of orders.

In the approach discussed here, the XSL is applied at the server to produce HTML that is directly sent back to the browser. This is achieved through the use of the LotusXSL package. This package is IBM’s implementation of the XSLT specification and is supported by the WebSphere Application Server. This programmatic application of XSL and its implications are described in more detail in 7.3, “Applying XSL to XML with WebSphere Application Server” on page 72.

The BrowseOrderServlet class extends the HttpServletRequest class and takes two parameters of type HttpServletRequest and HttpServletResponse, either through a POST or GET operation from the browser. Upon being called, the servlet will attempt to retrieve a valid HttpSession object from the parameter HttpServletRequest. If the session is found to be invalid or non-existent, the user is automatically redirected to the logon page. Similarly, the session must contain a “username” string value which will be required for the servlet to be able to complete its processing.
The servlet will also check the stream for an additional “orderno” parameter. This latter parameter is optional, but is nonetheless used to determine the output generated by the servlet: If a valid “orderno” parameter (that is, with a value different from null) is sent to the stream, HTML is generated to display detailed information about an individual purchase order. In case of an invalid or blank parameter, HTML is generated to display a list of all purchase orders for the current user.

The following abstract, although stripped of most of the code, only intends to show a brief outline of the servlet’s structure rather than an exhaustive listing of the source code:

```java
public void doGet(HttpServletRequest request, HttpServletResponse response) {
    ... try {
        // if the session fails, redirect the user to the login page
        if (session == null) {
            // if order number is supplied, retrieve corresponding order only
            if (orderno != null && orderno.length() > 0) {
                // create the XML Document
                xmlDoc.appendChild(generateOrderDetail(orderno));
                // Process the XML Document
                ...
            } else {
                // create the XML Document
                xmlDoc.appendChild(generateOrderList(username));
                // Process the XML Document
                ...
            }
        } catch (Exception e) {
            ...
        }
    }
}
```

A complete listing of the source code can be found in the source files.

### 9.5.2 Displaying order reports

When no order number is supplied, the servlet retrieves by default all purchase order information related to the current user. In this section, we will take a closer look at the `generateOrderList()` method, which returns a DOM tree containing the list of orders.
All database queries are handled through the DatabaseDOM interface (see Chapter 10, “Use and implementation of the DatabaseDom bean” on page 149). Therefore, in order to retrieve the list of orders from the database, the first step is to instantiate a new `DatabaseDOM` object:

```java
String headerURL = "/xmlapp/orders_map.xml";
...
orderDOM = new DatabaseDom(headerURL);
```

The `orders_map.xml` file is an XML document which is used when the DatabaseDOM object is instantiated to map column values to XML elements:

```xml
<?xml version="1.0"?>
<XMLDATABASEMAP>
<TEMPLATE>
  <ORDERSET>
    <ORDER ORDER_NO="ORDER_NO"
           CUSTOMER_ORDER_NO="CUSTOMER_ORDER_NO"
           USERNAME="USERNAME"
           ORDER_DATE="ORDER_DATE" />
  </ORDERSET>
</TEMPLATE>
</XMLDATABASEMAP>
```

From the above listing, we can see that the information enclosed within the `<TEMPLATE>` tag provides column mappings to the `<ORDER>` element and its attributes. The Document Type Declarations (DTDs) for these follow:

```xml
<!ELEMENT Order (Order_Item_List)>
<!ATTLIST Order
  Order_No ID #REQUIRED
  Customer_Order_No CDATA #IMPLIED
  ORDER_DATE CDATA #REQUIRED
```

The information contained within the `<DATABASE>` tag provides the DatabaseDOM object with parameters required in order to perform the SQL query, such as the datasource name and database driver.

On the other hand, the query statement itself is not provided in the template, but is dynamically built inside the code. The DatabaseDOM then executes the query, and a DOM tree is built from the result set. The source code written to achieve this follows:
// build SQL query
sSQL = new StringBuffer();
sSQL = new StringBuffer();
sSQL.append(" SELECT * FROM ORDER_HEADERS ")
 .append(" WHERE ORDER_HEADERS.USERNAME = "
 .append("  + username + " ");

orderDOM.setSelect(sSQL.toString());

// get Order Headers Tree
orderResult = orderDOM.getDomTree();

Once the tree is built, the generateList() method returns a object of the Node type to the calling program. 9.5.4, “Generating the HTML output” on page 139, explains how XSL is applied to the tree and sent to the browser as HTML. The source code for the generateList() method follows:

```java
public static Node generateOrderList(String username)
{
    String headerURL = "/xmlapp/orders_map.xml";
    DatabaseDom orderDOM;
    Node orderResult = null;
    StringBuffer sSQL;
    try
    {
        orderDOM = new DatabaseDom(headerURL);

        // build SQL query for this particular order
        sSQL = new StringBuffer();
        sSQL.append(" SELECT * FROM ORDER_HEADERS ")
            .append(" WHERE ORDER_HEADERS.USERNAME = "
            .append("  + username + " ");

        // get Order Headers Tree
        orderDOM.setSelect(sSQL.toString());
        orderResult = orderDOM.getDomTree();
    }
    catch (Exception e)
    {
        // error-handling
        e.printStackTrace();
    }
    return orderResult;
}
```

9.5.3 Retrieving individual order information

This section looks at the generateOrderDetail() method, which returns a DOM tree containing information for a single purchase order.

Because the order information related to a purchase order is spread across two database tables, ORDER_HEADERS and ORDER_ITEMS, we need to instantiate not only one but two DatabaseDOM objects:

```java
orderDOM = new DatabaseDom("/xmlapp/orders_map.xml");
itemDOM = new DatabaseDom("/xmlapp/orderitems_map.xml");
```
The `orderDOM` object uses the same XML document as in the previous section, `orders_map.xml`, as a map for the order header component. The `itemDOM` object is instantiated using the following document, `orderitems_map.xml`:

```xml
<?xml version="1.0"?>
<XMLDATABASEMAP>
  <TEMPLATE>
    <ORDER_ITEM_LIST>
      <ORDER_ITEM STATUS="ITEM_STATUS" SHIP_DATE="SHIP_DATE" ITEM_NO="ITEM_NO">
        <ORDER_QUANTITY>ORDER_QUANTITY</ORDER_QUANTITY>
        <PRODUCT PRODUCT_ID="PRODUCT_ID" SUPPLIER_PART_NO="SUPPLIER_PART_NO">
          <PRODUCT_NAME>PRODUCT_NAME</PRODUCT_NAME>
          <DESCRIPTION>DESCRIPTION</DESCRIPTION>
          <SUPPLIER_ID>SUPPLIER_ID</SUPPLIER_ID>
          <PRICE>UNIT_PRICE</PRICE>
          <STOCK_QUANTITY>UNITS_IN_STOCK</STOCK_QUANTITY>
          <MINIMUM_QUANTITY>MINIMUM_STOCK_LEVEL</MINIMUM_QUANTITY>
        </PRODUCT>
      </ORDER_ITEM>
    </ORDER_ITEM_LIST>
  </TEMPLATE>

  <DATABASE>
    <MAXRETURNROWS>100</MAXRETURNROWS>
    <JDBC>
      <URL>jdbc:db2:a118</URL>
      <DRIVER>COM.ibm.db2.jdbc.app.DB2Driver</DRIVER>
    </JDBC>
    <USERID>xmlguru</USERID>
    <PASSWORD>xmlguru</PASSWORD>
  </DATABASE>
</XMLDATABASEMAP>
```

Following the same procedure as we did in the previous section, we build the SQL query for each object dynamically. Two different queries are required:

- A query to retrieve the order header information from the `ORDER_HEADERS` table:

  ```java
  // build SQL query to retrieve order header
  headerSQL = new StringBuffer();
  headerSQL.append(" SELECT * FROM ORDER_HEADERS ");
  headerSQL.append(" WHERE ORDER_HEADERS.ORDER_NO = ");
  headerSQL.append(" " + orderno + " ");
  
  // build SQL query to retrieve order items
  itemSQL = new StringBuffer();
  itemSQL.append(" SELECT * FROM ORDER_ITEMS, PRODUCTS ");
  itemSQL.append(" WHERE ORDER_ITEMS.PRODUCT_ID = PRODUCTS.PRODUCT_ID ");
  itemSQL.append(" AND ORDER_ITEMS.ORDER_NO = ");
  itemSQL.append(" " + orderno + " ");
  ```
As a result, two DOM trees are generated, one which contains a single
<Order> element for the order_header and another one which includes one
or more <Order_Item> elements. To create a single tree, the order items
are appended to the order header in the following statement:

```java
// get Order Header Tree
orderDOM.setSelect(headerSQL.toString());
orderResult = orderDOM.getDomTree();

// retrieve item tree
itemDOM.setSelect(itemSQL.toString());
itemResult = itemDOM.getDomTree();

// append item tree to this order node
orderResult.getFirstChild().appendChild(itemResult);
```

The resulting single tree is then returned to the caller. The next section
explains how the tree is processed to create HTML. The source code for the
generateOrderDetail() method follows:

```java
public static Node generateOrderDetail(String orderno)
{
  try
  {
    String headerURL = "/xmlapp/orders_map.xml";
    String detailURL = "/xmlapp/orderitems_map.xml";
    DatabaseDom orderDOM;
    DatabaseDom itemDOM;
    Node orderResult = null;
    Node itemResult;
    StringBuffer headerSQL;
    StringBuffer itemSQL;
    orderDOM = new DatabaseDom(headerURL);
    itemDOM = new DatabaseDom(detailURL);

    // build SQL query to retrieve order header
    String headerSQL = new StringBuffer();
    headerSQL.append(" SELECT * FROM ORDER_HEADERS ");
    headerSQL.append(" WHERE ORDER_HEADERS.ORDER_NO = ");
    headerSQL.append(orderno);

    // build SQL query to retrieve order items
    String itemSQL = new StringBuffer();
    itemSQL.append(" SELECT * FROM ORDER_ITEMS, PRODUCTS ");
    itemSQL.append(" WHERE ORDER_ITEMS.PRODUCT_ID = PRODUCTS.PRODUCT_ID ");
    itemSQL.append(orderno);

    // get Order Header Tree
    orderDOM.setSelect(headerSQL.toString());
    orderResult = orderDOM.getDomTree();

    // retrieve item tree
    itemDOM.setSelect(itemSQL.toString());
    itemResult = itemDOM.getDomTree();

    // append item tree to this order node
    orderResult.getFirstChild().appendChild(itemResult);
  }
```

138 The XML Files: Using XML and XSL with IBM WebSphere 3.0
9.5.4 Generating the HTML output

9.5.4.1 The XSL processor

Once DOM trees have been generated, the servlet must send the information back to the browser in HTML format. The reasoning behind serving HTML rather than raw XML for the browser to interpret is explained in 7.3, “Applying XSL to XML with WebSphere Application Server” on page 72.

The transformation of the XML data is achieved by using the LotusXSL package. This package provides methods to programmatically invoke the XSL processor and XSL stylesheets to XML DOM trees and is implicitly supported by WebSphere.

```java
// retrieve the XSL URL
URL ssURL = new URL("http://localhost/xsl/orderdetail.xsl");
InputStreamReader reader = new InputStreamReader(ssURL.openStream());

// create the XML Document
Document xmlDoc = (Document)
Class.forName("com.ibm.xml.parser.TXDocument").newInstance();
xmlDoc.appendChild(generateOrderDetail(orderNo));

// Process the XML Document
XSLProcessor xslProcess = new XSLProcessor();
xslProcess.process(xmlDoc, reader, out);
```

A description of the above source code abstract follows:

- In this example, the `generateOrderDetail()` method is called to retrieve the information for a single purchase order.

- An `InputStreamReader` is instantiated and associated to a valid `URL` which points at the XSL stylesheet that we want to apply in this particular case.

- A blank `Document` is created, to which the DOM tree returned by the `generateOrderDetail()` method is appended.

