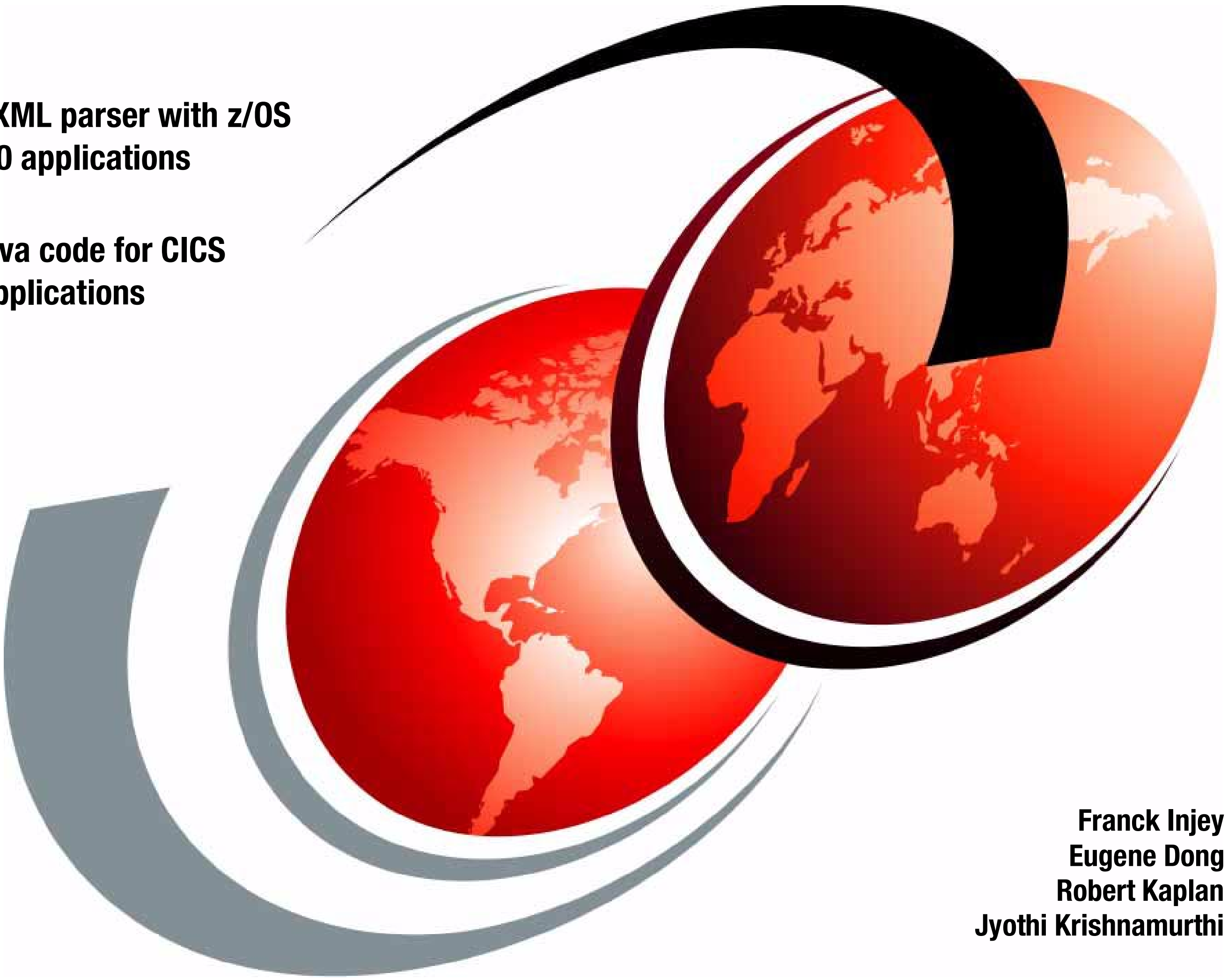
IBM

Using XML on z/OS and OS/390 for Application Integration

XML basics

Using the XML parser with z/OS and OS/390 applications

Sample Java code for CICS and IMS applications



Franck Injey
Eugene Dong
Robert Kaplan
Jyothi Krishnamurthi

Redbooks

ibm.com/redbooks

International Technical Support Organization

Using XML on z/OS and OS/390 for Application Integration

November 2001

SG24-6285-00

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 139.

First Edition (November 2001)

This edition applies to XML Toolkit for z/OS and OS/390 Version 1 Release 2, Program Number 5655-D44 for use with z/OS and OS/390

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HYJ Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

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.

Contents

Contents . . . iii

Preface . . . vii
The team that wrote this redbook . . . vii
Special notice . . . viii
IBM trademarks . . . viii
Comments welcome . . . viii

Chapter 1. Introduction to XML . . . 1
1.1 What is XML . . . 2
1.2 XML vs HTML . . . 2
1.3 Value of XML . . . 3
1.3.1 Business benefits . . . 4
1.3.2 Technical benefits . . . 8
1.3.3 Other benefits . . . 10
1.3.4 XML in a business scenario - an example . . . 10

Chapter 2. XML concepts . . . 13
2.1 Document validity and well-formedness . . . 14
2.2 Document Type Definition . . . 14
2.2.1 DTD example . . . 15
2.2.2 What's in a Document Type Definition . . . 16
2.3 NameSpaces . . . 18
2.4 DTD versus XML Schema . . . 19
2.5 XPath . . . 20
2.6 eXtensible stylesheet language (XSL) . . . 21
2.6.1 Cascading Style Sheets (CSS) . . . 21
2.6.2 XSL = fo: + XSLT . . . 22
2.7 XHTML . . . 23
2.8 Xlink . . . 25
2.9 Xpointer . . . 25
2.10 Real-life uses of XML . . . 25
2.10.1 Resource Description Framework (RDF) . . . 26
2.10.2 SABRE and Wireless Markup Language . . . 26
2.10.3 SMIL . . . 26
2.10.4 MathML . . . 26

Chapter 3. IBM WebSphere Application Server and XML . . . 27
3.1 Overview . . . 28
3.2 WebSphere programming model . . . 28
3.2.1 Servlets . . . 28
3.2.2 Java Server Pages (JSPs) . . . 29
3.2.3 Servlets, JSPs, and Java Beans . . . 29
3.3 Generating XML with WebSphere Application Server . . . 29
3.3.1 The println method . . . 29
3.3.2 The Document Object Model approach . . . 30
3.3.3 The JSP method . . . 31
3.3.4 Comparison of methods - generating XML . . . 32

Chapter 4. Java and XML 33
4.1 Java and XML: a perfect match 34
4.2 XML parsers 34
4.3 SAX 35
4.3.1 SAX classes and interfaces 35
4.3.2 SAX example 36
4.4 DOM 38
4.4.1 DOM hierarchy 38
4.4.2 DOM example 39
4.5 SAX or DOM 41
4.5.1 SAX advantages and disadvantages 41
4.5.2 DOM advantages and disadvantages 41

Chapter 5. XML Toolkit for z/OS and OS/390 43
5.1 XML Toolkit components 44
5.1.1 XML Parser for z/OS and OS/390, Java Edition 44
5.1.2 XML Parser for z/OS and OS/390, C++ Edition 45
5.2 XML Toolkit hardware and software prerequisites 46
5.3 XML Toolkit installation and configuration 46
5.3.1 Obtaining the Toolkit 46
5.4 Installing the XML Parser for z/OS and OS/390, Java Edition 47
5.4.1 XML Parser for z/OS and OS/390, Java Edition, samples 47
5.5 Installing the XML Parser for z/OS and OS/390, C++ Edition 48
5.5.1 XML Parser for z/OS and OS/390, C++ Edition, samples 50
5.6 Java Edition, development and runtime considerations 52
5.6.1 SAX and DOM API level 52
5.6.2 WebSphere Application Server considerations 53
5.6.3 VisualAge for Java considerations 53

Chapter 6. Application integration on z/OS and OS/390 55
6.1 DB2 XML Extender 56
6.2 XML and CICS integration 56
6.2.1 A Web-enabled CICS Application, using WebSphere and CTG 57
6.3 XML-to-back office applications 58
6.3.1 Servlet parsing, CICS Transaction Gateway and CICS application 58
6.3.2 CICS XML parsing, CICS application 60
6.4 XML MQSeries integration 61
6.4.1 Servlet parsing, MQ, CICS application 61
6.4.2 Servlet, MQ Integrator (parsing), CICS application 62
6.5 XML and IMS integration 63
6.5.1 Servlet, IMS ITOC, IMS application 63

Chapter 7. CICS TS applications and XML 65
7.1 CICS TS overview 66
7.2 CICS Transaction Gateway-WebSphere Application Server on z/OS or OS/390 scenario 66
7.2.1 External Call Interface (ECI) 67
7.2.2 Common Connector Framework and Enterprise Access Builder 68
7.3 CICS TS support for Java 69
7.3.1 CICS support for compiled Java code 69
7.3.2 CICS support for the JVM 70
7.4 XML support in CICS 70
7.4.1 Parsing XML in CICS TS 70
7.4.2 Parsing outside CICS TS 72

7.4.3 XML Parser Java Edition and code page conversion 74

Chapter 8. A CICS SampleApp with XML 77
8.1 Application overview 78
8.1.1 Price Change module 79
8.1.2 Price Quote module 80
8.2 SampleApp with XML 81
8.2.1 Defining the XML Request format 81
8.2.2 Creating the EAB records and commands 82
8.2.3 Servlet logic 83
8.2.4 Transaction driver logic 86
8.2.5 Sending the response to the user 86
8.2.6 Defining the XML Response format 87

Chapter 9. XML Parser Java Edition in CICS 89
9.1 JCICS application programming 90
9.2 JCICS and VisualAge for Java 90
9.3 Setting up CICS, Java, and the XML Parser 91
9.4 SAXCount390 sample program 93
9.4.1 Defining the SAXCount390 Java Program to CICS 95
9.4.2 Linking to the SAXCount390 Parser program from CICS 96
9.5 Sample link to a CICS program with COMMAREA 97

Chapter 10. IMS TM applications and XML 99
10.1 IMS overview 100
10.2 Java and XML support in IMS 100
10.3 IMS Connector for Java 100
10.4 IMS Connect 101
10.5 Accessing IMS using the IMS Connector for Java 101

Chapter 11. An IMS SampleApp with XML 103
11.1 Application overview 104
11.1.1 Price Change module 105
11.1.2 Price Quote module 106
11.2 SampleApp using XML 107
11.2.1 Defining the XML request format 107
11.2.2 Creating the EAB records and commands 108
11.2.3 Servlet logic 108

Chapter 12. IMS Java IVP Sample using XML 115
12.1 Overview 116
12.2 The XML 116
12.3 IMS Java Programming Model 117
12.3.1 Putting it together 118
12.3.2 Building your application 120

Appendix A. GNU gmake 123
What is GNU 123
The gmake configuration utility 123
Obtaining gmake 124
Installing gmake 124

Appendix B. Sample IMS IVP with XML 127
IVP 127
IVPDatabaseView 131

InputMessage 132
OutputMessage 132
ParseXMLInput 133
PhoneBookSegment 135
SPAMessage 135
XMLOutput 136

Appendix C. Additional material 137
Locating the Web material 137
Using the Web material 137
System requirements for downloading the Web material 138
How to use the Web material 138

Special notices 139

Related publications 141
IBM Redbooks 141
IBM Publications 141
Other resources 141
Referenced Web sites 142
How to get IBM Redbooks 142
IBM Redbooks collections 142

Abbreviations and acronyms 143

Index 145

Preface

This IBM Redbook will help you to understand how to integrate XML technology with business applications on z/OS. We start by providing a brief tutorial on XML, describing the advantages of its use. We explain its positioning, with emphasis on Java and the IBM WebSphere Application Server.

We then document XML implementation on z/OS and describe the XML Toolkit for z/OS and OS/390, from installation to usage.

We present examples of using XML on existing CICS and IMS applications, with a discussion of design alternatives, and explain how to install implementations in the WebSphere Application Server on z/OS. Finally, we provide sample Java code.

This redbook shows developers how XML can be used to communicate with existing CICS or IMS back office applications, as well as how it can be used to design new applications on z/OS.

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, Poughkeepsie Center.

Franck Injey is a Project Leader at the International Technical Support Organization, Poughkeepsie. He has 25 years of experience working on S/390 hardware and system performance. Before joining the ITSO, Franck was a Consulting IT Architect in France.

Eugene Dong is a developer in the IMS organization working on e-business enablement. He was the IMS Java team leader until recently and is now working on IMS support of XML. Prior to IMS, Eugene was with COBOL development.

Jyothi Krishnamurthi is an e-transformation specialist from Cognizant Technology Solutions, Chennai, India. She has two years of experience in projects which extend legacy applications to the Web using IBM technologies. Her areas of expertise include CICS Transaction Gateway, IMS Connect, Java, and XML technologies. She currently leads a product development initiative at Cognizant for e-transformation automation.

Robert Kaplan is a Systems Management Integration Consultant for IBM Global Services, Schaumburg. He has 25 years of experience working with S/390 operating systems and CICS software. His area of expertise include CICS Parallel Sysplex explotation, CICS performance, and CICS application integration.

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

Rich Conway
International Technical Support Organization, Poughkeepsie Center

Bach Doan, Chris Holtz, and especially Bob Love
IMS Development, IBM Silicon Valley Lab

Bill Carey
z/Series Software System Design, IBM Poughkeepsie

Robin Tanenbaum and Robert St John
WebSphere Design and Performance Analysis, IBM Poughkeepsie

Special notice

This publication is intended to help application developers who are responsible for designing application integration on z/OS using XML technologies. The information in this publication is not intended as the specification of any programming interfaces that are provided by the XML Toolkit for z/OS and OS/390. See the PUBLICATIONS section of the IBM Programming Announcement for the XML Toolkit for z/OS and OS/390 Version 1 Release 2 for more information about what publications are considered to be product documentation.

IBM trademarks

Comments welcome

Your comments are important to us!

We want our IBM Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  `ibm.com/redbooks`

- Send your comments in an Internet note to:

  `redbook@us.ibm.com`

- Mail your comments to the address on page ii.

1

Introduction to XML

This chapter provides a brief overview of XML and explains its value from a business perspective as well as from a technological perspective. It also presents a business scenario in which XML can be efficiently applied.

1.1 What is XML

XML stands for e**X**tensible **M**arkup **L**anguage. It is a simplified subset of the Standard Generalized Markup Language (SGML), which is owned by the World Wide Web Consortium (W3C). XML is an open standard protocol that provides a mechanism to create metalanguage that can define other markup languages. This means that almost any type of data can be easily defined in XML.