- The `XSLProcessor` is then instantiated and we call the `process` method with the `Document`, the `InputStreamReader` and output writer in order to apply the XSL to the XML data and send it to the output stream.
The same process is repeated when creating an XML tree containing the list or purchase orders but the `generateOrderList()` method is used instead.

### 9.5.4.2 Order list stylesheet

The XSL stylesheet for purchase orders (`orderlist.xsl`) creates a table which contains mainly order header information. Each row in the table also contains a link to a more detailed view of the corresponding purchase order. The transformed XML output can be seen in Figure 38.

```xml
<?xml version="1.0"?>
<!-- FileName: orderlist.xsl -->
<!-- Author: Christophe Chuvan-->
<!-- Created 4th Oct. 1999 -->
<!-- Last Modified 13th Oct. 1999 -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <HTML>
      <HEAD>
        <META http-equiv="Content-Type"
              content="text/html; charset=iso-8859-1"/>
        <META http-equiv="Expires" content="0"/>
        <STYLE TYPE="text/css">
          BODY{font-family : Verdana,sans-serif;}
          .formText{font-family : Verdana,sans-serif;}
          .button{font-family:Verdana,sans-serif;font-size:100%;}
          .tableText{font-weight : bold;font-family : monospace;background-color : white;text-align : right;}
        </STYLE>
      </HEAD>
      <BODY background="/xsl/world.jpg">
        <A href="/"><IMG src="/xsl/xmlapp.gif"></IMG></A>
        <H2>Past and Current Purchase Orders</H2>
        <HR width="100%">
        <HR/>
        <TABLE BORDER="1">
          <THEAD>
            <th>Order No</th>
            <th>Customer Order No</th>
            <th>Username</th>
            <th>Order Date</th>
            <th>Ship Date</th>
          </THEAD>
          <TBODY>
            <xsl:for-each select="ORDERSET/ORDER">
              <tr>
                <td><A href="/xmlapp/servlet/BrowseOrderServlet?orderno={@ORDER_NO}"
                     xsl:value-of select="@ORDER_NO"></A></td>
                <td><xsl:value-of select="@CUSTOMER_ORDER_NO"></td>
```

140  The XML Files: Using XML and XSL with IBM WebSphere 3.0
Figure 38. Order list screen
9.5.4.3 Order details stylesheet
The XSL stylesheet used to format XML data related to a single purchase order (orderdetails.xsl) is voluntarily parsed in its structure. To keep things simple, the data is broken down in two components, the order header and the order items, each displayed in a separate table. The resulting transformed XML output is shown in Figure 39.

```xml
<?xml version="1.0"?>
<!-- FileName: orderdetail.xsl -->
<!-- Author: Christophe Chuvan-->
<!-- Created 4th Oct. 1999 -->
<!-- Last Modified 13th Oct. 1999 -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
  <xsl:output method="html"/>
  <xsl:template match="ORDER">
    <HEAD>
      <META http-equiv="Content-Type"
           content="text/html; charset=iso-8859-1"/>
      <META http-equiv="Expires" content="0"/>
      <STYLE TYPE="text/css">
        BODY{font-family : Verdana,sans-serif;}
        .formText{font-family : Verdana,sans-serif;}
        TD{font-family : Verdana,sans-serif; font-weight: bold;}
        .button{font-family:Verdana,sans-serif;font-size:100%;}
        .tableText{font-weight : bold;font-family : monospace;background-color : white;text-align : right;}
      </STYLE>
    </HEAD>
    <BODY background="/xsl/world.jpg">
      <A href="/"><IMG src="/xsl/xmlapp.gif"></IMG></A>
      <HR width="100%"></HR>
      <H2>Purchase Order Details</H2>
    </BODY>
  </xsl:template>
</xsl:stylesheet>
```

<table>
<thead>
<tr>
<th>Order No</th>
<th>Customer Order No</th>
<th>Username</th>
<th>Order Date</th>
<th>Ship Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Click [A href="/xmlapp/servlet/BrowseOrderServlet"] Here [/A] to go back
</BODY>
</xsl:template>

<xsl:template match="ORDER_ITEM_LIST">

</xsl:template>

<xsl:stylesheet>

</xsl:stylesheet>
9.6 Supplier

After the order has been placed by the ordering system, the supplier receives an XML document describing the order. In the real world, an ordering system may post it to one of several suppliers. The subsystem at the supplier end receives the XML document and performs some minimal checking to find out whether the order placed is valid or not. If the order is not valid, this fact is acknowledged to the ordering system, but no processing takes place on that specific order. If the order is valid, the supplier subsystem picks the order number from the XML document and acknowledges the ordering system. The supplier subsystem stores the XML document and the order number in the supplier database.

Figure 39. Order detail screen
The other part of the supplier subsystem deals with viewing and modification of the received orders known as the Supplier Update and Viewing System. This system uses a JSP and displays all the received orders. The administrator who handles this system has the option to view the order, modify the ship date, and delete the processed order. The JSP which handles the above system is:

```xml
<?xml version="1.0"?>
<?xml-stylesheet href="http://localhost/MainMenu.xsl" type="text/xsl"?>

<!-- FileName: SupMainMenu.jsp -->
<!-- Author: Ramani Ranjan Routray -->
<!-- Created 7th Oct. 99 -->
<!-- Last Modified XXXX -->
<jsp:directive.page contentType="text/xml" />

<tsx:dbconnect id="SUP001" userid="xmlguru" passwd="xmlguru" url="jdbc:db2:Sup001" driver="COM.ibm.db2.jdbc.app.DB2Driver"></tsx:dbconnect>

<tsx:dbquery id="qry" connection="SUP001">
  select * from supplier_storage
</tsx:dbquery>

<ORDERS>
  <tsx:repeat>
    <ENTRY>
      <ORDER_NUMBER>
        <tsx:getProperty name="qry" property="ORDER_NO"/>
      </ORDER_NUMBER>
    </ENTRY>
  </tsx:repeat>
</ORDERS>

The output of this JSP is also an XML document, and is transformed to an HTML document using XSL. The combined output of the JSP and the XSL associated with it looks as shown in Figure 40.
The order numbers provide hyperlinks to view the individual orders (we display the XML document as it is, without any formatting). For modification, the individual orders must be selected, and then the administrator can choose to delete them or modify the ship date. The administrator provides the new ship date to modify the prior ship date. The deletion or modification of any order by the supplier is reported back to the order tracking system to synchronize the transactions. The deletion and modification of orders has been implemented by servlets. These servlets check the session, perform the database operations, and output plain documents to inform the user of the successful completion of the operation.
The structure of the supplier's database is very simple. It stores the order (XML document) against its number. The supplier database contains only one table "Supplier_Storage". This table contains two fields (order_no, order_details). The type of the field order_no is varchar(30) and order_details is of type long varchar.

The basic control flow is described in Figure 41.

Figure 41. Control flow of supplier unit
From the implementation point of view, the servlet `SupplierStorage` parses the incoming XML document and stores or rejects the order. The acceptance (processing and storage) or rejection of the order is also communicated to the ordering system through the same servlet. The main menu at the supplier end is maintained by the JSP `SupMainMen`. The output of this JSP is an XML document. This XML document is transformed to HTML through XSL. At the supplier end, the user can view the details of the order through the servlet `Menu` and can update any order. Update of the order is achieved through a servlet `Update`, which in turn uses another servlet `Modify` for database transactions and reporting back to the ordering system.
Chapter 10. Use and implementation of the DatabaseDom bean

This chapter outlines the implementation of the DatabaseDom helper bean. We cover the usage, limitations, and approach taken. The first part of this chapter is aimed at potential users of DatabaseDom, while the second part is designed to explain the implementation, and is aimed at developers.

10.1 Use, limitations, and approach

DatabaseDom was designed to fill a specific need in the XMLapp application. It provides two functions:

1. DatabaseDom reads data from a JDBC database table and formats it as an XML DOM tree.
2. DatabaseDom reads data from an XML DOM tree and updates a JDBC database table.

The former of these functions is considerably easier to implement than the latter. Reading data from a JDBC database table is simple, and data is easily converted from a database format into the string format required for XML. Converting data from an XML string format into a JDBC datatype is more complex. However this has already been implemented by the IBM Data Access Beans which ship with WebSphere, and so these were used to implement the database update function.

10.1.1 The template file

Fundamental to using the DatabaseDom class is the template file. The template file is built in XML, and consists of two parts — the template itself and the database definition. The database definition is optional, and the database parameters can either be set in the template file, or programmatically in the code. However, most users will define the database parameters in the template file, because it is easier, and it makes sense to keep the table definition and the template together.

In order to present the template file, let us look at a sample template file, based on the EMPLOYEE sample table that ships with DB2 UDB. Figure 42 shows the Data Definition Language (DDL) script which defines the EMPLOYEE table.
The EMPLOYEE table makes a good example, because it employs a variety of SQL datatypes.

Some points worth noting about the template are these:

- The XML structure can be complex — for example, sub-elements and attributes can be built based on the table columns.
- The element names do not need to match the column names.
- The table query can be fully defined using a schema name, table name, and where clause.

The matching template file is presented in Figure 43.
Figure 43. Template file for the EMPLOYEE table

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

<TEMPLATE>

<EMPLOYEE_LIST>

<EMPLOYEE NO="EMPNO" GENDER="SEX">

<NAME>
  <FIRST>FIRSTNME</FIRST>
  <MIDDLE_INITIAL>MIDINIT</MIDDLE_INITIAL>
  <LAST>LASTNAME</LAST>
</NAME>

<PAY>
  <SALARY>SALARY</SALARY>
  <BONUS>BONUS</BONUS>
  <COMMISSION>COM</COMMISSION>
</PAY>

<PERSONAL>
  <DOB>BIRTHDATE</DOB>
  <EDUCATION>EDLEVEL</EDUCATION>
</PERSONAL>

<JOB TITLE="JOB">
  <DEPARTMENT NO="WORKDEPT" />
  <PHONE>PHONENO</PHONE>
  <HIREDATE>HIREDATE</HIREDATE>
</JOB>

</EMPLOYEE>

</EMPLOYEE_LIST>

</TEMPLATE>

</XMLDATABASEMAP>
10.1.2 Definition of template file elements

The definitions of the template elements follow. The *bean method* listed with each element indicates a Java method which can be used to set the values. Using the bean methods overrides the values set in a template file.

**XMLDATABASE**

The root element.

**TEMPLATE**

The owning element of the XML template section. The first and only element of this should be a “rowset” holder. The rowset element will be the root element of the resulting XML DOM tree. It will contain one element per row.

The rowset holder should contain one child element, which is the row template. Each row in the query result will correspond with an element in the result DOM tree. Each of these elements will be based on the row template element, with the column values replaced in the template.

Any column names found as either attribute values, or as text nodes, will be replaced with the column values. Each column name can only be used once, but need not be used. Text nodes or attribute values that do not correspond to column names will be left in the template.

**DATABASE**

This element contains the setup data for the database connection.

**MAXRETURNROWS** (optional)

This element, if present, sets a maximum number of return rows.

*Bean Method*: `setMaxReturnRows(int)`.

**JDBC**

This element contains the elements which define the JDBC connection. It has two sub elements: URL and DRIVER. If the DATASOURCE element exists, these are ignored.

**JDBC/URL**

The JDBC URL.

*Bean Method*: `setURL(String)`

**JDBC/DRIVER**

The JDBC Driver class.

*Bean Method*: `setDBDriver(String)`
DATASOURCE (optional)

This element is for use inside WebSphere Application Server, which supports JDBC 2.0 connection pooling. If this is present, then the DATASOURCE/NAME element is used as in the JNDI lookup to select a DataSource defined in the WebSphere Administrative Console. Usually this would be of the form “jdbc/sample”.

Bean method: setDSName(String)

USERID

This defines a userid which has access rights to the table.

Bean method: setUserid(String)

PASSWORD

This defines a password which matches the userid.

Bean method: setPassword(String)

SELECT (optional)

If a SELECT element is present, this overrides the TABLE and WHERE elements. This specifies a database SELECT statement to use to create the database query. If a SELECT statement is used instead of specifying a TABLE, SCHEMA, and WHERE clause, then the bean cannot be used to update the database. However, using a SELECT statement can be more powerful, because JOINed tables and views can be used to dynamically create XML.

Bean method: setSelect(String)

TABLE (required for update)

This defines the table name of the selected table. The TABLE, SCHEMA, and WHERE clause are used together to create the query (SELECT * FROM SCHEMA.TABLE WHERE). If the SELECT is set, it overrides this.

Bean method: setTableName(String)

SCHEMA (required for update)

The database schema to look for the table in.

Bean method: setSchemaName(String)

WHERE (optional)

The where clause for the query — this should always start with “WHERE”.

Bean Method: setWhereClause(String)
10.1.3 Example output

When the example template in Figure 43 on page 151 is run against the DB2 UDB SAMPLE database, the resulting XML looks as shown in Figure 44.