There are two major advantages to using XML. The first is that XML is written in a plain text format. This allows it to be compatible with existing computing environments. The second advantage is the extensibility of XML: developers can create their own markup tags, or elements, to best represent the structure and nature of the data. When you define your XML documents, you are actually defining a language to suit your application needs.

XML excels as a format for describing data in a way that can be shared by multiple applications on many platforms. People, as well as computers, can understand the meaning of the data because the author can describe the data by defining each tag in terms that relate to the data structure. XML can be used as a universal data format, and for the exchange of information between systems on intranets or the Internet using Web browsers and Java. XML is also beginning to play an important role in e-business and B2B as a universal data format. 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—namely, XML.

XML is also portable and self-defining, down to the code page used, thus making it possible for other parties (within a business, across an industry, or across multiple industries) to know and understand these tags. Because XML tags represent the logical structure of the data, they can be interpreted and used in various ways by different applications.

XML encapsulates a family of technologies that provide a uniform way of describing, transforming, linking, accessing, and querying information in XML documents. The standards define the syntax of XML and the behavior of the code processing it.

1.2 XML vs HTML

It's worth mentioning that XML and HTML are not the same; while they both have their roots in SGML—and while XML has a "look and feel" similar to HTML—they are not even close to being the same. HTML is a specific markup language that is very useful for specifying how data should be displayed, and it is an application of SGML. XML itself operates at the same level as SGML, and is very powerful for specifying the structure and context of data.

Example 1-1 shows an HTML example:

Example 1-1 HTML

```
   <HTML>
   <HEAD>
   <TITLE>Information</TITLE>
   </HEAD>
   <BODY>
   <H2>Employee ID Search Results</H2>
   <TABLE border=1>
   <TR><TD>EmployeeID</TD><TD>0000002150</TD></TR>
   <TR><TD>Last Name:</TD><TD>SMITH</TD></TR>
   <TR><TD>First Name</TD><TD>ROBERT</TD></TR>
<TR><TD>Company</TD><TD>IBM</TD></TR>
```

```
<TR><TD>Address</TD><TD>PO BOX 2134</TD></TR>
<TR><TD>Zip</TD><TD>75034</TD></TR>
<TR><TD>City</TD><TD>New York</TD></TR>
</TABLE>
<BR>
</BODY>
</HTML>
```

It would be very hard to use any data from the HTML source code. Even though a great deal of data is included in the HTML, the application programs do not know how the data is defined. So, HTML is designed to visually present information to a user from the Web and is good at providing data for people, but not for application programs.

Example 1-2 shows a simple example of XML.

Example 1-2 XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<Employee>
<ID>0000002150</ID>
<Lastname>SMITH</Lastname>
<Firstname>ROBERT</Firstname>
<Company>IBM</Company>
<Address>PO BOX 2134</Address>
<Zip>75034</Zip>
<City>New York</City>
</Employee>
```

This time, there are tags enclosing each data item. It would be easy to use this data from any application program that knows the name of the tags. In this case, we do not care that this is not browsable data. Application programs would be able to interpret this data even though it is not suitable for browsing. In order to present this data to a Web browser, we would need to transform it from XML into something better suited for reading via a browser.

In short, XML by itself, describes *information*, while HTML describes *presentation*.

But XML-based languages like XHTML (refer to 2.7, “XHTML” on page 23) and XSL-FO (refer to 2.6.2, “XSL = fo: + XSLT” on page 22) may be every bit as visually-specific as HTML. XHTML is an obvious case since it is HTML re-expressed in XML compatible syntax, while XSL-FO plays into the role HTML was originally intended to occupy in that it has an abstract page structure and is rendered to meet the needs of specific environments and users.

XML facilitates the separation of data structure and rendering description, but that's in large part because XML is itself neutral, which allows tools to span that gap without having to step outside the XML boundaries.

1.3 Value of XML

In these sections we describe the commercial and technical benefits of XML, as well as other benefits.

1.3.1 Business benefits

A key player in B2B application space

To date, most of the explosion in the Web has come from Business-to-Consumer (B2C) applications (i.e., retail, customer service, reference, search engines, etc.). These involve interactions between a consumer and a business such as purchasing books from a Web site.

The next wave of Internet applications will involve Business-to-Business (B2B) applications like purchasing, order fulfillment and invoicing. The key differences between B2B and B2C applications are that B2B applications have much higher volumes, individual transactions are more complex, and they are trusted transactions that must be executed with very little overhead.

Consumers only purchase part of their goods and services over the Web. There is no overhead to a consumer to use both electronic and traditional commerce. However, for a business to gain value from the Internet, it has to convert significant percentages of its processes to the electronic medium. Supporting multiple ways of doing business is expensive. Therefore, it is more likely that a business will move all its transactions to the e-business world, driving larger numbers of electronic transactions. A B2C application involves much more error-checking, higher overhead, and lower volume. No B2C transaction is likely to involve the single purchase of hundreds, thousands, or even millions of individual items within a purchase or transaction, as is likely in a B2B transaction.

IBM provides products for both sets of applications but as discussed, their needs are very different. XML plays a major role in the B2B application space, due to higher volume requirements inherent in B2B transactions. Automatic processing of these transactions is essential, and XML is one of the most flexible ways to automate Web transactions.

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 to easily build tools that read data, write data and transform data between XML and other formats. This has allowed a number of business and industry consortia to build standard XML data formats. Areas such as Electronic Data Interchange (EDI), inter-bank payments, supply chain, trading, and document management are all subjects of ongoing XML-based standardization by industry consortia.

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 for creating 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.

A standard interface between dissimilar systems

XML can insulate applications from each other. It provides a universal data format that can be the input and output to any application. Since the data is self-defining, the receiving application gets all the information it needs to process the data without requiring knowledge of the sending application. This makes XML a natural fit for Electronic Data Interchange (EDI) applications, as well as for integrating business applications within an enterprise.

EDI generally refers to a set of standards that have been created for data exchange between companies. These are currently binary standards, but there are initiatives to enable the use of XML in EDI. With XML, however, there is no reason for some of the EDI requirements like a Value Added Network (VAN). Trading partners can exchange data across the Internet using XML. However, the current EDI standards are optimized for large volumes and XML over the Internet isn't optimized to the same extent. Therefore, XML is not a total replacement for the EDI standards, but it is complementary.

Over time, EDI will evolve to using more and more XML technology. There is a United Nations effort, in partnership with the Organization for the Advancement of Structured Information Standards (OASIS), to create a set of XML-based standards for global electronic business communications. This effort is called ebXML. For more information on OASIS and ebXML, refer to:

```
http://www.oasis-open.org
http://www.ebxml.org.
```

An inherent requirement in today's global e-business environment is to create a solid infrastructure with the ability to link applications together seamlessly, with minimal effort. The ability to invoke remote, dissimilar applications is critical, and there are many application development models that can enable this integration.

The Component Object Model (COM) and the Common Object Request Broker Architecture (CORBA) are two such models, from Microsoft ® Corporation and the Object Management Group (OMG), respectively. The Java programming language also has similar integration facilities. However, these are object-oriented standards that do not function very well with procedural applications, and one needs to know the model at either end of the communication link in order to use them. Also, these models do not work well in partner-facing applications because they do not easily cross firewalls.

XML provides an answer to these problems in a heterogeneous world where applications from different partners (e.g., Supply Chain Management) need to be linked together. There is an XML-based standard called the Simple Object Access Protocol (SOAP), which uses XML as a format for messages that can invoke services and send a response back to the caller. SOAP is independent of the application development models, but can work in conjunction with them. Being XML-based, it can go through firewalls because it can use the standard Hypertext Transfer Protocol (HTTP) as a transport. SOAP is further discussed in "XML messaging in Web services" on page 7.

For more information about SOAP refer to:

```
http://www.w3.org/TR/SOAP/
```

XML is also an important component in Web Services, a related set of application functions that supports information interchange implemented via SOAP or HTTP protocols. A service descriptor language, WSDL, describes these services, and Universal Description Discovery and Integration (UDDI) makes the applications locatable. SOAP is used both as an envelope for requests and responses, and for encoding data structures.

A Web Services Toolkit is available at:

http://alphaworks.ibm.com.

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 provides an excellent way of linking the application with others.

Insulating business applications from evolving technologies

Putting an XML front-end on existing applications makes them available to new delivery mechanisms like wireless and voice distribution. The only application rework required is the addition of the XML infrastructure. Since XML can easily be converted from one markup language to another, data encoded in XML is available to any new application that requires it.

Therefore, not only is the investment in existing applications preserved, but the applications are now e-business-ready (i.e., they are ready for any follow-on strategic changes for e-business). Also, complex transactions that might involve more than one application can be easily created. A Web server can create multiple XML documents representing complex transactions, which can be executed in any order, using workflow processes or other means.

As mentioned earlier, XML can make data available to both wireless and mobile applications. There is a form of XML called Wireless Markup Language (WML) that, when used with the Wireless Access Protocol (WAP; wapforum.org), can transmit formatted data to Personal Digital Assistants (PDAs), cell phones and other mobile devices. The context preservation in XML allows these devices to make intelligent use of the data for sophisticated applications. Other formats like VoiceML enable browsers to translate text into voice or vice versa for easier-to-use applications that do not require keyboards and display devices for presentations.

Facilitating information reuse

XML tagging captures the *context* of the data, as well as the data itself. This context (or semantics) is also referred to as *metadata*. Thus, a name can be recognized as a name, rather than just a string of characters. Once the data is in XML format, it can be sent to a printer, a Web browser, a PDA, a cell phone, or any other application. The data does not need to be reformatted or stored in any other format. Data from one application can be reused by another without additional programming.

An example where this is required is in globalization of applications. Using XML, the information can be easily translated into different languages, while maintaining the contextual meaning of the data without requiring that it be stored separately in each language. Translation can occur as dynamically as needed.

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 the ability 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 catalog. 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 redbook, 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.

XML messaging in Web services

Web services are a new development and have yet to really blossom. A set of standards has been defined to allow businesses to define services that they wish to make available. These standards can be found at:

`http://uddi.org/`

UDDI stands for Universal Description, Discovery and Integration. Services are defined and published in a directory where businesses that need the services can find them. Service brokers enable requesters to find these services. The three actions of publish, find, and bind are defined by UDDI standards.

The ability to find such published services and bind to them is essential in Business-to-Business e-commerce (B2B). The overhead of having manual entry or human intervention in processes like Supply Chain Integration across partners or Customer Relationship Management is too expensive, slow and cumbersome. Loss of time equates to being late to market or not being as responsive to one’s customers as one needs to be. Both of these can be catastrophic in today’s economy.

Services need to publish their interfaces and parameters in the UDDI directory. This information is defined using a standard called the Web Services Definition Language (WSDL), which is part of the UDDI specification. The actual invocation of the service uses a remote procedure call. However, since this is a call across businesses and between dissimilar applications, it cannot be a simple RMI (Java’s Remote Method Invocation) or IIOP (the Object Management Group’s Internet Inter-Orb Protocol for inter-object communications) call, as these cannot cross firewalls. Also, in order to invoke a service using a RMI call (for example), it is necessary to know the details of the application at the other end. Businesses are not willing to publish such proprietary information to their partners. UDDI takes this into account and uses a XML-based technology called SOAP (Simple Object Access Protocol) or XML Protocol to solve this problem.

SOAP is a specification that has been submitted to the W3C for standardization as the XML Protocol. It defines a message format that represents a remote procedure call. It is transported using Hypertext Transmission Protocol (HTTP). The service to be invoked and the parameters to be passed are defined in the SOAP message. The SOAP header contains the transport information that allows it to be delivered to the correct application. SOAP can also be used for intra-enterprise communication where applications are likely to change. You can read the SOAP specifications at:

`http://www.w3.org/TR/SOAP`

Enabling Web Applications

Hypertext Markup Language (HTML) is the breath that brought the Web to life. However, HTML only describes how data is to be presented. It does not contain information about the context of the data. Once data is put into HTML format, this context is lost. XML, on the other hand, retains the context of the data, and therefore, keeps the data alive and available for reuse. As Web applications move beyond browsing and simple commerce transactions to more complex interactions, this context becomes more vital. Without XML, Web applications will not evolve in sophistication since re-work and manual intervention will be required to insert context as data travels between applications.

Additionally, by preserving the meaning of the data, XML enables more sophisticated and focused searches and extractions. For example, a search for the word Mars conducted today might yield at least three responses:

- A candy bar
- A planet
- A Roman God

With XML tagging, you specify which of these is the target of the search and only those references tagged with a matching context will be presented in the result. This granularity is needed both for personalization of data based on profiles, and in e-marketplaces and portals where the volume of data is very large.

Today, data in many formats can be extracted, transformed and stored in XML format for future retrieval. XML can also be used to trigger the execution of specific transactions and return a response to an application. Products like IBM WebSphere Application Server, DB2 and CICS Transaction Gateway can make this data extraction and transaction execution simpler.

1.3.2 Technical benefits

XML usage offers the following technical benefits.

Reuse of data

The ability to reuse data is an important benefit. For example, different insurance companies could reuse the same customer's data, because it is in a common format. We could reuse the data when building our meta-policy form described in 1.3.4, "XML in a business scenario - an example" on page 10.

Of course, there are already many common file formats in the world of computing that have allowed data reuse. However, these have usually been proprietary and application-specific. XML is neither of those.

Separation of data and display

What are the benefits of separating data and presentation? First, without this separation, we could not achieve the simple reuse of the data. Second, the presentation changes. If you compare the Web sites of five years ago with the Web sites of today, you'll see that they're radically different. In fact, if you look at any successful Web site, you'll probably see at least one redesign a year.

Such redesign occurs not simply to be trendy; it occurs because successful Web sites analyze and react to feedback from users, and the site is redesigned to be more productive and intuitive. Let's again take a bookstore Web site as example: 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.

Extensibility

HTML has been a constantly evolving standard since it emerged, and one of the problems it's 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, which 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.

Let's return again to the library Web site as an example. When the library first indexed books, the Web did not exist. Probably the library catalog had 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.

Semantic information

Another 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.

Richness of data structure

Although simple, XML is powerful enough to compress complex data structures. Different applications require different structures for representing data. Currently, an XML document has essentially a rooted tree structure. However, other possible data structures might be better suited for a particular type of data (for example, tables and graph). For many applications, a tree structure is general enough and powerful enough to express fairly complex data. It is a sensible balance between expressive power and simplicity.

International character handling

One substantial benefit of using XML that should not be underestimated is its capability to handle international character sets. Today, business is often international in scope. This is especially true regarding Web applications, because the Internet easily leaps national borders. It is only natural that business transactions will contain data in different languages. The XML 1.0 recommendation is defined based on the ISO-10646 (Unicode) character set, so virtually all the characters that are used today all over the world are legal characters.

1.3.3 Other benefits

The other main benefits of XML are that it is human-readable, tree-based, and easy to program. 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.

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 reused 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 4, “Java and XML” on page 33.

The main advantage of XML is the preservation of the context with the data. As stated earlier, XML is also an industry standard protocol that is independent of any single vendor or application. The standard provides rules for conformity, ensuring that all vendors implementations result in consistent results. This is a very important advantage.

XML technology development is also open source through The Apache Software Foundation (apache.org). Most vendor implementations of XML and related technologies are based on the Apache code base. Again, this ensures consistency and adds stability and reliability of code to the list of advantages. It also means that any enterprise that wishes to implement XML infrastructures can do so without dependencies on specific products or vendors.

No technology is without drawbacks, especially one as new as XML. The biggest drawback is the fact that XML support is integrated to varying degrees by different vendors in products. Today, tools to generate XML automatically are not as pervasive as HTML tools. However, we are already seeing more of these come to market, and this movement will continue in the near future.

Also, since XML is character-based, it requires CPU for parsers and takes more disk space and network bandwidth than binary data formats.

1.3.4 XML in a business scenario - an example

The problem

Data transfer, as well as transfer time, has been one of the biggest areas of concern for business partners. While technology has provided the means for exchanging the business data electronically, the data formats still do not comply with each other. Interoperability is the key feature for data exchange, and a lack of such framework seems to be hampering business communication.

For example, today an insurance agent’s desk is crammed with a myriad of proprietary applications that he requires to interact with the software installed at the insurance carrier’s end. This situation arises simply because an application at one end needs to understand the data at the other end in its proprietary format.

For a new policy, the agent would like to do business with the insurance carrier offering the lowest quote for the given policy details. However, to get multiple quotes, the agent is forced to type the data into each of the proprietary applications provided by these carriers.

The solution: XML

The solution lies in data format standardization. XML has emerged as the standard for data interchange. ACORD is a non-profit organization which is developing XML standards for the insurance industry so that everyone can communicate in the same language.

Single Entry Multiple Company Interaction (SEMCI) has been a dream for insurance agents. With the emergence of XML as a technology solution, this is becoming a reality. The traditional business model will get transformed as follows:

- The insurance agent will type in the customer's policy data and submit to an insurance exchange.
- The insurance exchange will submit the policy data to all carriers that have registered with the exchange and comply with XML data format.
- The carriers will pick up the policy data, rate the policy, and submit back the quotation to the exchange.
- The exchange will send this data back to the requesting agent.

It's a clear win-win solution for both agents and carriers. Agents do not need to duplicate the data entry for getting multiple quotes, and insurance carriers are developing XML interfaces to their existing applications so that they can do business with such exchanges. Armed with an XML interface, they will now be ready for future business opportunities.

2

XML concepts

In Chapter 1, “Introduction to XML” on page 1, we discuss the reasons why XML was developed in the first place. In this chapter, we discuss basic XML concepts such as tags, Document Type Definitions, and namespaces.

2.1 Document validity and well-formedness

XML is a *metalanguage*, which means that it is a language for describing markup languages. This is done by defining a set of tags for each markup language. XML does not predefine any tags. It allows you to create your own tags. However, the process of defining tags and creating documents using the tags is not arbitrary. Therefore, there are a few rules that XML tags and documents should adhere to, in order to ensure that they are usable by any XML application.

XML has stricter constraints as compared to HTML, which tolerates minor structural irregularities (such as unclosed tags) in the documents it parses. A well-formed XML document should start with an XML declaration, and have a root element that contains all other elements. XML parsers will not accept documents that contain start tags, such as <AUTHOR>, without their corresponding end tags, in this example </AUTHOR>.This differs from HTML, 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 the following Web site for more information:

http://www.w3.org/XML

The notion of validity applies to XML documents which have a Document Type Definition (DTD) or XML Schema (see "DTD versus XML Schema" on page 19 for a description of Schema) associated with them. A Document Type Definition specifies the structure of the XML document by providing a list of elements, attributes, notations and entities that a document can contain.

When an XML document has a DTD associated with it, a validating parser will read the DTD and check whether the document adheres to the rules specified 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.

2.2 Document Type Definition

The DTD specifies the structure of an XML document, thereby allowing XML parsers to understand and interpret the document's contents. The DTD contains the list of tags which are allowed within the XML document and their types and attributes. 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 can be either be stored in a separate file or embedded within the same XML file. XML documents referencing a DTD will contain the `<!DOCTYPE>` declaration which either contains the entire DTD declaration, or specifies the location of an external DTD, as shown in the following example:

```
<!DOCTYPE LibraryCatalogue SYSTEM “library.dtd”>
```

An XML document is not required to have 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. DTDs are also used by tools to create XML documents.

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 Web sites may prove good starting points:

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

2.2.1 DTD example

The DTD has its own syntax, but it is similar to XML in that it also uses markup tags. Example 2-1 shows a simple internal DTD.

Example 2-1 DTD

```
<?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 this 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 a root element. The DOCTYPE name specified in the declaration must match that root element; refer to Example 2-2:

Example 2-2 DOCTYPE

```
authors:
<!DOCTYPE authors [
<!ELEMENT authors(author)+>
```

The second line constitutes an element declaration, and the plus sign (**+**) indicates that the author element must contain one or more author elements, which in turn are declared like this:

```
<!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; refer to Example 2-3:

Example 2-3 Element

```
<!ELEMENT firstname(#PCDATA)>
<!ELEMENT lastname(#PCDATA)>
<!ELEMENT title(#PCDATA)>
```

The last three elements contain only text and are therefore defined as parsed 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. Example 2-4 shows an example of a Document Type Declaration specifying an external DTD:

Example 2-4 External DTD

```
<?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.

2.2.2 What's in a Document Type Definition

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:

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

Refer to Example 2-5:

Example 2-5 DTD Elements

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

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

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

Refer to Example 2-6 on page 17:

Example 2-6 Element attributes

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

Entity is 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:

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

Refer to Example 2-7:

Example 2-7 Entity

```
<!ENTITY prodname "ACME Calendar">
(XML file:)
Thank you for choosing &prodname; as your primary scheduling program
(rendered:)
Thank you for choosing ACME Calendar as your primary scheduling program
```

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

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

Refer to Example 2-8:

Example 2-8 Parameter entities

```
<!ENTITY % commonAtts
"ID ID #REQUIRED
MAKE CDATA #IMPLIED
MODEL CDATA #IMPLIED">
<!ELEMENT CAR (#PCDATA)>
<!ATTLIST CAR %commonAtts>
<!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:

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

Refer to Example 2-9:

Example 2-9 Notations

```
<!NOTATION jpeg SYSTEM "jpeg.exe">
<!NOTATION gif SYSTEM "gif.exe">
<!ELEMENT person (#PCDATA)>
<!ATTLIST person
picformat NOTATION (jpeg | gif) #REQUIRED>
(XML file:)
<person picformat="jpeg">Kelly Brown</person>
```

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

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

Refer to Example 2-10:

Example 2-10 Comment

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

2.3 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 Example 2-11:

Example 2-11 Namespace

```
<xml version="1.0"?>
<library-entry xmlns:authr="authors.dtd"
xmlns:bk="books.dtd">
<bk:book>
<bk:title>XML Sample</bk:title>
<bk:pages>210</bk:pages>
<bk:isbn>1-868640-34-2</bk:isbn>
<authr:author>
<authr:firstname>Joe</authr:firstname>
<authr:lastname>Bloggs</authr:lastname>
<authr:title>Mr</authr:title>
</authr:author>
</bk:book>
</library-entry>
```

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 Uniform Resource Identifier (URI), 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 overridden 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 preceding example would then look as shown in Example 2-12 on page 18:

Example 2-12 namespace scope

```
<?xml version"1.0"?>
<library-entry xmlns:authr="authors.dtd"
xmlns:bk="books.dtd">
<bk:book>
<title>XML & WebSphere</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 belongs to the author namespace.

2.4 DTD versus XML Schema

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 does contain 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 Schema, a W3C Recommendation as of May 2001, aims to provide such functionality and also introduces new semantic capabilities such as support for namespaces and type-checking.

XML Schema is an XML language for describing and constraining the content of XML documents. The purpose of the XML Schema language is to provide an inventory of XML markup constructs with which to write schemas.

Schemas define and describe a class of XML documents by using these constructs to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content, attributes and their values, entities and their contents and notations. Schemas document their own meaning, usage, and function. Thus, the XML Schema language can be used to define, describe and catalogue XML vocabularies for classes of XML documents.

For more information on XML Schema, refer to the specification documents from the W3C:

XML Schema Part 0, Primer is available at the following URL:

`http://www.w3.org/TR/xmlschema-0/`

XML Schema Part 1, Structures is available at the following URL:

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

XML Schema Part 2, Data types is available at the following URL:

http://www.w3.org/TR/xmlschema-2/

2.5 XPath

The name XPath comes from its use as notations in URIs for navigating through XML documents. The aim of XPath is to address parts of an XML document; it 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 forward slashmark (/) signifies an absolute path. A simple example of this follows.

Let us consider an XML document (library.xml) that describes a Library System; refer to Example 2-13. This document will be used for XPath and XPointer examples.

Example 2-13 XPath and Xpointer

```
<? 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">
<title>WebSphere</title>
<copies>10</copies>
</book>
<book ID="B3.2">
<title>great novel</title>
<copies>10</copies>
</book>
<book ID="B5.5">
<title>good story</title>
<copies>10</copies>
</book>
</library>
```

The path `/child::book/child::copies` selects all `copies` element children of `book` which are defined under the document's root. The above path can also be written as `/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. For example:

- The path `/child::book[position()-1]` selects the first `book` element under root. This location step can also be written as `/book[1]` if you wish.
- For example, the path `/book/author/@*` selects all the `author` element attributes.
- The path `/book/author[@type='old']` selects all the `author` elements with type attribute equal to `'old'`.

2.6 eXtensible stylesheet language (XSL)

Extensible Stylesheet Language 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 document, 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).

Cascading Style Sheets 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 Cascading Style Sheets, and so a short description of that language is provided in the following section. Cascading Style Sheets can also be used as a formatting language for XML, but it is less powerful than XSL. Because this language 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 Cascading Style Sheets have been incorporated into XSL. More information on DSSSL can be found at the following Web address:

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

2.6.1 Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) 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. Cascading Style Sheets allows the Web developer to define styles that apply as follows:

- 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 Example 2-14. 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 browser being shipped that supports this feature is Microsoft Internet Explorer 5.0.

Example 2-14 Cascading Style Sheets (CSS)

```
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;
```

Cascading Style Sheet can be used within a document, or referenced in a separate stylesheet, which is the more common approach. For more information, refer to the following Web site:

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

2.6.2 XSL = fo: + XSLT

Although XSL has derived much from CSS, the approach of XSL is much more powerful, and has major differences. The latest W3C proposed recommendation for XSL is available at the following URL:

`http://www.w3.org/TR/xsl`

Because XSL defines an extra step in presenting data, it can do much more powerful presentation tasks than Cascading Style Sheet can. CSS always retains the order of the source tree, whereas XSL can reorder 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.

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/xslt11/`

XSLT defines a common language for transforming one XML document into another document. 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 as `fo:` in the XSL code. The main difference between CSS and XSL FO is that XSL is based on an XML format properly defined with a DTD, while CSS is not.

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 reorder 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), an XML format for drawing graphics.

XSL-FO (XSL Formatting Object) is another function of XSL. XSLT can be used to transform an XML document into another XML document. XSL-FO defines how to display the resulting tree. You can track the progress of the XSL-FO specification on the W3C Web site at:

`http://www.w3c.org/`

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 experimental, we chose not to cover them in detail in this redbook.

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 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.

2.7 XHTML

XML is often compared with HTML, but it’s important to note that XML is not an extension of HTML. As we mentioned before, XML and HTML are quite different in their purpose. HTML is used for presentation and XML is used for describing data.

XHTML (or eXtensible Hypertext Markup Language) is the real extension of HTML. XHTML is actually an extension of HTML 4.0; it is HTML using XML syntax.

The main differences between HTML 4.0 and XHTML are that in XHTML:

- HTML elements and attributes are written in lower-case characters.
- Every attribute must be enclosed in double quotes(") or single quotes(').

- The end tag must be present.
- The document must be in a tree structure, that is, the tags must be strictly nested.

For example, we could write HTML the following way:

Example 2-15 Valid HTML tags

```
<UL>
<LI>List1
<LI>List2
/UL>
```

This would not be allowed in XHTML. Example 2-16 shows the correct XHTML syntax.

Example 2-16 Correct XHTML

```
<ul>
<li>List1</li>
<li>List2</li>
</ul>
```

XHTML has the following three types of DTDs. Every XHTML document must use one of these DTDs.

XHTML Transitional	This is the closest DTD to the HTML DTD. This DTD can make the most of XHTML functionality including style sheet and also make it easy to transit from the existing HTML.
XHTML Strict	This is the strict DTD. We need to use W3C Cascading Style Sheets (CSS) with this DTD.
XHTML Frameset	This is a DTD for using Frameset (this is the same as HTML 4.0 Frameset except for changes due to the differences between XML and SGML).

A Strictly Conforming XHTML Document must meet all of the following criteria:

- It must validate against one of the three DTDs previously mentioned.
- The root element of the document must be `<html>`.
- The root element of the document must designate the XHTML namespace using the `xmlns` attribute `[XMLNAMES]`. The namespace for XHTML is `http://www.w3.org/1999/xhtml`.
- There must be a DOCTYPE declaration in the document prior to the root element. The public identifier included in the DOCTYPE declaration must reference one of the three DTDs previously mentioned: `Transitional`, `Strict` or `Frameset`.

Example 2-17 shows a simple example:

Example 2-17 XHTML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>XHTML sample</title>
</head>
<body>
<p>Hello World!</p>
</body>
</html>
```

For more information about XHTML, refer to the specification documents from the W3C:

`http://www.w3.org/TR/xhtml1/`

2.8 Xlink

Linking technology is really needed to use the Internet. One of the major reasons for the great success of HTML in the Internet is HTML linking capability. HTML linking makes it possible to connect all over the world by one click of a mouse. Example 2-18 shows a simple example of HTML linking:

Example 2-18 HTML Linking

```
<a href="http://fish.net/">Fish.net</a>
```

XML also has a linking technology: XML Linking Language, or Xlink. A simple use of Xlink is very similar to that of HTML linking. Example 2-19 shows an example:

Example 2-19 Xlink

```
<a xml:link="simple" href="http://fish.net/">Fish net</a>
```

But the HTML link function only allows for a one-way link. Xlink uses other technology, HyTime and Text Encoding Initiative (TEI), for multidirectional links. Actually, Xlink has many extensions, compared to HTML link. For example, Xlink supports the following functions:

- Mutual linking between two objects
- Linking from one object to multiple objects
- No need to have tags for linking

For more information about Xlink, refer to the specification documents from the W3C:

`http://www.w3.org/TR/xlink/`

2.9 Xpointer

The XML Pointer Language (Xpointer) is the language to be used as the basis for a fragment identifier for any URI reference that locates a resource whose Internet media type is one of `text/xml`, `application/xml`, `text/xml-external-parsed-entity`, or `application/xml-external-parsed-entity`.

In XPointer, one defines the addressing expression to link XML documents using XPath. For more information about XPointer, refer to the specification documents from the W3C:

`http://www.w3.org/TR/xptr`

2.10 Real-life uses of XML

XML is already used in a wide range of fields and industries including science and medicine, publishing, broadcasting, communications and financial services. In the following sections, we show examples of the world which XML has made possible.

For more examples, see the XML section in the IBM developerWorks site:

`http://www.ibm.com/developer/xml`

Alternatively, visit the xml.org catalog site:

`http://www.xml.org/xmlorg_registry/index.shtml`

2.10.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a W3C recommendation that integrates a variety of applications using XML as an interchange syntax. An RDF description, also referred to as metadata, can include the author of the resource, the date of creation, key word, information about the content, etc.

RDF has many uses on the Web, including organizing, searching, filtering, and personalizing Web sites. In order for RDF to have these benefits, however, Web sites, search engines, and directories must agree to use a standard format for metadata. RDF is an application of XML for encoding, exchanging, and reusing structured metadata.

2.10.2 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 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.

2.10.3 SMIL

In broadcasting and communications, XML is used to describe multimedia content such as sound, voice, and moving pictures. SMIL (or Synchronized Multimedia Integration Language) is one of the major languages to describe multimedia using XML. SMIL is not a protocol to describe these contents directly, but instead describes how to control the timing of playing and showing data. In SMIL, we can decide the layout of contents, synchronization timing, and so on, with many kinds of media files.

For more information about SMIL, refer to the specification documents from the W3C:

`http://www.w3.org/AudioVideo/`

2.10.4 MathML

In the scientific community, XML is also widely used because drawing complex expressions and characters is a suitable task for using XML. Mathematical Markup Language (MathML) is an XML solution used to draw mathematical expressions that are difficult to visualize using plain text format. MathML makes it possible to describe mathematical expression in plain text format.

There are many tools which use MathML. Using these tools and XHTML, it is possible to use MathML inside HTML and browse the expression directly.

For more information about MathML, refer to the specification documents from the W3C:

`http://www.w3.org/Math/`

3

IBM WebSphere Application Server and XML

In this chapter, we discuss WebSphere Application Server, the IBM offering to the e-business marketplace. WebSphere Application Server has integrated XML technology since Version 2. Combined with the connectors provided by the various back office products, it can act as an ideal application integration platform.

3.1 Overview

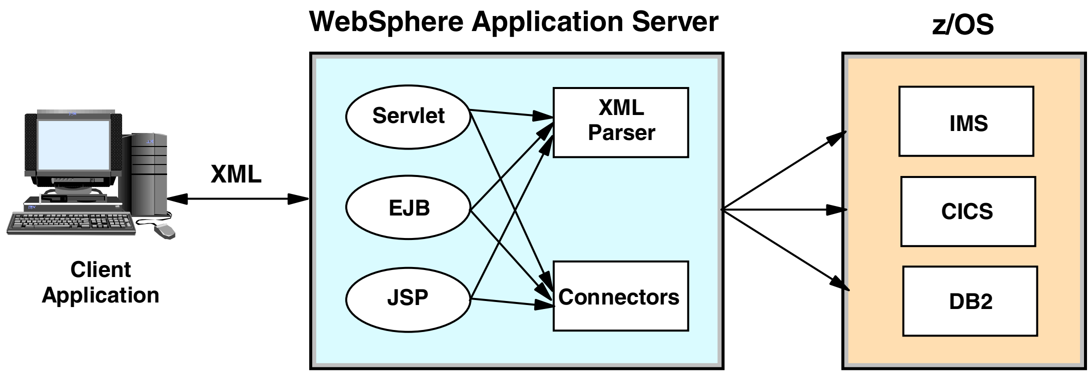


Figure 3-1 WebSphere Application Server

In Figure 3-1, the client application passes XML to the application server. It can be processed by a server component like a servlet, a Java Server Page (JSP), a Java Bean, or an Enterprise Java Bean (EJB).

This component can have the business logic determine how to process the request that is represented by the XML document. It can then be transformed into an IMS Message, a CICS COMMAREA, or a JDBC request, and be sent to the appropriate back office system. The Enterprise Access Builder (EAB) function of VisualAge for Java can be used to map the appropriate data structure and create the logic to do the transformation. Connectors such as IMS Connector for Java or the CICS Transaction Gateway can be used to transmit the request to the transaction manager.

Parsing and transformation servlets, however, currently need to be manually written.

The advantage of using WebSphere Application Server as an integration point is that there is a clear separation between presentation, business logic, and data. As the XML technologies evolve, WebSphere Application Server can serve up multiple applications as Web services.

3.2 WebSphere programming model

WebSphere Application Server supports three kinds of Java programming objects: Java Servlets, Java Server Pages (JSP), and Enterprise Beans. Using XML inside Enterprise Beans is really no different from using XML inside any Java program.

3.2.1 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—instead, 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.

For more information on servlets, see the JavaSoft specification:

http://java.sun.com/products/servlet/index.html

3.2.2 Java Server Pages (JSPs)

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; WebSphere evaluates them, then converts them into static output stream, which is sent to the client.

JSP’s 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 has been extended with an optional XML-compatible syntax to make it easier to create XML documents with JSPs. For more information, refer to:

`http://java.sun.com/products/jsp/index.html`

3.2.3 Servlets, JSPs, and Java Beans

The servlet and JSP programming model also includes the use of Java Bean component. Java Beans are the component model for Java, and Java Beans can be used within JSP’s. This allows JSPs to easily access Java data. A JSP can create a Java Bean 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.

3.3 Generating XML with WebSphere Application Server

The first step in using XML inside WebSphere Application Server is to generate and output XML. 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 Java Beans, and outputting the results in a servlet or JSP
- Using XML format JSP’s to set elements of the XML document based on Java data

3.3.1 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.

Servlet code in Example 3-1 shows how to generate simple XML output to a browser:

Example 3-1 Generate XML output to a browser

```
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.

Of course, this example only generates static XML, and could have been easily done with a static file. However, the same principles 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.

3.3.2 The Document Object Model approach

The Document Object Model (DOM) allows Java to manipulate XML documents as in memory objects. The DOM API is included in the XML4J 3.1.1 class library, and allows XML documents to be created, manipulated, and output.

The servlet shows in Example 3-2 has the same result as the previous servlet, but uses the DOM to create the XML structure.

Example 3-2 Create XML using a DOM

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;
import org.w3c.dom.*;
import org.apache.xerces.dom.DocumentImpl;
import org.apache.xml.serialize.XMLSerializer;
import org.apache.xml.serialize.OutputFormat;

public class buildDOM extends HttpServlet
{
public void doGet(HttpServletRequest request,HttpServletResponse response)
{
response.setContentType("text/xml");
try {
  PrintWriter out =response.getWriter();

  String domImplementation ="org.apache.xerces.dom.DocumentImpl";

// Create a document object
  Document d =(Document)Class.forName(domImplementation).newInstance();

// Create the 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);

// Serialize the DOM you've just created and return the result
  XMLSerializer serializer = new XMLSerializer(out, new OutputFormat(d));
  serializer.serialize(d);
}
catch (Exception e)
{
e.printStackTrace();
}
}}
```

Following are points to note about this example:

- 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 to 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 a problem.
- WebSphere and XML4J provide the tools to easily print XML from a Document object.

3.3.3 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 1.1 example generates an XML document with some dynamically generated values containing data about the server process:

Example 3-3 Generate an XML document using JSP 1.1

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

3.3.4 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. However, it requiresthe skills of 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 similar to the servlet since 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 it 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.

4

Java and XML

In the previous chapters, we describe the basics of XML and its use in different applications. In this chapter, we touch on the other important technology: Java and its role with XML.

4.1 Java and XML: a perfect match

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. Instead, 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. Java is, in fact, the ideal counterpart for XML and the reason can be summed up in a single phrase: Java is portable code, and XML is portable data.

Note that both technologies have their limitations. Java requires the developer to format network data and presentation and use technologies like Java Server Pages (JSP) that do not provide a real separation of content and presentation layers, while XML is simply metadata and does not have processing power of its own. However, Java and XML match together to fill in the gaps in the application development picture.

Writing Java code assures that any operating system and hardware with a Java Virtual Machine (JVM) can run the compiled bytecode. Adding to this XML's ability to represent input and output to the applications with a system-independent, standards-based data layer, and the data becomes portable.The application is completely portable, and can communicate with any other application using the same standards.

One technical advantage of Java over other languages is its built-in support for Unicode. With other languages, XML processing has to be done 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 for implementing XML processors and other XML-related tools and applications.

4.2 XML parsers

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 (well-formedness and validity) of the document.

Understanding the XML parser is very important. Although XML syntax will let us describe data in a plain text format and can be used as a universal data format, it is still just plain text data. An XML parser will help us use this data directly. An XML parser will parse a plain text XML document into an in-memory format that can be manipulated programmatically via an API.

An XML parser is sometimes called an XML processor. The XML Toolkit for z/OS and OS/390 includes XML parsers for Java and for C++. The Java parser is also called XML4J.

Each XML document must be verified as conforming to XML syntax guidelines. XML parsers verify the document syntax. An XML parser must check each XML document for well-formed syntax. Some XML parsers can also check for the validity of a document against a DTD or XML schema.

An XML parser also provides a structured interface to access the values of the elements and attributes of an XML document. Anyone can 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 are widely used:

- Simple API for XML
- Document Object Model

4.3 SAX

The 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 XML document is processed just once, passing each element event to the event handler. Every application which uses SAX must register SAX event handlers to a parser object. The SAX driver can be implemented by the XML parser vendor, or as an add-on to the parser.

4.3.1 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.

If we write this in Java code, the following shows this registration:

```
parser.setDocumentHandler(new myDochandler() );
```

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 org.xml.sax package contains the SAX core implementation classes, interfaces and exceptions. The org.xml.sax.helpers package contains convenience classes and a specific class (`ParserFactory`) for dynamically loading SAX parsers.

The implementation is included in the Java XML parser shipped as part of the XML Toolkit for z/OS and OS/390. For full descriptions, in JavaDoc format, of all classes and interfaces defined in SAX 1.0, visit the following site:

http://www.megginson.com/SAX/

4.3.2 SAX example

In order to use SAX in a Java application, we need a class that implements the most suitable interface for the application. The code fragment in Example 4-1 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.

Example 4-1 DocumentHandler methods

```
public class XmlDocHandler implements org.xml.sax.DocumentHandler
...
public void characters(char[] arg1, int start, int length) throws
org.xml.sax.SAXException {
System.out.println(new String( arg1 ));
}
public void startDocument() throws org.xml.sax.SAXException {
System.out.println("Start of document");
}
```

```
public void endDocument() throws org.xml.sax.SAXException {
System.out.println("End of document");
}
public void startElement(String name, org.xml.sax.AttributeList arg2)
throws org.xml.sax.SAXException {
System.out.println("Start of element " + name);
}
public void endElement(String name) throws org.xml.sax.SAXException {
System.out.println("End of element " + name);
}
```

The application that uses the `DocumentHandler` implementation in Example 4-1 on page 36 is simple. The Apache Xerces Java parser implements the parser interface in SAXParser class, as shown in Example 4-2:

Example 4-2 Parser interface

```
...
Parser parser = ParserFactory.makeParser("org.apache.xerces.parsers.SAXParser");
XmlDocumentHandler hndlr = new XmlDocumentHandler();
parser.setDocumentHandler(hndlr);
Parser.parse(anXMLFileURL);
...
```

Given the following XML document as input:

```
<?xml version="1.0"?>
<employee>
<person id="id1">
<name>
<lastname>Smith</lastname>
<firstname>Robert</firstname>
</name>
</person>
</employee>
```

The output of the SAX application would look like this:

```
Start of document
Start of element employee
Start of element person
Start of element name
Start of element lastname
Smith
End of element lastname
Start of element firstname
Robert
End of element firstname
End of element name
End of element person
End of document
```

4.4 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.

Using DOM, one can access XML data represented as a tree object model. In DOM programming, every element is handled as a “node” which is defined as an interface in the DOM.

DOM creates a tree of the entire XML document in memory and can access and manipulate any of these objects using the DOM API. Sometimes DOM is not the best API to use when handling a large document, since it holds the whole object tree in memory.

4.4.1 DOM hierarchy

DOM is a set of interfaces which are mainly composed of nodes. There is a hierarchy between these interfaces.

Figure 4-1 shows the hierachy of the interfaces; node object is the primary data type in this model. Using the DOM API, we can handle node objects in many ways. We introduce some of the methods which are frequently used in DOM programming.

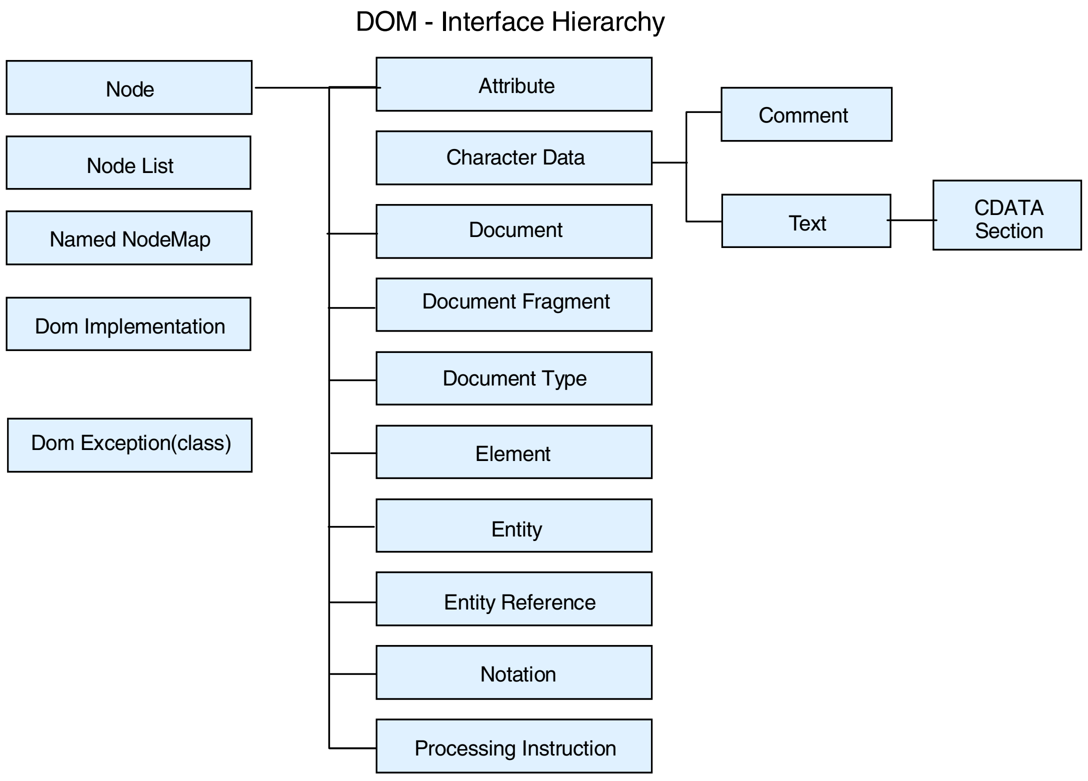


Figure 4-1 DOM Interface Hierarchy

There are two ways of handling node objects:

1. Referring and setting a value to each node object
2. Accessing the tree structure with a node object

For the first way (referring and setting value to each node object), we often use the following four methods:

`getNodeName()`	This method returns the name of a node. The name depends on the type of node it is. Sometimes it is an element name and sometimes it is an attribute name.
`getNodeValue()`	This method returns the value of a node. The value depends on the type of node it is.
`getNodeType()`	This method returns the type of a node.
`setNodeValue(arg)`	This method sets a value to the node.

For the second way (accessing tree structure with node object), we often use the following methods:

`getParentNode()`	This method gets the parent node.
`getPreviousSibling()`, `getNextSibling()`	These methods get the same tree level of node.
`getFirstChild()`, `getLastChild()`, `getChildNode()`, `getElementsByTagName()`	These methods get children nodes.
`appendChild()`, `removeChild()`, `replaceChild()`	These methods are used for adding and deleting a child node.

In this way, we can manipulate XML documents easily with the DOM API.

4.4.2 DOM example

When the simple XML document we used in our SAX example (see 4.3.2, “SAX example” on page 36) is processed using DOM, the resulting object tree will look like the one in Figure 4-2 on page 40. The shaded rectangles represent character data, and the others represent elements.

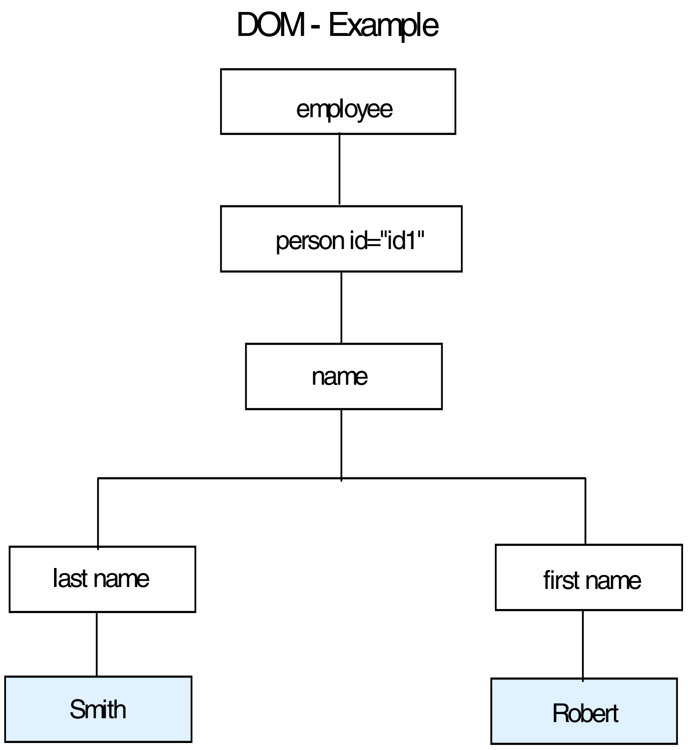


Figure 4-2 Sample DOM tree

Reading an XML document using DOM is relatively easy, provided that a good parser is available. Among other things, the XML Toolkit for z/OS and OS/390 provides a robust and very complete implementation of the W3C DOM API.

Example 4-3 shows a simplified example of how to read and manipulate an XML document using the `DOMParser` class:

Example 4-3 Read and manipulate an XML document using the DOMParser class

```
DOMParser parser = new DOMParser();
parser.parse(uri);
Document document = parser.getDocument();
print(document); // implemented in the code
Node n = document.getLastChild().getFirstChild();
n.setNodeValue("New Value");
print(document);
```

For more information about DOM, refer to the specification documents from the W3C Web site:

http://www.w3.org/DOM/

4.5 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.

4.5.1 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.

However, because the events must be handled as they occur, it is impossible for a SAX application, for example, to traverse backward 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.

4.5.2 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.

5

XML Toolkit for z/OS and OS/390

In this chapter we introduce the XML Toolkit for z/OS and OS/390. We list parser operating environments, outline system prerequisites of the XML Toolkit for z/OS and OS/390, and provide instructions for obtaining the XML Toolkit for z/OS and OS/390 from the Web.

The material provided in this chapter is intended to complement (not replace) that included in the *Program Directory for XML Toolkit for z/OS and OS/390,* GI10-0665, which is available in PDF format at:

`http://www.ibm.com/servers/eserver/zseries/software/xml/pdf/gi100665.pdf`

Note: As this publication goes to print, the XML Toolkit for z/OS and OS/390 Version 1 Release 3 has been announced. This new release provides updates to the XML Java and C++ parsers, based on Apache Xerces-C 1.5.0 and Xerces-J 1.4.2, including SAX 2.0 and DOM 2.0 API support as well as Namespace, Schema and JAXP support. The release also includes Java and C++ XSLT Processor support, called LotusXSL, based on the Apache Xalan_C 1.2 and Xalan_J 2.2 processors.

For more information, see the XML Toolkit Web page at:

`http://www.ibm.com/servers/eservers/zseries/software/xml`

5.1 XML Toolkit components

In November 1999, IBM announced that it was donating its Java (XML4J) and C++ (XML4C) parser technologies to the Apache XML project. The Apache XML project team renamed the parser technologies “Xerces.” Information about the Apache XML project is documented at their Web site:

```
http://xml.apache.org/index.html
```

The XML parsers in the XML Toolkit for z/OS and OS/390 are based on the Apache Xerces code base. The Java parser is based on the Apache version 1.2.1 code base, while the C++ parser is based on Apache version 1.3.0 and ICU 1.6 (International Components for Unicode).

The toolkit includes public APIs and experimental APIs. Public APIs are those that have reached W3C recommendation status. Public APIs are considered stable, meaning that their interface is not expected to change. Fixes will be made available for Severity 1 or 2 problems with public APIs.

Experimental APIs are those that have not reached W3C recommendation status. They are subject to change and are not supported. Consequently, fixes may not be made available for these experimental APIs.

The XML Toolkit for z/OS and OS/390 V1R2 includes:

- XML Parser for z/OS and OS/390, Java Edition
- XML Parser for z/OS and OS/390, C++ Edition

Each parser included in the toolkit includes public and experimental APIs, as follows:

- Public APIs
  - DOM Level 1
  - SAX Level1
- Experimental APIs
  - DOM Level 2
  - SAX Level 2
  - Java Parser partial April 7 W3C Schema specification

5.1.1 XML Parser for z/OS and OS/390, Java Edition

The z/OS and OS/390 technologies that can be integrated with the Java XML parser are listed in Table 5-1.

Table 5-1 Java XML Parser Operating Environments

Parser	CICS	WebSphere Application Server	IMS	Batch
Java Parser	Yes	Yes	Yes	Yes

CICS TS

There are two ways to use XML with CICS:

1. The XML Java parser can be used with CICS TS version 1.3 and above. The Java program must run under the Java Virtual Machine (JVM). Each CICS transaction running under a JVM runs on a separate TCB.

Java programs within CICS using the Java XML APIs must run in a JVM in order to prevent operating system waits from affecting other CICS transactions.

2. The XML Java parser can be used in a Java program that runs outside of the CICS environment; it converts the parsed data into a COMMAREA, and communicates with CICS.

WebSphere Application Server

There are no restrictions when using XML Java parser APIs with WebSphere Application Server. However, WebSphere Application Server also provides its own set of XML Document Structure Services, which contains an XML parser. See 5.6.2, “WebSphere Application Server considerations” on page 53.

IMS Transaction Manager

There are two ways to use XML with IMS:

1. For existing IMS transactions, parsing and transformation servlets convert an XML stream to and from an IMS transaction message data structure.
2. New or modified IMS applications can use the Java parser APIs. Java IMS application programs need to be compiled using the High Performance Compiler option in the VisualAge for Java. The reason that Java programs within IMS must be compiled with the High Performance Compiler option in the VisualAge for Java is because IMS does not use JVMs. Instead, each IMS transaction runs under its own TCB.

Batch

Batch Java programs that invoke the XML Java parser must run in a JVM.

5.1.2 XML Parser for z/OS and OS/390, C++ Edition

The z/OS and OS/390 technologies that can be integrated with the C++ XML parser are listed in Table 5-2.

Table 5-2 C++ XML Parser Operating Environments

Parser	CICS	WebSphere Application Server	IMS	Batch
C++ Parser	No	Yes	Yes	Yes

CICS TS

Using C++ parsers within CICS is not supported. This is because CICS C++ programs run under a single TCB (the quasi-reentrant (QR) TCB) and are not allowed to access files that are not controlled and managed by CICS TS.

WebSphere Application Server

There are no restrictions when using XML C++ parser APIs with WebSphere Application Server.

IMS Transaction Manager

There are no restrictions when using XML C++ parser APIs with IMS.

Batch

There are no restrictions when using XML C++ parser APIs in batch.

5.2 XML Toolkit hardware and software prerequisites

On S/390 hardware, software prerequisites are:

- OS/390 Version 2 Release 6 or higher
- OS/390 Version 2 Release 6 Language Environment or higher

On zSeries hardware, software prerequisites are:

- z/OS Version 1 Release 1
- z/OS Version 1 Release 1 Language Environment

Each environment requires UNIX System Services to be active. Java 1.1.8 (5655-A46) with APAR OW47220 (or higher release level installed) is a prerequisite to using the XML Parser for z/OS and OS/390, Java Edition.

5.3 XML Toolkit installation and configuration

In the following sections, we describe how to install and configure the XML Toolkit.

5.3.1 Obtaining the Toolkit

The XML Toolkit for z/OS and OS/390 is distributed on a product tape, and it can also be downloaded from the Web. When installing from the product tape, use the instructions listed in *Program Directory for XML Toolkit for z/OS and OS/390,* GI10-0665.

If you have a version of the XML Toolkit on your current system, you will want to install the XML Toolkit for z/OS and OS/390 Version 1 Release 2 into its own SMP/E environment.

When installing (using SMP/E) from the files that you download from the Web, refer to the XMLSMPE.README.txt download file and the SMP/E instructions included in the program directory.

The XML Toolkit for z/OS and OS/390 home page is:

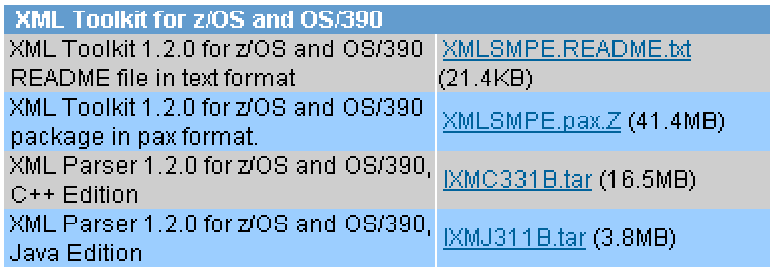
```
http://www.ibm.com/servers/eserver/zseries/software/xml
```

To download the XML Toolkit for z/OS and OS/390 files from the Web, go to:

```
http://www.ibm.com/servers/eserver/zseries/software/xml/download
```

After clicking the link XML Toolkit for z/OS and OS/390 **download**, fill out the required registration information. Once your registration is completed, the XML Toolkit for z/OS and OS/390 file download page will be displayed.

The files that can be downloaded from the Web are listed in Figure 5-1 on page 47:



XML Toolkit for z/OS and OS/390	
XML Toolkit 1.2.0 for z/OS and OS/390 README file in text format	XMLSMPE.README.txt (21.4KB)
XML Toolkit 1.2.0 for z/OS and OS/390 package in pax format.	XMLSMPE.pax.Z (41.4MB)
XML Parser 1.2.0 for z/OS and OS/390, C++ Edition	IXMC331B.tar (16.5MB)
XML Parser 1.2.0 for z/OS and OS/390, Java Edition	IXMJ311B.tar (3.8MB)

Figure 5-1 XML Toolkit for z/OS and OS/390 download files

5.4 Installing the XML Parser for z/OS and OS/390, Java Edition

If you are going to use SMP/E to install the XML Toolkit for z/OS and OS/390, refer to the instructions included in *Program Directory for XML Toolkit for z/OS and OS/390,* GI10-0665.

Non-SMP/E installation instructions for XML Parser for z/OS and OS/390, Java Edition are as follows.

1. From the TSO command environment invoke the OS390 shell and enter the command:

   ```
   OMVS
   ```

2. Create the directories needed to place the IXMJ311B.tar file in directory /usr/lpp/ixm/IBM:

   ```
   cd /usr/lpp/
   mkdir ixm         (if this directory does not exist)
   cd ixm
   mkdir IBM         (if this directory does not exist)
   cd IBM
   ```

3. Upload (in BINARY format) the file IXMJ311B.tar to: /usr/lpp/ixm/IBM.
4. Make sure you are in directory /usr/lpp/ixm/IBM, then extract files from the tar archive by entering the command:

   ```
   tar -xvozf IXMJ311B.tar
   ```

The extract will create a XML4J-3_1_1 directory, with subdirectories for samples, XML documents (data), and documentation in HTML format (docs).

5.4.1 XML Parser for z/OS and OS/390, Java Edition, samples

Sample Java programs that show the use of the SAX and DOM APIs are included with the XML Parser for z/OS and OS/390, Java Edition. Refer to the sample programs listed in Table 5-3 for examples of using the toolkit's SAX and DOM APIs.

Table 5-3 Java Parser samples

XML Parser example	Example description
SAXCount	Counts the elements, attributes, spaces and the characters in an XML file.
SAXWriter	Parses an XML file and prints it out.
DOMCount	Counts the elements, attributes, spaces and characters in an XML file

XML Parser example	Example description
DOMFilter	Parses an XML document searching for specific elements by name or elements with specific attributes.
DOMWriter	Parses an XML document and prints it out.
IteratorView	An interactive UI sample that displays the DOM tree.
TreeWalkerView	An interactive UI sample that displays the DOM tree.

Setting up to run the Java parser samples

To run the Java parser samples, include the xerces.jar and xercesSamples.jar in your CLASSPATH. Figure 5-2 shows statements for including XML Parser for z/OS and OS/390, Java Edition samples within your CLASSPATH.