```xml
<?xml version="1.0"?>
<EMPLOYEE_LIST>
    <EMPLOYEE NO="000250" GENDER="M">
        <NAME>
            <FIRST>DANIEL</FIRST>
            <MIDDLE_INITIAL>S</MIDDLE_INITIAL>
            <LAST>SMITH</LAST>
        </NAME>
        <PAY>
            <SALARY>19180.00</SALARY>
            <BONUS>400.00</BONUS>
            <COMMISSION>1534.00</COMMISSION>
        </PAY>
        <PERSONAL>
            <DOB>1939-11-12</DOB>
            <EDUCATION>15</EDUCATION>
        </PERSONAL>
        <JOB TITLE="CLERK">
            <DEPARTMENT NO="D21"/>
            <PHONE>0961</PHONE>
            <HIREDATE>1969-10-30</HIREDATE>
        </JOB>
    </EMPLOYEE>
    <EMPLOYEE NO="000300" GENDER="M">
        <NAME>
            <FIRST>PHILIP</FIRST>
            <MIDDLE_INITIAL>X</MIDDLE_INITIAL>
            <LAST>SMITH</LAST>
        </NAME>
        <PAY>
            <SALARY>17750.00</SALARY>
            <BONUS>400.00</BONUS>
            <COMMISSION>1420.00</COMMISSION>
        </PAY>
        <PERSONAL>
            <DOB>1936-10-27</DOB>
            <EDUCATION>14</EDUCATION>
        </PERSONAL>
        <JOB TITLE="OPERATOR">
            <DEPARTMENT NO="E11"/>
            <PHONE>2095</PHONE>
            <HIREDATE>1972-06-19</HIREDATE>
        </JOB>
    </EMPLOYEE>
</EMPLOYEE_LIST>
```

Figure 44. Results of applying the Employee template
10.1.4 Bean methods

Apart from methods described in 10.1.2, “Definition of template file elements” on page 152, there are only three significant methods to call on the bean:

- `DatabaseDom(String templateURL)`
  - The constructor, takes the URL of template XML file
- `getDomTree()`
  - Executes the query, creates and returns the DOM tree
- `updateDB(Node domtree)`
  - Updates the database based on the row entries of the given DOM tree.
  
Please observe the restrictions and notes in the next section.

10.1.5 Updating the database

Updating the database places some extra requirements on the use of the `DatabaseDom` bean:

- The `DatabaseDom` can only be used to update a single table.
- If the database has any primary keys, those columns must be represented in the XML template. They will be used to decide whether the update to the database is an SQL UPDATE or INSERT. If the database has no primary key, all updates will create new rows using the SQL INSERT command.
- If any columns are declared as “non-null” in the database, they must exist in the XML template.
- The TABLE and SCHEMA properties must be defined.
- The element names of the template should all be distinct from each other.

10.2 The implementation of the `DatabaseDom` bean

`DatabaseDom` is a combination of Java JDBC, IBM Data Access bean, and DOM programming. We will only cover the XML DOM programming in detail.

The most interesting aspect of `DatabaseDom` is the mapping between the XML DOM tree and the database columns. Because the XML template can be arbitrarily deep, the mapping must be implemented through the use of recursive methods. Recursive methods are those that call themselves, and they are an important programming method for dealing with complex tree structures. For example, most generic DOM programs such as formatters employ recursion.
The mapping is instead implemented as a Java Hashtable, which is a mapping between keys and values. The key is the column name. The value stored is a Path inner class. The inner class Path is required because a complex return type is needed for a recursive method.

The Path class is defined as shown in Figure 45.

```java
static final int UNDECIDED = 0;
static final int ELEMENT = 1;
static final int ATTRIBUTE = 2;

class Path {
    Node node;
    int type;
    Vector tagnames;
    Path(Node n)
    {
        this.node = n;
        this.type = UNDECIDED;
        this.tagnames = new Vector();
    }
    Path(Node n, int type, Vector tn)
    {
        this.node = n;
        this.type = type;
        this.tagnames = tn;
    }
}

Figure 45. The Path inner class definition
```

The aim of the Path class is to define a path to a particular element or attribute. The Path class actually stores two paths to each element or attribute. The first is the Node object in a “reference” DOM tree for the template. The second is a Vector containing the list of tag names from the root of the template to the given leaf node. The reason for storing two different copies of the path is that different approaches are taken for generating a DOM tree from the database, and for updating the database based on an XML DOM tree.
10.2.1 Generating XML from the database

The template object is initially created by parsing the template file. Once the template is parsed, the row template node is stored as a master. An important aspect of the DOM interface is that objects are “live”. Suppose you create an Element, and append it as a child to a Document. Even after the element has been appended, updates to the Element are reflected in the Document.

We created a very simple way of generating an XML fragment based on the template. The original master object has its leaf nodes saved as Node objects in the mapping. The saved nodes are used to update the original master object using the `setNodeValue(String value)` method. Once the updates are in place, the master is cloned using the `cloneNode()` method. This demonstrates some of the power of the “live” behavior of the DOM interface.

10.2.2 Updating the database from an XML DOM tree

Unfortunately, the approach used to create XML fragments will not work to extract data from a given XML fragment. However, the DOM API has excellent support for extracting and finding nodes. In particular, the method `getElementsByTagNames(String)` is very powerful. By recording a list of tag names from the root of the tree to the given node, we can then navigate back along the route at a later time to extract the value of the node at that position in the tree. In addition to recording the list of tag names, we also record if it is an element (a Text node), or an attribute. An example of a simple XML tree, some paths, and the code to extract the values of the nodes are given in Figure 46.

```xml
<Book ISBN="0198537379">
  <BookTitle>The SGML Handbook</BookTitle>
  <Author>Charles Goldfarb</Author>
</Book>
```

Title Path: ELEMENT [BOOK, BOOKTITLE]
Author Path: ELEMENT [BOOK, AUTHOR]