```
export XERCESJROOT=/usr/lpp/ixm/IBM/XML4J-3_1_1
export CLASSPATH=.:$XERCESJROOT/xerces.jar:$XERCESJROOT/xercesSamples.jar:$CLASSPATH
```

Figure 5-2 CLASSPATH to run Java parser samples

Running the Java parser samples

As mentioned, the SAX and DOM sample programs are included in the toolkit. To run the SAX and DOM sample programs that count the elements, attributes, spaces, and characters in an XML document, enter the commands listed in Figure 5-3.

```
java sax.SAXCount $XERCESJROOT/data/personal.xml
java dom.DOMCount $XERCESJROOT/data/personal.xml
```

Figure 5-3 SAXCount and DOMCount Java Parser samples

5.5 Installing the XML Parser for z/OS and OS/390, C++ Edition

If you are going to use SMP/E to install the XML Parser for z/OS and OS/390, C++ Edition, refer to the instructions included in *Program Directory for XML Toolkit for z/OS and OS/390,* GI10-0665.

Non-SMP/E installation instructions for XML Parser for z/OS and OS/390, C++ Edition are as follows:

1. From the TSO command environment, invoke the OS390 shell and enter the command:

   ```
   OMVS
   ```

2. Create the directories needed to place the IXMC331B.tar file in directory /usr/lpp/ixm/IBM

   ```
   cd /usr/lpp/
   mkdir ixm     (if this directory does not exist)
   cd ixm
   mkdir IBM     (if this directory does not exist)
   cd IBM
   ```

3. Upload (in BINARY format) the file IXMC331B.tar to: /usr/lpp/ixm/IBM.

4. Make sure you are in directory /usr/lpp/ixm/IBM, then extract files from the tar archive by entering the command:

   ```
   tar -xvozf IXMC331B.tar
   ```

   The tar extract will create a xml4c3_3_1 directory, with subdirectories for samples, XML documents (data), and documentation in HTML format (docs.)

5. From the ISPF 3.2 panel, allocate two PDS datasets using the file attributes listed in Table 5-4. We used SYS1 for the high-level qualifier (HLQ):

SYS1.SIXMMOD1	Used for the library files required to run the XML Parser for z/OS and OS/390, C++ Edition on z/OS or OS/390
SYS1.SIXMEXP	Used by the binder to resolve references to functions and variables

Table 5-4 C++ Parser PDS data set attributes

	SIXMMOD1	**SIXMEXP**
Data set name	SYS1.SIXMMOD1	SYS1.SIXMEXP
3390 cylinders	20	1
Directory blocks	3	1
Record format	U	U
Record length	0	80
Block size	32760	3200
Data set name type	PDS	PDS

6. From ISPF panel 6, issue the following (case-sensitive) command:

   ```
   ogetx '/usr/lpp/ixm/IBM/xml4c3_3_1/lib/IBM' 'SYS1.SIXMMOD1'
   ```

   Ignore the message: `file not found` for the directory entries.

   SYS1.SIXMMOD1 will now contain four files:

   ```
   IXMICUDA
   IXMICUD1
   IXMICUUC
   IXM4C33
   ```

7. From ISPF panel 6, issue the (case-sensitive) command:

   ```
   ogetx '/usr/lpp/ixm/IBM/xml4c3_3_1/lib/IBM/EXP/IXM4C33' 'SYS1.SIXMEXP(IXM4C33)'
   ```

   Dataset SYS1.SIXMEXP will now contain member name IXM4C33. This module will be used by the binder to resolve references to functions and variables. Any application that is to invoke the XML Parser for z/OS and OS/390, C++ Edition must include module IXM4C33.

5.5.1 XML Parser for z/OS and OS/390, C++ Edition, samples

Sample C++ programs that show the use of the SAX and DOM APIs are included with the XML Parser for z/OS and OS/390, C++ Edition. Refer to the sample programs listed in Table 5-5 for examples of using the toolkit's C++ SAX and DOM APIs.

Table 5-5 C++ Parser samples

XML Parser example	Example description
SAXCount	Counts the elements, attributes, spaces and the characters in an XML file
SAXPrint	Parses an XML file and prints it out
DOMCount	Counts the elements, attributes, spaces and characters in an XML file
DOMPrint	Parses an XML document and prints it out
MemParse	Parses XML in a memory buffer outputting the number of elements and attributes
Redirect	Redirects the input stream for external entities
PParse	Demonstrates progressive parsing
StdInParse	Demonstrates streaming XML data from standard input
EnumVal	Shows how to enumerate the markup declarations in a DTD validator
CreateDOMDocument	Displays the XML input file in a graphical tree based interface

Figure 5-4 C++ Parser samples

The gmake utility

Building the XML Parser for z/OS and OS/390, C++ Edition samples requires the use of the gmake utility. If you do not have the gmake utility on your system, use the instructions listed in Appendix A, "GNU gmake" on page 123, to install gmake.

Building the samples for z/OS and OS/390 UNIX Environments

1. Set the root path for the XML Parser for z/OS and OS/390, C++ Edition:

```
export XERCESCROOT=/usr/lpp/ixm/IBM/xml4c3_3_1
```

2. Set up the environment variables required to configure the sample makefiles:

```
unset _CXX_CXXSUFFIX
export CXX=c++
export CXXFLAGS=-2
```

3. Create the makefiles by issuing the **`configure`** command from the samples directory:

```
cd $XERCESCROOT/samples
configure
```

4. Build the C++ parser samples:

```
export _CXX_CXXSUFFIX=cpp
export _CXX_CCMODE=1
gmake
```

Running the samples in z/OS and OS/390 UNIX Environments

1. Set up your LIBPATH to include your XML Parser for z/OS and OS/390, C++ Edition library:

```
export LIBPATH=$XERCESCROOT/lib:$LIBPATH
```

2. Run the sample applications:

```
SAXCount $XERCESCROOT/samples/data/personal.xml
```

Building the samples for z/OS and OS/390 Batch Environments

1. From the ISPF 3.2 panel, allocate a PDS data set using the attributes listed in Table 5-6.
   - We used SYS1 for the high-level qualifier (HLQ).
   - SYS1.SAMPLES.LOAD will be for the sample application programs.

Table 5-6 C++ application samples data set

Data set name	SYS1.SAMPLES.LOAD
3390 cylinders	2
Directory blocks	3
Record format	U
Record length	0
Block size	32760
Data set name type	PDS

2. Set the root path for the XML Parser for z/OS and OS/390, C++ Edition:

```
export XERCESCROOT=/usr/lpp/ixm/IBM/xml4c3_3_1
```

3. Set up environment variables required to configure the sample make files:

```
export LOADMOD=SYS1.SAMPLES.LOAD
export LOADEXP=SYS1.SIXMEXP
export OS390BATCH=1
unset _CXX_CXXSUFFIX
export CXX=c++
export CXXFLAGS=-2
```

4. Create the make files by issuing the **`configure`** command from the samples directory:

```
cd $XERCESCROOT/samples
configure
```

5. Build the C++ parser samples:

```
export _CXX_CXXSUFFIX=cpp
export _CXX_XSUFFIX_HOST=SIXMEXP
export _CXX_CCMODE=1
gmake
```

SYS1.SAMPLES.LOAD will contain the programs listed in Figure 5-5:

```
CRDOMDOC
 DOMCOUNT
 DOMPRINT
 ENUMVAL
 MEMPARSE
 PPARSE
 REDIRECT
 SAXCOUNT
 SAXPRINT
 SAX2CONT
 SAX2PRNT
 STDINPAR
```

Figure 5-5 SYS1.SAMPLES.LOAD members

Running the sample in z/OS and OS/390

To run the SAXCOUNT sample program as a job, update the JOBCARD and submit the sample JCL listed in Figure 5-6:

```
//userjob    YOURJOBCARD
//TEST       EXEC PGM=SAXCOUNT,
//           PARM='//usr/lpp/ixm/IBM/xml4c3_3_1/samples/data/personal.xml'
//STEPLIB    DD   DISP=SHR,DSN=SYS1.SIXMMOD1
//           DD   DISP=SHR,DSN=SYS1.SAMPLES.LOAD
/*
```

Figure 5-6 JCL to run SAXCOUNT sample program

5.6 Java Edition, development and runtime considerations

Special attention should be taken when writing or using Java-developed code to control which level of the parser you are using.

This may be of particular importance whenever you have performance or functional requirements. In an area where technology is moving at a very fast pace, using the latest level of XML parser may yield significant improvements.

5.6.1 SAX and DOM API level

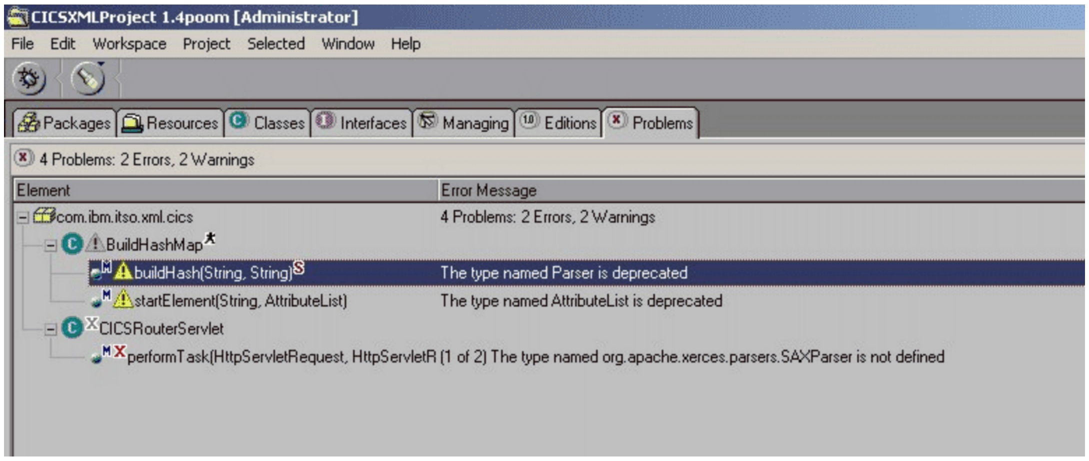
As mentioned in “XML Toolkit components” on page 44, the XML Toolkit is delivered with both level 1.0 and 2.0 APIs.

SAX 1.0 and DOM 1.0 are supported APIs, while SAX 2.0 and DOM 2.0 are not supported since they were still considered experimental at the time version 3.1.1 of the toolkit was built. Since SAX 2.0 and DOM 2.0 are included, level 1.0 APIs have been deprecated. Therefore, this may lead to warning messages being issued when developing Java code using SAX 1.0 or DOM 1.0 APIs.

In summary:

- SAX 1.0 and DOM 1.0 are the supported levels, but for some methods they may produce warning messages of the form: `Type named xxxxxxxxxxxx has been deprecated`; see Figure 5-7 on page 53.

  Refer to the API documentation for each element to verify that the reason for the message is because SAX or DOM 2.0 are included in the toolkit. If this is the case, you can ignore the warning message.
- SAX 2.0 and DOM 2.0 are still experimental and not supported. However, they will not cause any deprecated warning.


CICSXMLProject 1.4poom [Administrator]
File Edit Workspace Project Selected Window Help
Packages Resources Classes Interfaces Managing Editions Problems

4 Problems: 2 Errors, 2 Warnings

Element	Error Message
com.ibm.itso.xml.cics	4 Problems: 2 Errors, 2 Warnings
BuildHashMap	
buildHash(String, String)	The type named Parser is deprecated
startElement(String, AttributeList)	The type named AttributeList is deprecated
CICSRouterServlet	
performTask(HttpServletRequest, HttpServletR	(1 of 2) The type named org.apache.xerces.parsers.SAXParser is not defined

Figure 5-7 Deprecated warnings in VisualAge for Java

5.6.2 WebSphere Application Server considerations

WebSphere Application Server provides its own set of XML Document Structure Services, which contains an XML parser. For our tests, we used WebSphere Application Server 3.5, which includes an XML parser at the XML4J API 2.0.15 level.

However, WebSphere Application Server provides a standard mechanism to allow you to include your own version of the parser for use by your Web application. To do that, you need to include the parser files as part of the Web application classpath for use by your specific Web application.

For the XML parser, define `/usr/lpp/ixm/IBM/XML4J-3_1_1/xerces.jar` in the webapp classpath of your application.

Although possible, making changes at the WebSphere Application Server or system classpath level is not a recommended solution.

5.6.3 VisualAge for Java considerations

VisualAge for Java provides its own XML parser and does not support the ability to use a parser other than the one shipped with it.

For our tests, the XML parser contained within VisualAge for Java version 3.5.3 was at level 2.0.16. Since our goal was to utilize the latest parser available delivered with the XML Toolkit for z/OS or OS/390, we imported the parser into VisualAge for Java to accomplish our testing.

Note: Such an action is *not* supported by VisualAge for Java, and can cause problems within the VisualAge for Java environment. The upcoming WebSphere Studio Application Developer (WSAD) J2EE-compliant server-side Java tool, follow-on to the current VisualAge for Java IDE, will include pluggable parser support. When this is available, the issue will no longer exist.

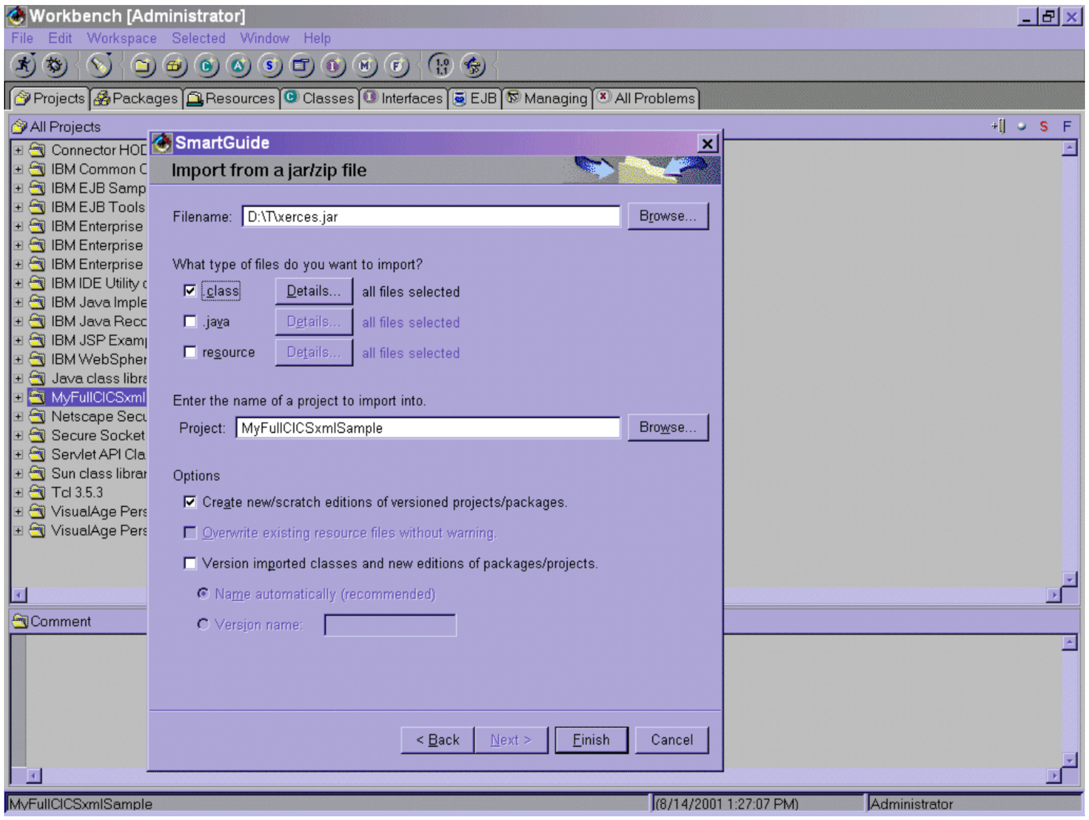
To learn more and test drive the new WebSphere Studio Site Developer and WebSphere Studio Application Developer that build on the WebSphere Studio Workbench technology announced May 30, 2001, refer to the following Web site:

http://www.ibm.com/software/webservers/studio/preregister.html

We used the following procedure to import the XML Java classes from xerces.jar into in VAJ (refer to Figure 5-8):

- Transfer the xerces.jar file from z/OS to the VAJ workstation using ftp, in binary format.

  The file /usr/lpp/ixm/IBM/XML4J-3_1_1/xerces.jar is part of the XML Tookit for z/OS and OS/390 installation.
- Start VisualAge for Java and go to the workbench.
- Select the project.
- Select **Import** from the File menu.
- Select the **Jar File** radio button in the SmartGuide Import dialog and click **Next**.
- In the SmartGuide Import from a jar/zip file dialog, click the **Filename Browse** button and navigate to the directory containing the xerces.jar file.
- Select the xerces.jar file and click **Open** to return to the SmartGuide Import from jar/zip file dialog.
- Ensure that the check box for class is checked.
- Click **Finish**.
- Click **OK**.
- Click **Finish**.


Workbench [Administrator]
File Edit Workspace Selected Window Help
Projects Packages Resources Classes Interfaces EJB Managing All Problems
All Projects
Comment
SmartGuide
Import from a jar/zip file
Browse...
Details...
< Back
Next >
Finish
Cancel

Filename: D:\T\xerces.jar

What type of files do you want to import?

- [✓] .class — all files selected
- [ ] .java — all files selected
- [ ] resource — all files selected

Enter the name of a project to import into.

Project: MyFullCICSxmlSample

Options

- [✓] Create new/scratch editions of versioned projects/packages.
- [ ] Overwrite existing resource files without warning.
- [ ] Version imported classes and new editions of packages/projects.
  - ● Name automatically (recommended)
  - ○ Version name:

MyFullCICSxmlSample (8/14/2001 1:27:07 PM) Administrator

Figure 5-8   Import xerces.jar into Visual Age for Java

6

Application integration on z/OS and OS/390

XML being interoperable, self-defining, easily understandable, and Unicode-based makes it a perfect choice for integrating applications. Since XML is merely a data format, applications are responsible for creating, transmitting, and processing XML documents. Applications invoke parsers to process the XML documents.

The parser checks the well-formedness of the document. If a DTD is provided, it ensures that the document matches the rules laid out therein. The document is converted into Unicode if it is encoded differently. The applications can extract data stored in the document by using one of two standard APIs: Document Object Model (DOM), and Simple API for XML (SAX).

XML can be considered as the base technology to be used for application integration. To make its usage simpler, different products have begun to provide facilities to integrate applications using this young and dynamic technology.

In this chapter we present several XML-to-back office scenarios.

6.1 DB2 XML Extender

DB2 Extender provides new data types, functions, and stored procedures which enable you to use SQL as an access method for processing and producing XML documents. Using the DB2 Extender Document Access Definition scheme, XML documents are mapped to relational data. When a database is enabled for XML, DB2 Extender creates a Document Type Definition reference table.

DB2 XML Extender provides two methods for managing XML documents with DB2:

1. The XML Columns method enables you to store XML documents into DB2 columns that are enabled for XML. The documents can be retrieved, searched, and updated. Elements and attributes can be mapped to DB2 tables, allowing for fast, indexed searches.
2. The XML Collections method enables you to produce XML documents from existing DB2 data, or produce DB2 data from XML documents.

DB2 V7 for OS/390 is required for the DB2 XML Extender. For more information on using DB2 XML Extender, refer to *Integrating XML with DB2 XML Extender and DB2 Test Extender,* SG24-6130.

6.2 XML and CICS integration

CICS TS Version 1.3 is the first CICS release to include support for Java programs running within CICS. Java programs within CICS can run under a Java Virtual Machine (JVM) or be compiled using the High Performance Java option of VisualAge for Java.

JVM and HPJ have their own TCB; they do not rely on the CICS TCB. A Java transaction runs under control of a JVM that uses its own TCB (called a J8 TCB).

To use the Java version of the XML parser within CICS TS, the Java program must run under the JVM. Since each Java transaction would traverse the Java Native Interface (JNI) when running CICS TS V1.3, it is recommended that the Java parser be used for long-running transactions only in order to amortize the CPU cost over a longer usage. CICS TS Version 2 is the recommended level to use, since it includes Java performance improvements such as persistent reusable JVMs and enhanced garbage collection.

At this time, there is no parsing capability for second generation languages (like COBOL.) IBM is developing a set of APIs and parsing capability for the COBOL environment, but it is not generally available.

Though it is possible to invoke a C++ parser from a COBOL program, the XML C++ parser is not supported within a CICS transaction; I/Os to access non-CICS managed external data files would disrupt CICS pseudo-TCB management.

The next section illustrates a sample Web-to-CICS back-end application using IBM WebSphere and CICS Transaction Gateway.

6.2.1 A Web-enabled CICS Application, using WebSphere and CTG

A sample browser to CICS TS application is shown in Figure 6-1:

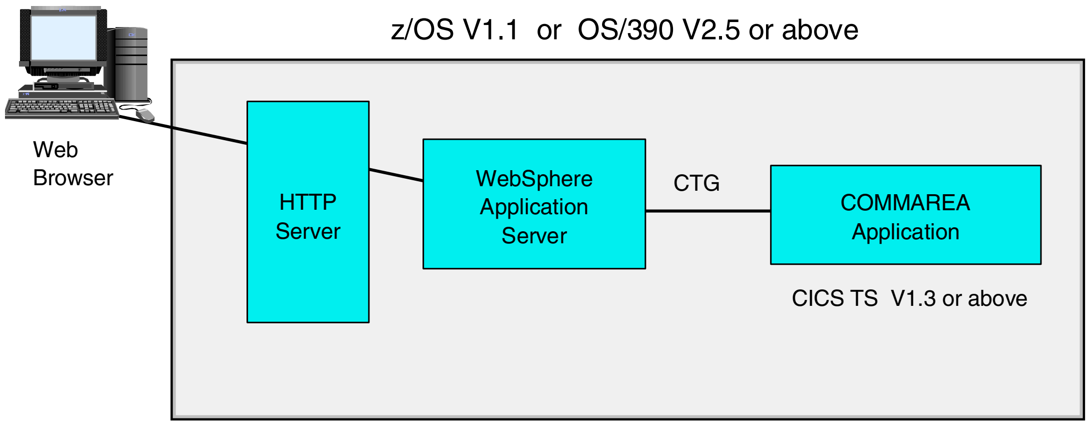


Figure 6-1 Browser-to-back-end CICS application

Description

1. An HTTP request is sent from the client to the Web Server.
   - For applets, the WebSphere Application Server returns an HTML page with embedded applet tags to the browser. The browser requests download of the applets and appropriate Java classes. The applet uses the JavaGateway class to set up a connection to the gateway process through one of four supported network protocols (TCP/IP sockets, SSL sockets, HTTP or HTTPS.) The ECIRequest class is used to send a request to the CICS Transaction Server.
   - For servlets, the servlet is invoked by the HTTP request either through a URL, a form, or server side include. The servlet runs within the JVM of the WebSphere Application Server servlet engine. The servlet uses the JavaGateway class to set up the local gateway connection. The servlet then uses the `ECIRequest` class to send a request to the CICS Transaction Server.
2. For applets or servlets, the CICS Transaction Gateway builds an EXCI request, uses the Java Native Interface (JNI) to call the CICS External Communication Interface (EXCI), and sends the request to the CICS Transaction Server.
3. The EXCI sends the request with the COMMAREA to CICS through XCF or MRO.
4. The EXCI response returns to the CICS Transaction Gateway.
   - When the request was issued from a servlet, the CICS Transaction Gateway returns the response to the servlet.
   - For an applet, the CICS Transaction Gateway sends the response back to the browser.

Session state data

WebSphere Application Server Standard Edition provides support for managing HTTP session state data. Shared session state objects can be used with a Web server running in scalable server mode in a Parallel Sysplex.

Security

- For the servlets, SSL support is offered by the HTTPS support in the Web Server. Support includes externally signed certificates and client authentication.

- For applets, only self-signed certificates are supported. There is no support for client certificates.

Benefits

- CICS Transaction Gateway supports many network protocols and application architectures.
- CICS Transaction Gateway is part of the Common Connector Framework (CCF).

Considerations

- On OS/390, CICS Transaction Gateway includes support for the ECI interface. The OS/390 CICS Transaction Gateway does not support the EPI interface.
- The maximum message size per transmission is 32 K. For data sizes that exceed 32 K, the application would need to make multiple calls to transmit the data.