```java
String ISBN = ((Element)Node).getElementsByTagName("BOOK").
  getAttributes().getNamedItem("ISBN").getNodeValue();
String Author =
  ((Element)Node).getElementsByTagName("BOOK").getFirstChild().
  getElementsByTagName("AUTHOR").getFirstChild().getNodeValue();
```

Figure 46. Examples of paths to particular nodes on an XML tree
These paths are generated by a recursive function, and stored in the Path object. The Path object is stored as the value in the mapping Hashtable against the column name.

When the time comes to update the database, the Bean finds the set of nodes which correspond to rows in the table. For each row node, a recursive function takes the Path information and retrieves a value for that column. All the columns in the mapping for that table are retrieved and stored in a “rowdata” Hashtable, and are passed to a method insertOrUpdateRow(Hashdata rowdata). This method attempts to update the row, and if no existing row is found, it inserts the data as a new row.

10.2.3 Creating the column to XML mapping

The original XML mapping is created by recursively searching the XML DOM tree for text nodes and attributes, and then storing the textual values of these as keys in the mapping object. The method is given below in Figure 47.
protected void RecurseAndSetHashMaps(Path p)
{
    // Firstly, get any attributes and add them into the mapping
    NamedNodeMap attrs = p.node.getAttributes();
    if (attrs != null) {
        for (int i = 0; i < attrs.getLength(); i++) {
            // add any attributes into the mapping
            Attr a = ((Element)p.node).getAttributeNode(attrs.item(i).
                   getNodeValue());
            Vector tg = (Vector)p.tagnames.clone();
            tg.addElement(p.node.getNodeName());
            tg.addElement(attrs.item(i).getNodeName());
            addMap(attrs.item(i).getNodeValue(),
                   new Path(attrs.item(i), ATTRIBUTE, tg));
        }
    }
    // Secondly, check if this is a leaf node. If so, add into mapping
    if (p.node.getNodeType() == Node.TEXT_NODE ||
        p.node.getNodeType() == Node.CDATA_SECTION_NODE)
    {
        String key = p.node.getNodeValue().trim();
        if (!key.equals(""))
        {
            addMap(key, new Path(p.node, ELEMENT, p.tagnames));
        }
    }
    // finally, if the node has children, recursively act on them
    if (p.node.hasChildNodes()) {
        NodeList nl = p.node.getChildNodes();
        for (int i=0; i < nl.getLength(); i++) {
            Vector tg = (Vector)p.tagnames.clone();
            tg.addElement(p.node.getNodeName());
            RecurseAndSetHashMaps(new Path(nl.item(i), UNDECIDED, tg));
        }
    }
}

Figure 47. RecurseAndSetHashMaps method of DatabaseDom

This concludes the description of DatabaseDom.
Chapter 11. Conclusions

This chapter draws some conclusions based on our experience building the ITSO XML application, outlines lessons learned, and concludes with suggested areas for future exploration.

11.1 Successes

The ITSO XML application demonstrates:

- “Business-to-Consumer” and “Business-to-Business” use of XML
- An application where the interface is 100% XML + XSL
- The use of XML to database mapping
- The powerful use of the Document Object Model
- The use of server-side XSL transformation
- The different options to generate XML in IBM WebSphere: servlets, DOM, and JSPs

Although we did not have the time or infrastructure to create alternative interfaces for the application, it is quite possible to create a Wireless Markup Language (WLM) enabled version of this application, because every page of the application was generated using a stylesheet.

XML boundaries

One aspect of any XML application is the interface between the XML-based data and the rest of the system. Some examples of the XML boundaries in the application are:

- XML is created from the database by the DatabaseDom object.
- XML is created from the database by JSPs.
- XML is created from user input by servlets and DOM.
- XML is converted to HTML using XSL.
- XML is sent unconverted to a supplier.
- XML is transformed to other XML and then sent to other suppliers.

Figure 48 shows the major transformations used in the ITSO XML application.
Almost all applications need to convert existing data into and out of XML. Storing XML natively is currently not efficient, and most applications need to work from existing databases. However, database technology is developing in the area of XML, and new database extenders will allow XML to be stored and searched directly inside a database, as well as providing the ability to map into and out of existing databases.

The DatabaseDom technology provided an easy way of mapping databases into complex DOM trees.

We minimized conversions to and from DOM trees by storing the current order as a DOM tree in the user's session data. This was a successful strategy because the DOM interface is very powerful, allowing complex manipulations.
Another example where we kept the XML without transforming it was the supplier system. The supplier system was deliberately kept simple, and we kept the XML stored in a database as a string. This meant that implementing the supplier system was very simple.

**Design Approach**

Our design approach was to initially build the data model and the DTDs. From there, we built the Web architecture as a set of flows, and finally we implemented the application.

Building and working from a common DTD provided a lot of structure to the application from the start, and allowed the work to be separated into independent units. For example, each subsystem of the application had well-defined output based on the DTD.

### 11.2 Lessons Learned

Object-oriented design makes use of an excellent architecture called “Model, View, Controller”. The Model is the data and formatting, the View is the presentation, and the Controller contains the process logic. XML and XSL map well into Model and View. However, in our application, the Controller logic was split between servlets, JSPs, and links and logic which were embedded inside XSL stylesheets.

This meant that the application is very flexible to changes in the presentation, but not flexible to changes in the application flow. The “traditional” servlet/JavaBean/JSP model is better designed in this regard.

We thought of two approaches to address these considerations:

- In a future application, we might examine the use of a separate XML data structure to model the links between pages. This could be processed by the XSL stylesheet to provide the links to new applications. In effect, this was partially implemented by the XSL stylesheet that we designed to manage the static XML elements such as the logon form. This approach would mean that the XSL stylesheet would contain two aspects — transformation of the data, and transformation of the controller data into links.

- The second approach would be to use XSLT Islands much more extensively. XSLT Islands would allow the controller logic to be more clearly defined in a single place, the JSP.
11.3 Areas for future exploration

We thought of a number of areas that would bear further exploration. Briefly, these are:

- Defining an XML configuration file for the application, containing the paths of the XML map files, the user and password for access to the database, and other configuration items
- Building a WML interface to the application
- Developing an XSL sheet to transform the DatabaseDom maps into equivalent JSP pages
- Using XSL stylesheets to “farm” existing HTML pages into structured XML data (HTML -> XHTML -> XML).
- Using XML JSPs to build an XML interface to EJB Access Beans, providing non-CORBA clients an XML/HTTP interface to EJBs
Appendix A. WebSphere Installation

This guide is not a replacement for the IBM WebSphere Getting Started guide, which comes with the product. This guide simply takes people through installing a single configuration of WebSphere — Windows NT, Advanced, IBM HTTP Server, DB2, and a single node. The reason for this guide is that many people have asked how to get started, and this is a good starting place.

Software levels used are:

- IBM JDK 1.1.7p Build level 0823.
- IBM HTTP Server 1.3.6 (GA level).

This guide also notes any differences if you are using Advanced Edition.

The steps involved are:

1. Check pre-requisites and network configuration.
2. Create a user for WebSphere and DB2 UDB (Optional)
3. Install HTTP Server and configure.
4. Install DB2 UDB and Fix pack.
5. Install WebSphere, configure, and test.

A.1 Prerequisites

Hardware:

- Pentium II or better, 128 MB minimum, 512 MB recommended
- 200 MB or more free disk space

Software:

- Microsoft Windows NT v4.0 SP4 or SP5
- TCP/IP networking

Important Note

Sun JDK 1.2 causes unexpected results if you have it installed. It installs itself in the Windows System directory, and therefore takes precedence to other JDKs. To prevent these problems, uninstall JDK1.2 before installing WebSphere 3.0.
If you already have DB2 6.1 installed at an earlier Fix pack, you can install the Fix pack over it. If you have DB2 6.1 FP2 already installed, you can skip that section. WebSphere also supports DB2 5.2 at Fix pack level 11.

Note

Users of earlier betas should completely de-install, delete any WebSphere install directories and drop any config databases (WAS). WebSphere v3.0 will *not* install on top of IBM HTTP Server 1.3.3, although the two servers can co-exist, as can WebSphere 2.0 and 3.0.

If you wish to use WebSphere Application Server on a laptop, you should first configure your networking. See Part A.9, “Configuring a laptop/mobile machine with a constant IP address” on page 188 before following the rest of this guide.

**A.2 Create a user for WebSphere and DB2 UDB**

The WebSphere, Web server, and DB2 processes need to run under the permissions of a given user. For a demonstration or development machine it makes life easier to create a new user with administrative rights, and to run these processes under this user’s permissions. This is *not* suitable for a production system.

We chose to create a user named *xmlguru*, and to use this for all the software. We chose to use the password as *xmlguru* as well, because this machine did not require any security. The *xmlguru* password is coded into the sample XMLapp application.

This step is optional, but recommended for an easy life. Start the NT User Manager. Select **User->New User**. Fill in the fields:

- **Username:** xmlguru
- **Full Name:** XML Guru
- **Description:** XML Guru
- **Password and Confirm Password:** As you choose

Make sure that only the “Password Never Expires” field is checked.
The screen should look like the screenshot in Figure 49.

![New User dialog box](image)

*Figure 49. User manager entry for xmlguru user*
Now click on the **Groups** button. Select **Administrators** and then click **Add**. The screen should look like Figure 50.

![Group Memberships](image)

*Figure 50. Group membership for xmlguru*

Click **OK**. Click **OK** again.

Now select the user **xmlguru**, and then select **Policies->User Rights...** from the menu. At the bottom of the dialog box is a field — **Show Advanced User**
**Rights** — select this. Now choose the **Right — Act as part of the operating system** from the drop down list, as in Figure 51.

![User Rights Policy](image1)

*Figure 51. Setting the User Rights for the xmlguru user*

Now click **Add**. On the new dialog box, click **Show Users**, scroll down to the **xmlguru** entry, and click **Add**. Click **OK**, then **OK** again. You can now close the User Manager.

### A.3 Install the IBM JDK 1.1.7P

1. Run the file: **ibm-jdk-n117p-win32-x86.exe**. See Figure 52

![InstallShield Self-extracting EXE](image2)

*Figure 52. Installing the JDK*
2. Click **Yes**, then select **Next** and then accept the license agreement. Leave the defaults and then select **Next**. See Figure 53.

![Select Components](image)

Figure 53. Selecting components for the IBM JDK Installation

3. Select **Next** again. The installation will start.

4. Finally, select **Finish**.

### A.4 Install IBM HTTP Server 1.3.6

1. Run the installation program (for example, `HTTPServer_ex.exe`) Click **Next**, and accept the license agreement. Consider changing the
installation directory for this. The following example in Figure 54 uses C:\http.

Figure 54. Selecting the path to install the IBM HTTP Server 1.3.6

2. Select Yes to create the directory. Then select Next.

3. Select the Typical installation stream, click Next and then Next to accept the Program folder.

4. For the user ID and password, consider using a single Windows NT user ID and password for all of the WebSphere and support software packages — DB2, HTTP and WebSphere. Also consider using id which has administrative rights, although this would not be recommended for a production system. For this redbook we chose the xmlguru.

5. Fill in the user ID and password here. If you need to create a new user ID and password, use the Windows NT User Manager to do this. Note the comments in the README files — if this machine is a Domain Controller you need to perform an additional step, as described in the README file.

6. Click Install. Read the README, and finish the install package.

7. Configure the HTTP Admin server password as described below.

8. Start a command prompt and from the c:\http directory type:

   `htpasswd -m conf\admin.passwd <userid>`

   Type a password, and confirm it.
9. Configure the ServerName of the Web server at this point. (Or, using the latest (1.3.6) version of HTTP Server, configure the Web server at a later time through the Administration console.) Using Notepad or WordPad, edit $c:\http\conf\httpd.conf$.

Add the line:  \texttt{ServerName yourhost.ibm.com}

Replace \texttt{yourhost.ibm.com} with the domain name of your machine. If you don’t know your host name, type \texttt{hostname} on a command prompt. Don’t reboot yet, because you are going to install more software.

\section*{A.5 Install DB2 UDB 6.1 and Fix pack}

This installation consists of two steps: first, installing DB2; second, installing the Fix pack.

\subsection*{A.5.1 Installing DB2 UDB 6.1}

1. If you have DB2 as a zip file from the Web site, unzip the DB2 zip file to a temporary directory, using the “Use folder names” option, so that it recreates the directory structure. Otherwise, insert the DB2 installation CD.
2. Run `setup.exe` file from the directory. Select **Next**, choose the Workgroup edition and then select **Next** (see Figure 55.)

![Select Products](image.png)

**Figure 55. Selecting installation options**

3. Choose the **Typical** installation, leave the default installation directory (`C:\SQLLIB`), and click **Next** again.
4. Overwrite the default user ID/password with the same user ID you chose previously. This will make it the DB2 administration ID as well (Figure 56).

![Figure 56. Setting the DB2 userid and password](image)

5. Select **Next** then **Next** to confirm the installation.

6. When the installation completes, select **Restart now**, and let the system reboot.

7. After restarting, the “DB2 First Steps” should start. In the “First Steps,” select **Create the SAMPLE database**. As shown in Figure 57, you will be prompted:

![Figure 57. DB2 First Steps, creating the sample database](image)
8. Select Yes; a sample database is created. Exit First Steps.

A.5.2 Installing the Fix pack

9. Before installing the Fix pack, stop the DB2 services. (See Figure 58.) From the services panel of the Control Panel, stop the services:
   DB2 - DB2
   DB2 - DB2DAS00
   DB2 Governor
   DB2 JDBC Applet Server
   DB2 JDBC Applet Server - Control Center
   DB2 Security Server

   ![Stopping DB2 services in the Control Panel](image)

   Figure 58. Stopping DB2 services in the Control Panel

   **Tip**

   If you have the Netfinity Support Manager or other monitoring agents, you may need to stop these as well, since they can lock DB2.

10. Unzip the Fix pack zip file in a temporary directory. (You may delete the DB2 install directory if you unzipped it previously.)

11. Run setup.exe from the Fix pack temporary directory. Click Next, Next and Next again. The installation should start.

12. Select Yes to restart your computer. After this, you will test the installation so far, and then install WebSphere.
A.6 Test the installation so far

1. After rebooting, login as the user you have been using to configure the software.

2. Start up a browser and browse http://localhost. The screen should look similar to Figure 59.

![Figure 59. Initial screen of the IBM HTTP Server](image)

3. Select Configure the Server and you will be prompted for the username and password that you entered in the `htpasswd` command. See Figure 60.
Figure 60. Entering a password for the IBM HTTP Server configuration page
4. When you enter these, you should see the configuration page for the Web server. It looks similar to the page shown in Figure 61.

![Figure 61. Initial page of the IBM HTTP Administration Server](image)

5. Use the basic settings to check the server name parameter you set earlier.

Before continuing, read the Tip documented in Figure 62.
Figure 62. Restart button on the IBM HTTP AdminServer

6. You have now tested and configured DB2 and HTTP Server. You can also check that you have the right JDK installed and configured by starting a command prompt and typing:

```
java -fullversion
```

You should see the response:

```
java full version "JDK 1.1.7 IBM build n117p-19990823 (JIT enabled: ibmjitc)
```

If you don’t, check that c:\jdk1.1.7 has been added to your path.

### A.7 Installing WebSphere Application Server

1. Start the installation by executing the given installation executable. This will start an installation shield package.

#### Note

You will need 30-50 MB free in your temp directory (usually on the C drive), even if you are installing on another drive, because the installshield package unpacks to the temp directory.

2. Click **Next** to pass the initial page.
3. Choose the **Development Kit (Full Installation)** which will install everything you need for Standard Edition.

---

**Advanced Edition Note**

If you are installing **Advanced Edition**:

Choose the **Custom Installation**. Use the installation directory as suggested. Click **Next**, and you are prompted with the options screen.

Now select the **IBM HTTP Server V1.3.6** on the right-hand side. You can install the WebSphere plug-ins (which connect the Web server to the servlet engine) into multiple Web servers at once. However, in this example we will install the plug-ins into the recently installed IBM HTTP Server.

Select the bottom option: **Configure administrative domain with default**......

This step is important, because the Advanced installation does not install the sample servlets or configure an EJB container or servlet engine. This is because an Advanced installation will often be into an existing administrative domain, whereas a Standard installation will always be standalone.

Click **Next**.
It should have found the IBM JDK that was installed. If you have other JDKs installed, make sure you select the IBM JDK, and click Next (see Figure 63).

![Select Java Development Kit](image)

**Figure 63. Selecting the Java Development Kit**

4. You will now be prompted with the major configuration page, which configures the database access and user IDs. The database access is most important to WebSphere 3.0, because WebSphere 3.0 stores the configuration of the server in a set of specially defined tables in the database. (It actually uses Entity Beans using bean-managed persistence to manage the server configuration). Although this page is called the security/database options, the server is installed with full security turned off. Fill in the screen with the following settings:

- **Security — User**: `xmlguru`
- **Security — Password (and Confirm)**: `<your chosen password>`
- **Database — Type**: `DB2`
- **Database — Name**: `was`
- **Database — User**: `xmlguru`
- **Database — Password**: `<your chosen password>`
The details are shown in Figure 64.

Figure 64. Security and database settings for WebSphere Application Server

**Note**

*If you use an invalid user ID to install WebSphere, it will not successfully register the WebSphere Admin Server to the NT services database.*

5. Click **Next**, and **Next** again to begin the installation.
6. Near the end of the installation, you are prompted for the location of the IBM HTTP Server configuration file — and it should successfully find it for you, so you should see the information shown in Figure 65:

![Figure 65. Selecting the Web server's configuration path](image)

7. Click OK, and it will finish updating the files and installing.

   The WebSphere Application Server release notes are always available online at [http://www.ibm.com/software/webservers/appserv/library.html](http://www.ibm.com/software/webservers/appserv/library.html)

   Click Finish, and choose to restart later, as we will do more configurations first.

**Configuring a database for the admin configuration**

Because this is the first installation, create the database to store the configuration. If you recall, it was defined by its JDBC URL as `jdbc:db2:was`.

We will create the `was` database, and set the DB2 application heap size for it.

   a. From the Start menu, choose:

      Programs->DB2 for Windows NT->Command Line Processor

   b. Now type:

      `CREATE DATABASE WAS`

   c. Wait a while, then type:

      `UPDATE DB CFG FOR WAS USING APPLHEAPSZ 256`
This should be instantaneous. Type **quit** to leave the CLP, and then **exit** to finish the command prompt.

8. Restart the machine.

---

**Note**

If you wish to re-install WebSphere, you can uninstall WebSphere, delete the **was** database, re-install WebSphere, and then redo these steps.

---

**A.8 Testing your installation**

1. When you have logged back in, check on the installation. First, check that WebSphere Admin Server has been registered in the Services panel. Open the Control Panel, select **Services**. If you scroll down you should see: IBM WS AdminServer as shown in Figure 66.

![Services panel showing the IBM WS AdminServer entry](image)

*Figure 66. Services panel showing the IBM WS AdminServer entry*

2. You can now start it from here. Wait patiently!

---

**Tip**

You can control the service from a command line or batch file using the following commands:

```
net start "IBM WS AdminServer"
net stop "IBM WS AdminServer"
```
If it does not start successfully, you can see the logs in:

```bash
c:\WebSphere\AppServer\logs\tracefile
```

and

```bash
c:\WebSphere\AppServer\logs\nanny.trace
```

(You may be wondering what the “nanny” is. It is the process that monitors the AdminServer and restarts it if anything goes wrong.)

---

### Explaining the AdminServer

The AdminServer doesn’t run any servlets, Java Server Pages, or Enterprise Beans. Each node can have multiple JVMs known as *servers* which run the application code. The *AdminServer* manages these servers, and if they stop, it restarts them. Each *server* can have *EJB Containers* to run EJBs, and a *servlet engine* to run servlets. Servers can either be independent, running different code, or *clones* which are identical, and the infrastructure manages spreading the incoming requests across the servers. A *cluster* consists of multiple nodes, each with clones.

Although the installation has created a default server, it hasn’t started it, so you can’t run anything yet.

3. To start the server, run the Admin Console, which is how you manage WebSphere 3.0. From the Start menu, select

```bash
Programs->IBM WebSphere->Application Server 3.0->Administrator’s Console
```
This will start a new command shell, followed by a window showing the startup screen. You should see a display like that shown in Figure 67:

Figure 67. WebSphere Standard Administrative Console

4. Wait until the little icon in the bottom left corner stops rotating. Then start to administer the server:
   a. The basic administration console is split into three tabs — Tasks, Types, and Topology. Click on the Topology tab, and then click on the + sign next to “WebSphere Admin Domain” to expand the view.
   b. Your host name should be listed. Expand the view of that node, and you should see an entry called “Default Server.” Expand that, and you will see the default container and servletEngine.
c. Right-click on the Default Server, and select **Start**.
A blue visual indicator on the bottom right of the screen shows you something is happening. When the server is started, the dialog shown in Figure 68 is displayed.

![Information dialog](image)

**Figure 68. Server started dialog**
d. Click **OK**.

---

**Note**

Once the server is started, it is marked in the config database that it should be running. If it stops, or if you reboot the machine, the AdminServer will automatically restart it. Even if the AdminServer fails, it will continue to run.
5. Let us test it. Open a browser and go to \texttt{http://localhost/servlet/snoop}, which is a standard sample servlet and has been installed by default. You should see a result similar to that shown in Figure 69.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{snoop.png}
\caption{Sample servlet: snoop}
\end{figure}

\textbf{Requested URL:}

http://localhost/servlet/snoop

\textbf{Init Parameters}

\texttt{param1: test-value1}

\textbf{Request information:}

Request method: GET
Request URL: /servlet/snoop

Congratulations!!!

That completes the installation and testing of WebSphere Application Server Standard Edition. If you have installed Advanced Edition, you will wish to configure and test an Enterprise Bean. That is detailed in A.10, “Configuring and testing an Enterprise Bean” on page 191.

\section*{A.9 Configuring a laptop/mobile machine with a constant IP address}

When running WebSphere Application Server, it is important that your hostname remains fixed. WebSphere Advanced version 3.0 is a completely "networked" system capable of being clustered into a single domain, and the administration and other aspects rely upon the IP networking. When
WebSphere starts for the first time, it records the hostname. When it restarts, it needs to be able to contact that IP address again.

If you wish to use WebSphere "disconnected", you must always be able to "ping" your hostname.

If you have a desktop machine with a fixed IP address, you can skip this section!

There are other ways of doing this. However, this way works:
1. In the Network section of the Control Panel, in the Adapters tab, install the "MS Loopback Adapter" (see Figure 70).

![Network Configuration](image)

*Figure 70. Adding the MS Loopback Adapter to Windows NT networking*
2. In the Protocols tab, open TCP/IP properties, and give it a fixed TCP/IP address of 10.0.0.1 (which is reserved for private use), and a subnet mask of 255.255.255.0. (See Figure 71.)

![Microsoft TCP/IP Properties](image)

Figure 71. Configuring a TCP/IP address for the Loopback Adapter

No default gateway for this adapter was specified.
3. Go to the Bindings tab, select all protocols, expand TCP/IP Protocol, and then move the MS Loopback Adapter to the top. (See Figure 72.)

![Network Configuration](image)

*Figure 72. Ordering the adapters so that the Loopback Adapter is first*

4. Once you reboot, you can ping your hostname whether or not you are network connected.

**A.10 Configuring and testing an Enterprise Bean**

*This section only applies to WebSphere Advanced Edition. It is not required for the XMLapp application used in this redbook.*
Aims: to test an Enterprise Bean inside WebSphere 3.0 Advanced Edition, configure one of the samples, and use a sample servlet to test it. The samples are documented, at:
Start->Programs->IBM WebSphere->Application Server v3.0->Samples
Configure and run all the samples once you have finished this initial installation and test.

This exercise uses the Increment Bean. This bean is very similar to the Inc bean provided with WebSphere v2.0; it is simply a counter.

A.10.1 Setting up a DataSource

Because this bean is an Entity Bean, it will require a database to store the values of the bean. WebSphere 3.0 uses a concept from JDBC 2.0 to implement access to persistent stores (for example, databases), although WS3.0 is not yet JDBC2.0 compliant. This is called a DataSource.

Data Sources

DataSources are important for two reasons:

1. DataSources implement connection pooling to databases, so WebSphere can re-use database connections.

2. For BMP or session beans, the DataSource creates a model where the bean can ask the application server for a connection. This gives the application server the hook it needs to grab the XA resource manager handle so that it can become the transaction co-ordinator for the transaction.

The way this works is by defining a JDBCDriver to the Application Server, then a particular database as a DataSource. This database is given a local name, which will be made available in the JNDI name space. Servlets and BMP or Session Beans that want to access the DataSource can retrieve it using a JNDI lookup, and then get a standard JDBC connection from the DataSource. This is documented under Connection Pooling in the Help.

The DataSource is the object that applications will be configured to use. WebSphere 3.0 separates the DataSource from the JDBC driver. This is important, because you may want to reconfigure a DataSource to use an alternative driver (for example, one that supports two-phase commits for transactions.)

1. To configure a datasource, go to the Types tab and right-click on JDBCDrivers. (See Figure 73.)
2. Select Create. In the name area, enter DB2AppDriver, and in the Implementation class select: COM.ibm.db2.jdbc.app.DB2Driver. Leave the URL prefix as jdbc:db2 and JTA enabled as false. DB2 UDB 6.1 FP1a does support JTA and two-phase commit under WebSphere. However, it is unnecessary at this stage. (See Figure 74.)

3. Click Create. You should see the message shown in Figure 75.
4. Right-click on **DataSources** and select **Create**. You will be prompted with a dialog. Enter “sampledb” as the Name, and “sample” as the Database Name. (See Figure 76.)

**Figure 76. Creating a new DataSource**

The first entry — **Name** — is used to create the JNDI directory entry for the database. For CMP Enterprise Beans (like Increment), the server will automatically pick this up. For servlets and non-CMP beans, this will make a JNDI name of “jdbc/sampledb.” See the connection pool sample program for a programming example.

The second entry — **Database Name** — is the name of the database, and is concatenated with the URL prefix field (see above) to make the full JDBC URL of the database. The name in this example, **sample**, was created when you installed DB2. The last field is the Driver, and **DB2AppDriver** is the only choice, because it is the only JDBCDriver defined.
5. Click on the **Advanced** tab. You will see the connection pool parameters, which control the performance of the pool. (See Figure 77.)

![Create an DataSource](image)

**Figure 77. Advanced properties of the DataSource**

The minimum and maximum connection pool sizes are clear. The Connection timeout (default 300 seconds = 5 minutes) means that if the pool reaches the maximum, new connections will wait this long for a new connection before timing out.

The Idle timeout of 1800 seconds (30 minutes) means that if connections in the pool are unused for 30 minutes, they will be closed down to the minimum. The orphan timeout field indicates how long a connection can remain allocated before the server decides that the owner thread has died and the connection should be recovered.

6. Accept the defaults by clicking on **Create**.

If you now click on the Topology view, you should see the **sample** and **DB2Driver** objects. (See Figure 78.)

![Topology view of the DataSource and JDBCDriver objects](image)

**Figure 78. Topology view of the DataSource and JDBCDriver objects**
If you don’t see these objects, try refreshing the view — select the **WebSphere Admin Domain** and click on the refresh button. (See Figure 79.)

![Refresh Button](image)

*Figure 79. The refresh button*

7. The last step in creating the DataSource is to Install the driver — right-click on the **DB2AppDriver** object and select **Install**. (See Figure 80.).

![Install Driver](image)

*Figure 80. Installing the Driver*

Because you could be administering a cluster (but aren’t yet), first select the node on which to install — in this case there will only be one — and then click browse:

You are browsing for the JDBC driver JAR/ZIP file which contains the JDBC driver for this database. For DB2 and the standard installation, this is **C:\SQLLIB\JAVA\DB2JAVA.ZIP**. Select this and click **Install**.
A.10.2 Deploying and installing the Enterprise Bean

1. In the AdminConsole, and from the Topology view, make sure the node (yourhost), and Default Server is expanded. Right-click (see the note that follows) on the Default Container entry: choose Create->EnterpriseBean.

Note

Be aware that the right-click behavior is different when the object is running. To prevent “accidents”, you need to hold down the right mouse button while you browse the pop-up menu, and then let go on the selection you want to make. This means a random click won’t impact the server.

You will be prompted with a dialog as shown Figure 81.

![Deploy Enterprise Bean dialog](image_url)

Figure 81. Deploy Enterprise Bean dialog

2. Click on Browse and navigate to C:\WebSphere\AppServer\deployableEJBs\ (or the directory where it is installed). You are browsing the server directories here, so even if you were using the console on a remote machine, this would be the server directory you see.
Select the **Increment.jar** and double-click on it. You should be able to see “inside” the JAR file now, as shown in Figure 82.

![Figure 82. Browsing for EJB JAR files and Enterprise Beans](image)

3. Double-click on the **.ser** file to select it. You will be prompted as shown in Figure 83.

![Figure 83. Deployment dialog](image)

---

**Note**

Each EJB-JAR file can contain more than one EJB. WebSphere 3.0 includes a EJB-JAR browser, and this example uses an individual bean to deploy.
This will kick off the process to “deploy” the bean. In WebSphere terminology, this is the part of the deployment process that generates the code to actually run the enterprise bean in the server. This includes code to persist the state (for an entity bean), manage transactions, and also the RMI-IIOP stubs and ties.

4. Click **Yes** and the window shown in Figure 84 will be displayed:

![Enterprise bean is deploying. Property sheet will be populated after deployment is finished.](image)

*Figure 84. Bean deployment notice*

Be patient — WebSphere is generating code as well as compiling it!

When it finishes, you will see the message shown in Figure 85.

![Information dialog](image)

*Figure 85. Deployment completed dialog*
5. Click **OK** and you will be presented with the Bean dialog again, but now with the fields filled in, as shown in Figure 86.

![Deploy Bean dialog after selecting a Bean](image)

**Figure 86. Deploy Bean dialog after selecting a Bean**

**Note**

The Database Access field (greyed out) is set to Shared, which is equivalent to Option C locking/caching, and supports clustering.
You can now look at the deployment descriptor by clicking the **Edit** button. (See Figure 87.)

<table>
<thead>
<tr>
<th>Deployment Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
</tr>
<tr>
<td><strong>Entity</strong></td>
</tr>
<tr>
<td><strong>Session</strong></td>
</tr>
<tr>
<td><strong>Security</strong></td>
</tr>
<tr>
<td><strong>Transactions</strong></td>
</tr>
<tr>
<td><strong>Isolation</strong></td>
</tr>
<tr>
<td><strong>Environment</strong></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
</tr>
<tr>
<td><strong>Class Names</strong></td>
</tr>
<tr>
<td><strong>Enterprise Bean Class</strong></td>
</tr>
<tr>
<td><strong>Home Interface Class</strong></td>
</tr>
<tr>
<td><strong>Remote Interface Class</strong></td>
</tr>
<tr>
<td><strong>JNDI</strong></td>
</tr>
<tr>
<td>JNDI Home Name</td>
</tr>
<tr>
<td>Increment</td>
</tr>
<tr>
<td>Set</td>
</tr>
<tr>
<td>OK</td>
</tr>
</tbody>
</table>

**Figure 87. Viewing the Deployment Descriptor inside WebSphere Admin Console**

Only the JNDI name and the environment variables are editable here. This is because updating other properties requires re-generating the code. To update other properties, edit the deployment descriptor using VisualAge for Java Enterprise Edition v3.0, or use the standalone EJB tool, which is included with WebSphere — **jetace**. Don’t change any properties now.

6. Once you have finished viewing the DD, click **OK** to return to the first dialog, and select the DataSource tab. Click **Change** and select the **sample** DataSource. (See Figure 88.)
7. Click **OK**.

   Because WebSphere is running under the user ID that has full rights to DB2, you can leave the user ID and password blank. Alternatively, you can enter a valid user ID and password with access to this database.

8. Finally, make sure that the **Create Table** check box is selected — this will use the persistence engine to create a default “top-down” object-relational mapping from the EJB, and use that to create a table in the database to store the container-managed properties of the bean.
The information shown in Figure 89 is displayed.

![Figure 89. Setting the Advanced properties of a bean](image)

This defines the pool size for the bean. In general, WebSphere tries to optimize Enterprise Bean access by re-using existing beans where the semantics of the EJB specification allow it. Note that the maximum is **not** a hard maximum, but the maximum number of instances that will be held alive.

9. Click **Create**.

The dialog shown in Figure 90 is displayed.

![Figure 90. Successful creation of an Enterprise Bean](image)

10. Click **OK** to clear the dialog.

11. The last step is to start the bean. One of the nice features of WebSphere is that you can deploy a new Enterprise Bean into an already running container. To start the bean, right-click on the **Increment** object in the topology view and select **Start**.
12. To test this, open the samples page:
   a. Start->Programs->IBM WebSphere->Application Server 3.0->Samples
   b. Click on Enterprise Beans in the left frame, and then choose Increment from the right frame.
   c. Select Run this Sample, and you should see the display shown in Figure 91.

![Figure 91. WebSphere Increment sample](image-url)
13. Click **Visit**, and information on the number of visits should be displayed, as shown in Figure 92.

![Figure 92. Results of using the Increment Bean sample](image)

Keep clicking on **Visit**, and the number of visits will increase by one each time. **Congratulations!** You have installed and configured an EJB application.

14. To see what is going on in the database, start up the DB2 Command Line Processor:
   a. **Start->DB2 for Windows NT->Command Line Processor**
   b. and type:
      
      ```
      CONNECT TO SAMPLE
      SELECT * FROM EJB.INCREMENTBEANTBL
      ```
      
Data similar to that shown in Figure 93 is displayed:
A.10.3 Summary

If you got this far, you have:

1. Installed and configured the support software (IBM HTTP Server 1.3.6, DB2 6.1 FP 2, IBM JDK1.1.7p 0823.
2. Installed and configured a single node setup for WebSphere 3.0.
3. Started the default server.
4. Tested a servlet, and therefore the link between IHS1.3.6 and WSAS3.0.
5. Created a new DataSource.
6. Deployed an EnterpriseBean.
7. Tested the EnterpriseBean and container-managed persistence.

Well Done!
Appendix B. ITSO XML Application demo installation

In this section we describe the necessary installation and configuration steps that have to be done in order to run the ITSO XML Application. It is assumed that the basic infrastructure — WebSphere, DB2, and IBM HTTP Server — are installed and functional. The installation of these components is described in Appendix A, “WebSphere Installation” on page 165.

---

Sample Code on the Internet

The sample code for this redbook is available as the SG245479.zip file on:
ftp://www.redbooks.ibm.com/redbooks/SG245479/

Download sg245479.zip and read the README.TXT file included in the zip file. Any updates to the book will also be found there.

---

B.1 Installing the databases

The ITSO XML Application requires two databases (in addition to the WAS database that is required by WebSphere run-time). The application database is called A118, and the personalization database is called PROFILES.

---

Important

The components of the ITSO XML Application use a hard-coded userid/password combination for database access. To avoid running into trouble, we suggest that the first thing you should do is to create a new user for your system. The userid is xmlguru, and the password is xmlguru, both in lowercase. If you do not want to set the database user privileges for this user using the DB2 utilities, we suggest that you make this user an administrative one.

This approach is suitable for demonstration purposes only, and not recommended for production environments.

The personalization database is created manually by using the DB2 Command Window and entering the command:

db2 create db profiles

---
The A118 database definitions and sample data is shipped in the A118ddl.zip file, which includes three DDL scripts. File xmlapp.ddl contains the necessary definition for creating the database and the tables, and file data.ddl contains the sample data. You should run this script by using the DB2 Command Window in this order:

```
db2 -f xmlapp.ddl -t
db2 -f data.ddl -t
db2 -f sup001.ddl -f
```

You need to install the jdbc driver for DB2 before defining the datasources for WebSphere. In the Types view, select JDBCDrivers; and in the pop-up menu, select Create. Let's call it DB2Driver and use the implementation class com.ibm.db2.jdbc.app.DB2Driver. See Figure 76 on page 194 for details.

After the driver for DB2 has been defined, you can define the datasources. The datasources to be defined are A118, PROFILES, and SUP001. The datasources are added in the Types view by selecting Datasources->Create. See Figure 18 on page 98 for details.

Next, you have to install the driver to your host. In the Topology page of WebSphere Administration Console, select the DB2Driver object, and in the pop-up menu, select Install. In the window that is displayed, select your host name and the jar file containing DB2 JDBC classes (see Figure 94).

![Figure 94. Installing the DB2Driver](image)

Now you can use the datasources in your Web applications.
B.2 Configuring WebSphere Application Server 3.0

Several things have to be done in order to be able to run the ITSO XML Application successfully. You have to obtain updates to WebSphere Application Server 3.0, and it has to be configured properly. Only after these steps have been completed, can the application files be installed and run successfully.

B.2.1 Updates to common classes

The updates you need to have are:

- LotusXSL update
- Bean Scripting Framework (optional XSLT Islands support)

These updates can be found at http://www.alphaworks.ibm.com.

LotusXSL update is needed for DefaultApplyXSL servlet support, since the version of lotusxsl.jar that comes with WebSphere Application Server does not contain this servlet. You can either download the lotusxsl.jar file from the IBM Alphaworks Web site and replace the file in the WebSphere lib directory, or just place these missing classes in a place where WebSphere find them. We decided to replace the jar file.

If you want to install new classes that are shared by all IBM WebSphere Application Server applications, you have two options when deciding where to put them. One option is to place all Java class files (not jar or zip files) in the \WebSphere\Appserver\classes directory, with appropriate subdirectories required by packages. If you have a zip or a jar file, the jar (or zip) file containing the classes can be included in the startup parameter list of WebSphere. See Figure 20 on page 100 for details.

If you want to test the XSLT Islands technology (not needed for running the application when using the DefaultApplyXSL servlet), you need to download the Bean Scripting Framework implementation from Alphaworks. File wasbsf.jar contains BSF 2.0 and other patches to WebSphere Application Server.

Note!

These instructions are for WebSphere Application Server 3.0. Some of these updates (especially BSF) are likely to be included in FixPaks to version 3.0. If you have installed a FixPak for WebSphere, you should check if you really need to install the updates.
The XML Files: Using XML and XSL with IBM WebSphere 3.0

Server v3.0. This file has to be included in the command line arguments classpath of WebSphere.

With all requirements for the ITSO XML Application, the whole startup string in our test environment was:

```
-mx128 -classpath c:\jars\wasbsf.jar;c:\jars\pwup.jar;
```

To set the startup string, go to the Topology page, then select Default Server.

**B.2.2 Application definition and DefaultApplyXSL**

WebSphere Application Server uses the notion of a Web Application to separate applications with different requirements from each other. It is possible to use the default application of WebSphere, but we decided to create a new application, because some changes are needed in the application configuration. Using the WebSphere Administration Console, in the Topology page, expand the WebSphereAdminDomain tree until you can see the servletEngine. Select that, and in the pop-up menu select Create->Web Application. Fill in the values as shown in Figure 95. Be sure to remove the default /webapp portion of the Web application path. Click on Create.

![Figure 95. Web application definition](image.png)

The next thing to do is to define the DefaultApplyXSL servlet. In the Topology view, select your newly created application, and in the pop-up menu select Create->Servlet. Fill in the values as shown in Figure 96.
After you have created the DAXSL servlet, you need to set up its properties. In the Topology view, select the DAXSL servlet, and enter the values as shown in Figure 97.

Now select the xmlapp Web application, and define the MIME-filter for text/xml content type as shown in Figure 98.
Because parts of the ITSO XML Application are implemented as Java Server Pages, you need to install a JSP enabler for the xmlapp application. In the Tasks page, select Add a JSP enabler, and click the Start Task button (the green button in the toolbar). Set the values as shown in Figure 99.
B.2.3 User Profiles

You have to enable the user profile services, and specify the PROFILES database as the datasource for user profiles.

In the Topology view select User Profile Manager. In the Enable page set Enable User Profile: to yes. In the Datasource page set the Datasource: to PROFILES and use xmlguru for userid and password. In the Classes page update the Data Wrapper class to itso.xmlapp.util.PwUserProfile.

Detailed instructions are provided in “Logon and user profile management” on page 97. See Figure 100.

![WebSphere Standard Administrative Console](image)

Figure 100. WebSphere Standard Administrative Console

B.2.4 The supplier application

You need to set up another Web application called supplier. This application needs the DefaultApplyXSL servlet support, JSP 1.0 Support and support for serving servlets by classname (Auto-Invoker). See “Application definition and DefaultApplyXSL” on page 210 for instructions on how to set up these items.

This part of the application does not rely on named servlets, so you need to enable servicing of servlets by using their fully qualified names. In the Tasks view, select **Add a Servlet** and click **Start Task** button. Click the Next button until you can select the application (supplier), and finally, make the selection as shown in Figure 101.
B.3 Installing the application files

The following table shows the names of the zip files and where you should unpack them in order to be able to run the application.

<table>
<thead>
<tr>
<th>Zip file</th>
<th>Where to unpack</th>
</tr>
</thead>
<tbody>
<tr>
<td>A118was.zip</td>
<td>/WebSphere/Appserver/</td>
</tr>
<tr>
<td>A118http.zip</td>
<td>/IBM HTTP Server/htdocs (or any other document root)</td>
</tr>
<tr>
<td>A118xmlapp.zip</td>
<td>C:/ (root of the C: drive)</td>
</tr>
<tr>
<td>A118jar.zip</td>
<td>c:/jars</td>
</tr>
</tbody>
</table>

A118was.zip contains all servlets, JSPs and support classes needed by the ITSO XML Application.

A118http.zip includes the static XML, XSL and DTD files.

A118xmlapp.zip includes the XML template documents needed by the database interface.

A118jar.zip includes the jar files we used in the application (so there is no need for downloading the updates).
The next thing to do is to name the servlets, because, for the main application part, we do not use fully qualified names when calling servlets. The supplier part uses the fully qualified names, so we do not need to name them (but we need to use the Auto-Invoker instead).

**Note!**

It is always a good idea to use named servlets. When the servlets are named, your servlet package name or class name can be changed without changes in the applications that are using the servlet. Another reason is security: only servlets with names can be invoked, and you can use the security features of WebSphere Application Server to protect named servlets.

You have already created a named servlet called DAXSL (see Figure 96 on page 211). Using the same procedure, create four more servlets in the xmlapp application according to Table 7:

**Table 7. Defining servlets used in xmlapp**

<table>
<thead>
<tr>
<th>Name</th>
<th>Fully qualified name</th>
<th>Servlet Web Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogonServlet</td>
<td>itso.xmlapp.servlets.LogonServlet</td>
<td>/xmlapp/servlet/LogonServlet</td>
</tr>
<tr>
<td>RegisterServlet</td>
<td>itso.xmlapp.servlets.RegisterServlet</td>
<td>/xmlapp/servlet/RegisterServlet</td>
</tr>
<tr>
<td>BrowseOrderServlet</td>
<td>itso.xmlapp.servlets.BrowseOrderServlet</td>
<td>/xmlapp/servlet/BrowseOrderServlet</td>
</tr>
<tr>
<td>RegChange</td>
<td>itso.xmlapp.servlets.RegisterChangeServlet</td>
<td>/xmlapp/servlet/RegisterChangeServlet</td>
</tr>
</tbody>
</table>

It is always a good idea to use named servlets. When the servlets are named, your servlet package name or class name can be changed without changes in the applications that are using the servlet. Another reason is security: only servlets with names can be invoked, and you can use the security features of WebSphere Application Server to protect named servlets.

B.4 Optional XSLT Island

Changing the user registration information is implemented by two JSP files and a servlet. The JSPs have be installed in a separate application that has not DefaultApplyXSL servlet installed and configured as a mime-filter. The only thing you have to do is to create a new Web application called *island*, with the default Web path *island*. The JSPs and the servlet have already been installed. You need to add a JSP 1.0 enabler for the application.
B.5 Running the application

To start using the application:

**Customer:**  http://server/xmlapp/xml/MainMenu.xml

**Supplier:**  http://server/supplier/SupMainMenu.jsp
Appendix C. Data Definition Languages (DDLs)

This section contains the Data Definition Language (DDL) files used to create each table in the application database. The DDL specifies the table design as well as database constraints associated with each table, such as primary and foreign keys.

C.1 SUPPLIERS table

------------------------------------------------
-- DDL Statements for table "XMLGURU "."SUPPLIERS"
------------------------------------------------

CREATE TABLE "XMLGURU "."SUPPLIERS" (  
"SUPPLIER_ID" CHAR(10) NOT NULL ,  
"SUPPLIER_NAME" VARCHAR(50) ,  
"PHONE" VARCHAR(50) ,  
"EMAIL_ADDRESS" VARCHAR(50) ,  
"STREET_ADDRESS" VARCHAR(50) ,  
"CITY" VARCHAR(50) ,  
"STATE" VARCHAR(50) ,  
"ZIP" VARCHAR(50) ,  
"COUNTRY" VARCHAR(50) ,  
"XSL_URL" VARCHAR(255) ,  
"B2B_URL" VARCHAR(255) )  
in "USERSPACE1" ;
ALTER TABLE "XMLGURU "."SUPPLIERS" PCTFREE 0;

-- DDL Statements for primary keys on Table "XMLGURU "."SUPPLIERS"
ALTER TABLE "XMLGURU "."SUPPLIERS"  
ADD PRIMARY KEY  
("SUPPLIER_ID") ;

C.2 PRODUCTS table

-- DDL Statements for table "XMLGURU "."PRODUCTS"
CREATE TABLE "XMLGURU "."PRODUCTS" (  
"PRODUCT_ID" CHAR(10) NOT NULL ,  
"PRODUCT_NAME" VARCHAR(50) ,  
"DESCRIPTION" VARCHAR(80) ,  
"SUPPLIER_ID" CHAR(10) NOT NULL ,  
"SUPPLIER_PART_NO" CHAR(10) NOT NULL ,  
"UNIT_PRICE" DECIMAL(10,2) ,  
"UNITS_IN_STOCK" SMALLINT NOT NULL WITH DEFAULT 0,  
"MINIMUM_STOCK_LEVEL" SMALLINT NOT NULL WITH DEFAULT 0)  
in "USERSPACE1" ;
ALTER TABLE "XMLGURU "."PRODUCTS" PCTFREE 0;

-- DDL Statements for primary keys on Table "XMLGURU "."PRODUCTS"
ALTER TABLE "XMLGURU "."PRODUCTS"
ADD PRIMARY KEY
("PRODUCT_ID");

-- DDL Statements for foreign keys on Table "XMLGURU "."PRODUCTS"
ALTER TABLE "XMLGURU "."PRODUCTS"
ADD CONSTRAINT "SQL990922110107360" FOREIGN KEY
("SUPPLIER_ID")
REFERENCES "XMLGURU "."SUPPLIERS"
("SUPPLIER_ID")
ON DELETE RESTRICT
ON UPDATE NO ACTION;

C.3 ORDER_HEADERS table

CREATE TABLE "XMLGURU "."ORDER_HEADERS"
(
"ORDER_NO" VARCHAR(30) NOT NULL ,
"CUSTOMER_ORDER_NO" CHAR(35) ,
"ORDER_DATE" DATE ,
"SHIP_DATE" DATE ,
"USERNAME" VARCHAR(64) )
IN "USERSPACE1" ;

ALTER TABLE "XMLGURU "."ORDER_HEADERS" PCTFREE 0;

-- DDL Statements for primary keys on Table "XMLGURU "."ORDER_HEADERS"
ALTER TABLE "XMLGURU "."ORDER_HEADERS"
ADD PRIMARY KEY
("ORDER_NO");

C.4 ORDER_ITEMS table

CREATE TABLE "XMLGURU "."ORDER_ITEMS"
(
"ORDER_NO" VARCHAR(30) NOT NULL ,
"PRODUCT_ID" CHAR(10) NOT NULL ,
"ITEM_PRICE" DECIMAL(10,2) ,
"ITEM_STATUS" VARCHAR(30) ,
"ITEM_NO" SMALLINT NOT NULL ,
"ORDER_QUANTITY" SMALLINT ,
"SHIP_DATE" DATE )
IN "USERSPACE1" ;

ALTER TABLE "XMLGURU "."ORDER_ITEMS" PCTFREE 0;

-- DDL Statements for primary keys on Table "XMLGURU "."ORDER_ITEMS"
ALTER TABLE "XMLGURU "."ORDER_ITEMS"
ADD PRIMARY KEY
    ("ORDER_NO", "ITEM_NO");

-- DDL Statements for foreign keys on Table "XMLGURU "."ORDER_ITEMS"

ALTER TABLE "XMLGURU "."ORDER_ITEMS"
    ADD CONSTRAINT "SQL990922105918130" FOREIGN KEY
        ("ORDER_NO")
    REFERENCES "XMLGURU "."ORDER_HEADERS"
        ("ORDER_NO")
    ON DELETE RESTRICT
    ON UPDATE NO ACTION;

ALTER TABLE "XMLGURU "."ORDER_ITEMS"
    ADD CONSTRAINT "SQL990922110439660" FOREIGN KEY
        ("PRODUCT_ID")
    REFERENCES "XMLGURU "."PRODUCTS"
        ("PRODUCT_ID")
    ON DELETE RESTRICT
    ON UPDATE NO ACTION;
Appendix D. Special notices

This publication is intended to help Web developers to take advantage of XML technology and develop XML Applications under IBM WebSphere Application Server. The information in this publication is not intended as the specification of any programming interfaces that are provided by the IBM WebSphere Application Server. See the PUBLICATIONS section of the IBM Programming Announcement for IBM WebSphere Application Server for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee
that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

<table>
<thead>
<tr>
<th>IBM</th>
<th>AS400</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>CICS</td>
</tr>
<tr>
<td>CT</td>
<td>DB2</td>
</tr>
<tr>
<td>MQSeries</td>
<td>Netfinity</td>
</tr>
<tr>
<td>RS/6000</td>
<td>SP</td>
</tr>
<tr>
<td>System/390</td>
<td>VisualAge</td>
</tr>
<tr>
<td>WebSphere</td>
<td>XT</td>
</tr>
<tr>
<td>400</td>
<td></td>
</tr>
</tbody>
</table>

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET and the SET logo are trademarks owned by SET Secure Electronic Transaction LLC.
Other company, product, and service names may be trademarks or service marks of others.
Appendix E. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

E.1 International Technical Support Organization publications

For information on ordering these ITSO publications see “How to get IBM Redbooks” on page 229.

- *WebSphere Application Servers: Standard and Advanced Editions*, SG24-5460
- *IBM WebSphere and VisualAge for Java Database Integration with DB2, Oracle, and SQL Server*, SG24-5471
- *Developing an e-Business Application for the IBM WebSphere Application Server*, SG24-5423
- *Enterprise JavaBeans Development with VisualAge for Java*, SG24-5429
- *Programming with VisualAge for Java Version 2*, SG24-5264
- *IBM WebSphere Performance Pack Usage and Administration*, SG24-5233
- *Connecting the Enterprise to the Internet with MQSeries and Visual Age for Java*, SG24-2144
E.2 Redbooks on CD-ROMs

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at http://www.redbooks.ibm.com/ for information about all the CD-ROMs offered, updates and formats.

<table>
<thead>
<tr>
<th>CD-ROM Title</th>
<th>Collection Kit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>System/390 Redbooks Collection</td>
<td>SK2T-2177</td>
</tr>
<tr>
<td>Networking and Systems Management Redbooks Collection</td>
<td>SK2T-6022</td>
</tr>
<tr>
<td>Transaction Processing and Data Management Redbooks Collection</td>
<td>SK2T-8038</td>
</tr>
<tr>
<td>Lotus Redbooks Collection</td>
<td>SK2T-8039</td>
</tr>
<tr>
<td>Tivoli Redbooks Collection</td>
<td>SK2T-8044</td>
</tr>
<tr>
<td>AS/400 Redbooks Collection</td>
<td>SK2T-8046</td>
</tr>
<tr>
<td>Netfinity Hardware and Software Redbooks Collection</td>
<td>SK2T-8046</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (BkMgr)</td>
<td>SK2T-8040</td>
</tr>
<tr>
<td>RS/6000 Redbooks Collection (PDF Format)</td>
<td>SK2T-8043</td>
</tr>
<tr>
<td>Application Development Redbooks Collection</td>
<td>SK2T-8037</td>
</tr>
<tr>
<td>IBM Enterprise Storage and Systems Management Solutions</td>
<td>SK3T-3694</td>
</tr>
</tbody>
</table>

E.3 Other publications

These publications are also relevant as further information sources:

E.4 Web sites referenced in this book

http://www.alphaworks.ibm.com/tech/bsf
http://www.alphaworks.ibm.com/tech/LotusXSL
http://www.ibm.com/developer/xml
http://www.jclark.com/dsssl
http://www.megginson.com/SAX
http://www.netcraft.com/survey
http://www.oasis-open.org
http://www.schema.net
http://www.w3.org/DOM
http://www.w3.org/pub/W3C/TR/WD-xml-961114.html
http://www.w3.org/Style/CSS
http://www.w3.org/TR/WD-xsl
http://www.w3.org/TR/WD-xslt
http://www.w3.org/TR/xml-stylesheet
http://www.w3.org/TR/xmlschema-1
http://www.w3.org/TR/xmlschema-2
http://www.w3.org/XML
How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.


  Search for, view, download, or order hardcopy/CD-ROM redbooks from the redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this redbooks site.

  Redpieces are redbooks in progress; not all redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the redbooks fax order form to:

  **e-mail address**

  In United States  usib6fpl@ibmmail.com

  Outside North America  Contact information is in the “How to Order” section at this site:


- **Telephone Orders**

  United States (toll free)  1-800-879-2755

  Canada (toll free)  1-800-IBM-4YOU

  Outside North America  Country coordinator phone number is in the “How to Order” section at this site:


- **Fax Orders**

  United States (toll free)  1-800-445-9269

  Canada  1-403-267-4455

  Outside North America  Fax phone number is in the “How to Order” section at this site:


This information was current at the time of publication, but is continually subject to change. The latest information may be found at the redbooks Web site.

---

**IBM Intranet for Employees**

IBM employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at [http://w3.itso.ibm.com/](http://w3.itso.ibm.com/) and clicking the ITSO Mailing List button.

Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at [http://w3.ibm.com/](http://w3.ibm.com/) for redbook, residency, and workshop announcements.

© Copyright IBM Corp. 2000
IBM Redbook Fax Order Form

Please send me the following:

<table>
<thead>
<tr>
<th>Title</th>
<th>Order Number</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First name | Last name
Company
Address
City | Postal code | Country
Telephone number | Telefax number | VAT number
☐ Invoice to customer number
☐ Credit card number

Credit card expiration date | Card issued to | Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.
Glossary

**attribute** In XML, a name="value" pair that can be placed in the start tag of an element. The value must be quoted with single or double quotes.

**Bean Scripting Framework (BSF)** The Bean Scripting Framework is a technology that allows scripting languages such as JavaScript and Python to be used together with Java.

**Cascading Style Sheet (CSS)** CSS defines a stylesheet language for HTML 4.0. CSS allows a Web page designer to separately specify style elements of a Web page, such as colors, fonts and font styles.

**case-sensitive** Indicates whether an application, processor, or operating system distinguishes between upper and lower case. If it does, it is case-sensitive. XML tags are case-sensitive, but HTML tags are not.

**content model** In XML, the expression specifying what elements and data are allowed within an element.

**Document Object Model (DOM)** This allows the representation and manipulation of an XML document in memory as a programming object. DOM is defined by the World-Wide Web Consortium.

**DOM** (see Document Object Model).

**DOM Tree** A DOM Tree is an in-memory representation of an XML Document.

**Document Type Definition (DTD)** A DTD is a definition of which Elements and Attributes are acceptable in a specific XML file. The DTD therefore defines a subset of XML which may be used for a particular application.

**EBNF** Extended Backus-Naur Form. A formal set of production rules that comprise a grammar defining another language, such as XML.

**element** In XML, a start tag and its end tag, plus the content between the tags. An empty tag is also an element.

**empty declaration** In XML, the DTD declaration for an empty tag. For example, if `<foo/>` is an empty tag, the empty declaration looks like: `<!ELEMENT foo EMPTY>.

**empty tag** In XML, a start and end tag combined in one tag. The tag has a trailing slash, so an XML parser can immediately recognize it as an empty tag and not bother looking for a matching end tag. For example, if `foo` is an empty tag, it looks like `<foo/>.

**entity** In XML, an entity declaration provides the ability to have constants or replacement strings, which are expanded by a pre-processor. An entity declaration maps some token to a replacement string. Later the token can be prefixed with the & character and the replacement string is put in its place.

**Enterprise JavaBeans (EJB)** The Enterprise JavaBeans specification defines a way of building transactionally aware business objects in Java.

**Java Server Page (JSP)** Java Server Pages are Web pages that include dynamic tags which are executed on the server. JSPs are the presentation layer for Web-based applications built in Java.

**Servlets** Servlets are Java objects which execute on the server in response to a browser request. They can either generate HTML or XML directly, or call a JSP to produce the output.

**URI/URL** A Uniform Resource Identifier (URI) and Uniform Resource Locator (URL) uniquely defines a location on the Web. URLs are familiar to anyone who browses the Web (for example http://www.ibm.com), and the term URI is a more general term which also incorporates other schemes for identifying resources.

**valid** An XML document is valid if its content conforms to the rules in its DTD.
**Web Application** A WebSphere Web application is a collection of static pages, JSPs and Servlets that share a common URL prefix, and together make a complete application.

**well-formed** An XML document is well-formed if there is one root element, and all its child elements are properly nested within each other. Start tags must have end tags, and each empty tag must be designated as such with a trailing slash. Also, all attributes must be quoted, and all entities must be declared.

**white-space** In XML, characters that are not visible, but used in formatting documents or programs. These characters include the SPACE, TAB, NEWLINE, and CARRIAGE-RETURN characters.

**XSL Stylesheet** The eXtensible Stylesheet Language defines stylesheets for XML Documents. It is composed of two parts: the formatting objects, and XSLT (see below). XSL is defined by the WorldWide Web Consortium.

**XSLT eXtensible Stylesheet Language Transformations** This defines the part of the XSL specification which allows the stylesheet to reformat and reorganize the XML data. It is most often used to transform XML into XSL.

**XSLT Islands** XSLT Islands is a technology for embedding XSL scripts inside a JSP. This allows XML and Java data to be mixed together to produce HTML or XML output.
### List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>BSF</td>
<td>Bean Scripting Framework</td>
</tr>
<tr>
<td>CAE</td>
<td>client application enabler</td>
</tr>
<tr>
<td>URI</td>
<td>Universal Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>DLL</td>
<td>dynamic link library</td>
</tr>
<tr>
<td>DNS</td>
<td>domain name server</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>IDE</td>
<td>integrated development environment</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>JAR</td>
<td>Java archive</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>NLS</td>
<td>National Language Support</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>PIN</td>
<td>personal identification number</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development</td>
</tr>
<tr>
<td>RDBMS</td>
<td>relational database management system</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SQL</td>
<td>structured query language</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet</td>
</tr>
<tr>
<td>URI</td>
<td>Universal Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Universal Resource Locator</td>
</tr>
<tr>
<td>XLink</td>
<td>relational database management system</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>XPath</td>
<td>XML Path Language</td>
</tr>
<tr>
<td>XPointer</td>
<td>XML Pointer Language</td>
</tr>
<tr>
<td>XSL</td>
<td>eXtensible Style Language</td>
</tr>
<tr>
<td>XSLT</td>
<td>XSL Transformations</td>
</tr>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
</tbody>
</table>
Index

A
Attributes 18

B
BasketBean 119, 121, 123

C
Cascading Style Sheets 4
CDATA 19
Channel Definition Format 46
character data 13
CLASSPATH 48
classpath 100
Comments 20
CRM 6
CSS 4

D
Data Definition Language (DDL) 91, 217
DatabaseDom 123, 149, 155, 161
databases
  A118 207, 208
  PROFILES 91, 207, 208
  SUP001 208
  SUPPLIERS 91
DefaultApplyServlet 109
DefaultApplyXSL 55, 63, 101, 110
DHTML 39
DOCTYPE 16
Document Object Model (DOM) 36, 39, 42, 47, 68
  API 40, 42, 66, 68
  DocumentFragment 40
  DOM HTML 39
Document Type Declaration 16, 17
Document Type Definition (DTD) 13, 14, 15, 92
DTD 14, 15, 53, 163

E
e-business 45
ECMAScript 39
EDI 6
EJB 164
Electronic Data Interchange 6
Elements 17

Entities 19
external DTD 17

H
HTML 3, 5, 7, 13
HTTP 3, 5, 65, 68
HttpSession 100
HyperText Markup Language 3
HyperText Transmission Protocol 3

I
IBM XML for Java Parser 63

J
Java Server Pages 45, 65, 70, 89
Java Servlets 65
JDBC 149

L
LotusXSL 45, 55, 62, 72

M
media.properties 83

N
namespace prefix 21
Namespaces 20
NCSA 9
Node 40
Notations 20

P
Parameter Entities 19
Parser
  TX compatibility 54
PCDATA 17
PDA 46
PROFILES database 91

S
SAX
  AttributeList 37
  DocumentHandler 37
  DTDHandler 37
<table>
<thead>
<tr>
<th>Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorHandler</td>
<td>37</td>
</tr>
<tr>
<td>Locator</td>
<td>37</td>
</tr>
<tr>
<td>Parser</td>
<td>37</td>
</tr>
<tr>
<td>servlet</td>
<td>101</td>
</tr>
<tr>
<td>SGML</td>
<td>13</td>
</tr>
<tr>
<td>Simple API for XML</td>
<td>36</td>
</tr>
<tr>
<td>Standard Generalized Markup Language</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>text/xml</td>
<td>83</td>
</tr>
<tr>
<td>text/xml-auto</td>
<td>83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Resource Identifier</td>
<td>21</td>
</tr>
<tr>
<td>URI</td>
<td>21</td>
</tr>
<tr>
<td>USERPROFILE</td>
<td>97</td>
</tr>
<tr>
<td>userprofile</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>validity</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebSphere Application Server</td>
<td>45</td>
</tr>
<tr>
<td>well-formed</td>
<td>14</td>
</tr>
<tr>
<td>well-formedness</td>
<td>35</td>
</tr>
<tr>
<td>Wireless Markup Language</td>
<td>46</td>
</tr>
<tr>
<td>WML</td>
<td>164</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XHTML</td>
<td>164</td>
</tr>
<tr>
<td>XML for Java Parser</td>
<td>45</td>
</tr>
<tr>
<td>XML parsers</td>
<td>10, 14</td>
</tr>
<tr>
<td>XML Schemas</td>
<td>22</td>
</tr>
<tr>
<td>XML/XSL</td>
<td>72</td>
</tr>
<tr>
<td>XML4J</td>
<td>47, 68</td>
</tr>
<tr>
<td>XML4J API</td>
<td>71</td>
</tr>
<tr>
<td>XPath</td>
<td>25, 31, 46</td>
</tr>
<tr>
<td>XSL</td>
<td>45, 55, 72, 161</td>
</tr>
<tr>
<td>XSL processor</td>
<td>55</td>
</tr>
<tr>
<td>XSLT</td>
<td>46, 55, 84</td>
</tr>
<tr>
<td>XSLT Islands</td>
<td>72, 79, 84, 109, 111, 163</td>
</tr>
</tbody>
</table>
IBM Redbooks evaluation

The XML Files: Using XML and XSL with IBM WebSphere 3.0
SG24-5479-00

Your feedback is very important to help us maintain the quality of IBM Redbooks. Please complete this questionnaire and return it using one of the following methods:

- Use the online evaluation form found at http://www.redbooks.ibm.com/
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
_ Customer _ Business Partner _ Solution Developer _ IBM employee
_ None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

______________________________________________________________

______________________________________________________________

______________________________________________________________

What other redbooks would you like to see published?

______________________________________________________________

______________________________________________________________

______________________________________________________________

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

______________________________________________________________

______________________________________________________________

______________________________________________________________


© Copyright IBM Corp. 2000