- The length of the COMMAREA being transmitted is one factor of throughput.

Software requirements

For up-to-date information on functions and software products level for e-business applications on z/OS and OS/390, refer to the z/OS and OS/390 Planning Wizard for e-business Web page:

http://www.ibm.com/servers/eserver/zseries/zos/wizards/ebiz/

6.3 XML-to-back office applications

6.3.1 Servlet parsing, CICS Transaction Gateway and CICS application

Figure 6-2 shows XML as an external input to the application. The XML input document is processed by a WebSphere servlet. The servlet invokes the XML parser and formats the COMMAREA to be passed to the CICS application.

There are no changes to the CICS back-end application.

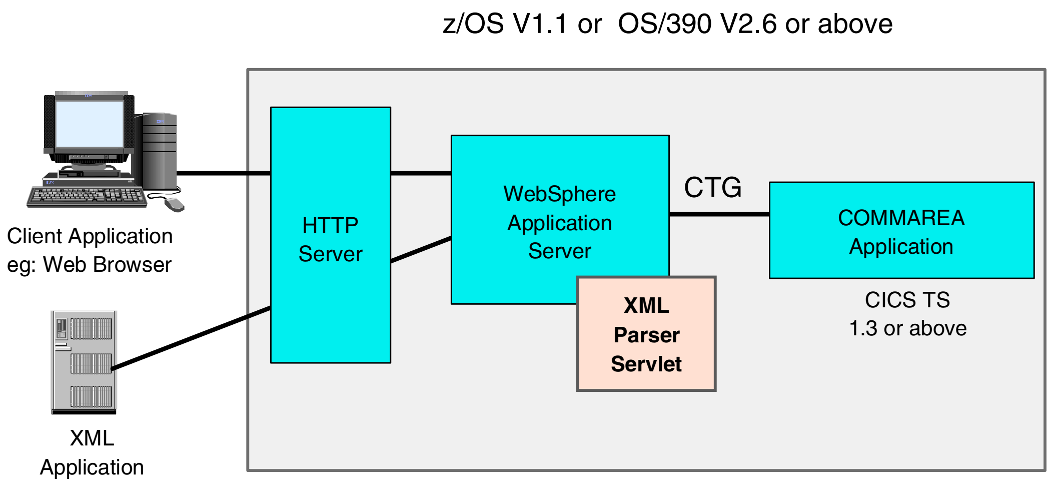

z/OS V1.1 or OS/390 V2.6 or above
Client Application eg: Web Browser
XML Application
HTTP Server
WebSphere Application Server
XML Parser Servlet
CTG
COMMAREA Application
CICS TS 1.3 or above

Figure 6-2 CICS back-end application with XML parser

Description

1. An HTTP request is sent from the XML client to the Web server.
2. A servlet is invoked by the HTTP request. The servlet runs within the JVM of the WebSphere Application Server servlet engine. The servlet uses the parser APIs to transform the source XML document and the JavaGateway class to set up the local gateway connection.
3. The servlet uses the ECIRequest class to send the request to the CICS Transaction Server. The request with COMMAREA is sent to CICS through XCF or MRO.
4. The EXCI response returns to the CICS Transaction Gateway.
5. The servlet uses the parser APIs to build an XML document from the returned COMMAREA.
6. The servlet sends the XML response to the client.

Session state data

WebSphere Application Server Standard Edition provides support for managing HTTP session state data. Shared session state objects can be used with a Web server running in scalable server mode in a Parallel Sysplex.

Security

- For servlets, SSL support is offered by the HTTPS support in the Web server. Support includes externally signed certificates and client authentication.

Benefits

- B2B Web services are added without modification to the back-office application.
- It can use extended units of work.
- CICS Transaction Gateway is part of the Common Connector Framework. It supports many network protocols and application architectures.

Considerations

- Standardization of XML tags required for element reference.
- On OS/390, CICS Transaction Gateway includes support for the ECI interface. The OS/390 CICS Transaction Gateway does not support the EPI interface.
- The maximum COMMAREA size passed to a CICS application is 32 K, but the total length of information transmitted to the application can exceed 32 K. For data sizes that exceed 32 K, the application can make multiple calls to transmit the data.

  Another solution can also be to translate one XML-tagged message into one unit of work containing multiple commarea-based transactions.
- The total length of data being transmitted is one factor that affects the overall application throughput.

Software requirements

- CICS Transaction Gateway (the release depends on the version and release of CICS TS and the version of JDK running WebSphere Application Server)
- XML Toolkit for z/OS and OS/390 1.2
- More information on software levels is available on the z/OS and OS/390 Planning Wizard for e-business Web page:

  `http://www.ibm.com/servers/eserver/zseries/zos/wizards/ebiz/`

6.3.2 CICS XML parsing, CICS application

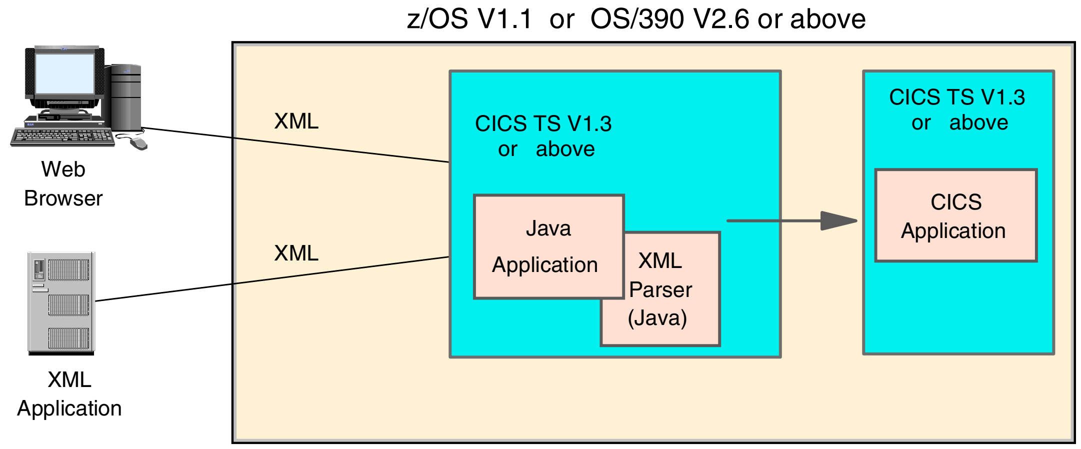


Figure 6-3 XML parser within CICS

Description

1. An HTTP request is sent from the client to the CICS Server.
2. The CICS Transaction Server Sockets domain provides TCP/IP support to handle HTTP and IIOP requests.
3. The front-end program is responsible for parsing and returning XML-tagged data.

Session state data

The CICS Transaction Server “Web-aware front-end” is responsible for managing session state data. These can include DB2, VSAM, TD/TS queues.

Security

- CICS supports SSL natively-defined in TCP SERVICE RDO definition.

Benefits

- Enables two-tier Web-to-host implementation
- Multiple calls to transactions coordinated by CICS as one update
- Solution based on CICS-only skills

Considerations

- Standardization of XML tags is required for element reference.
- The maximum message size per transmission is 32 K. For data sizes that exceed 32 K, the application would need to make multiple calls to transmit the data.
- The maximum COMMAREA size passed to a CICS application is limited to 32 K, but the total length of information being transmitted from the browser (or the application) to CICS can exceed 32 K.
- The JVM in CICS TS is not reusable until CICS TS V2.
- The total length of data being transmitted is one factor that affects the overall application throughput.

6.4 XML MQSeries integration

MQSeries products are building blocks for integrating applications. MQseries provides assured message delivery across numerous platforms, supports a variety of communication protocols, and provides standard connectors to products such as CICS and IMS.

With MQSeries Integrator you can route, transform, filter, and warehouse messages in accordance with business-defined rules. MQSeries Integrator includes a framework for extending its functionality.

User-written or third-party plug-ins for specific requirements can be used as MQSeries extensions. Today, user-written XML parser plug-ins can be used to transform XML messages to back office expected formats. Future enhancements to MQSeries will provide facilities for the construction and parsing of XML messages.

For more information on integrating XML with MQSeries, refer to *Business Integration Solutions with MQSeries Integrator,* SG24-6154.

6.4.1 Servlet parsing, MQ, CICS application

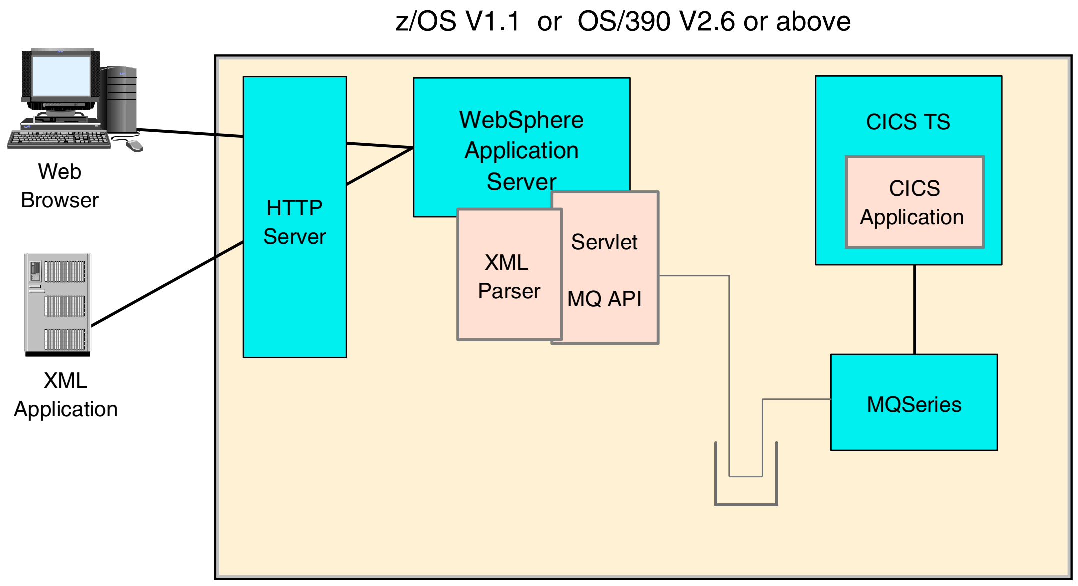


Figure 6-4 Servlet XML parser with MQ API

Description

1. An HTTP request is sent from the XML client to the Web Server.
2. The servlet is invoked by the HTTP request; it runs within the JVM of the WebSphere Application Server servlet engine. The servlet uses the parser APIs to transform the source XML document.
3. The servlet uses the MQSeries class to send the request to the CICS Transaction Server.
4. The servlet uses the parser APIs to build an XML document from the returned MQSeries data.
5. The servlet sends the XML response to the client.

Session state data

WebSphere Application Server Standard Edition provides support for managing HTTP session state data. Shared session state objects can be used with a Web server running in scalable server mode in a Parallel Sysplex.

Security

- For the servlets, SSL support is offered by the HTTPS support in the Web Server. Support includes externally signed certificates and client authentication.

Benefits

- B2B Web services added without modification to the back office application.

Considerations

- Standardization of XML tags is required for element reference.
- The maximum message size per transmission can exceed 32 K without the need to make multiple calls to transmit the data, as do applications that use the CICS Transaction Gateway.

6.4.2 Servlet, MQ Integrator (parsing), CICS application

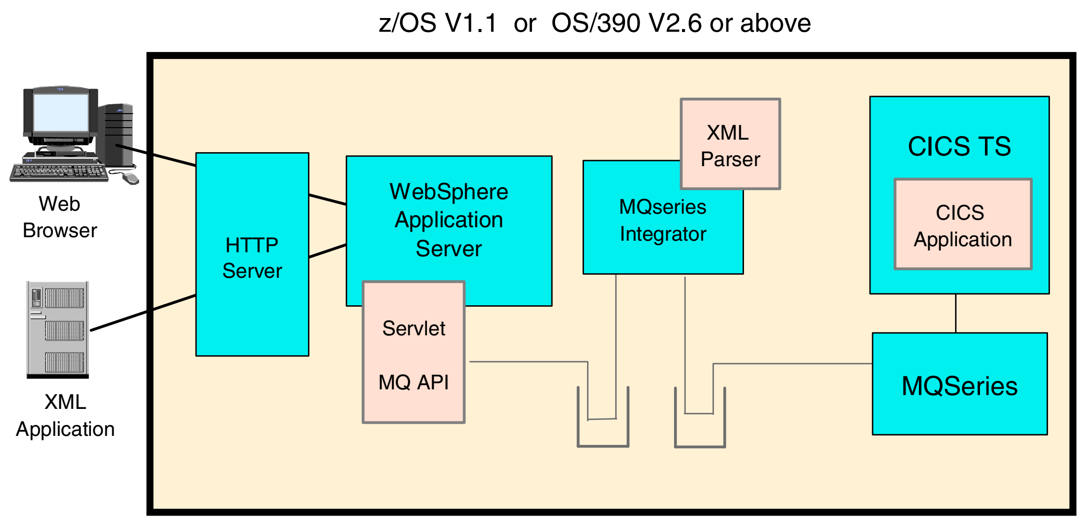


Figure 6-5 Java Servlet for MQ message (MQSI parser plug-in)

Description

1. An HTTP request is sent from the XML client to the Web Server.
2. The servlet is invoked by the HTTP request; it runs within the JVM of the WebSphere Application Server servlet engine. The servlet uses the MQSeries class to send the request to the MQSeries Server.
3. MQSeries Integrator calls the application plug-in to parse and transform the XML message. MQSeries delivers the transformed message to the back-end application that processes the request and sends an MQ message response.
4. The servlet sends the XML response to the client.

Session state data

WebSphere Application Server Standard Edition provides support for managing HTTP session state data. Shared session state objects can be used with a Web server running in scalable server mode in a Parallel Sysplex.

Security

- For servlets, SSL support is offered by the HTTPS support in the Web Server. The support includes externally signed certificates and client authentication.

Benefits

- B2B Web services added without modification to back office application.
- Message transformation is separated from servlet application.

Considerations

- Standardization of XML tags is required for element reference.
- The maximum message size per transmission can exceed 32 K without the need to make multiple calls to transmit the data, as do applications that use the CICS Transaction Gateway.

6.5 XML and IMS integration

6.5.1 Servlet, IMS ITOC, IMS application

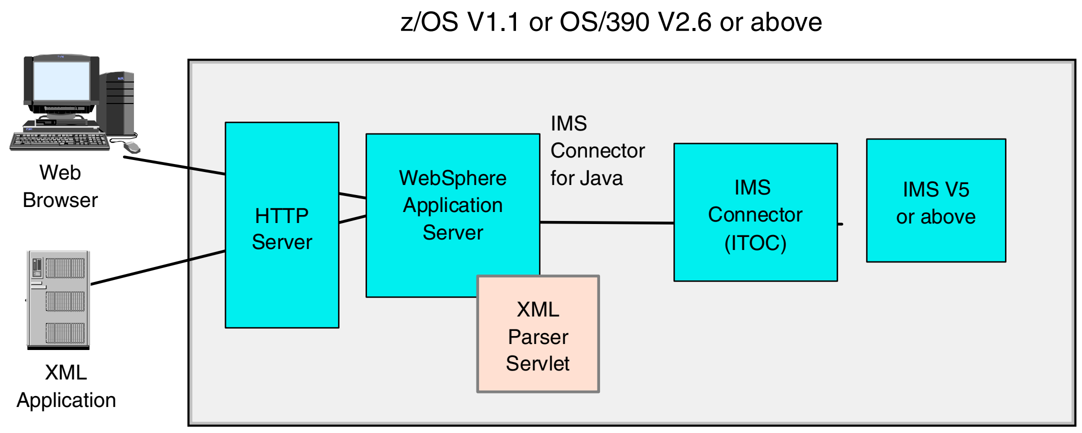


Figure 6-6 XML parser servlet with IMS Connector

Description

1. An HTTP request is sent from the XML client to the Web Server.
2. The servlet is invoked by the HTTP request; it runs within the JVM of the WebSphere Application Server servlet engine. The servlet uses the XML parser APIs to transform the source XML document. The servlet uses the Java IMS connector to access the IMS back office application
3. The servlet uses the XML parser APIs to build an XML document from the returned IMS data.
4. The servlet sends the XML response to the client.

Session state data

WebSphere Application Server Standard Edition provides support for managing HTTP session state data. Shared session state objects can be used with a Web server running in scalable server mode in a Parallel Sysplex.

Security

- SSL support is offered by the HTTPS support in the Web Server. Support includes externally signed certificates and client authentication.

Benefits

- B2B web services added without modification to back office application.

Considerations

- Standardization of XML tags is required for element reference.

7

CICS TS applications and XML

In this chapter, we discuss the use of CICS TS applications and XML, and provide scenarios.

7.1 CICS TS overview

CICS TS is the IBM general-purpose online transaction processing (OLTP) software. It is a powerful application server that runs z/OS, OS/390 and VSE systems. It is flexible enough to meet your transaction processing needs, whether you have thousands of terminals or a client/server environment with workstations and LANs exploiting modern technology such as graphical interfaces or multimedia.

It takes care of the security and integrity of your data while looking after resource scheduling, thus making the most effective use of your resources. CICS seamlessly integrates all the basic software services required by OLTP applications, and provides a business application server to meet your information processing needs of today and the future.

The CICS strategy is to enable customers to quickly and easily Web-enable their existing applications—without requiring major changes to those applications or forcing specific Web architectures—since each user has their own starting point, architectural requirements driven by particular business needs, and different objectives for the end solution. CICS helps users to gradually exploit the new Java-based technologies, while gaining real business advantage from each step along the way, and it provides a number of different Web-enabled solutions to meet these different requirements, as follows:

- Direct access to CICS without the need for intermediate gateways or Web servers
- Connection to CICS through the IBM WebSphere Application Server for OS/390
- 3-tier solutions via intermediate Web servers and the CICS Transaction Gateway
- Access to existing CICS 3270 applications without changing any existing code
- Customizing end-user interfaces to provide a Web-style look and feel to the application
- The ability to connect from Java applets and servlets, allowing a more interactive end user interface
- Standard CORBA clients using standard IIOP protocols for true distributed object programming

7.2 CICS Transaction Gateway-WebSphere Application Server on z/OS or OS/390 scenario

The CICS Transaction Gateway is a set of client and server software components incorporating the facilities of the CICS Universal Client that allow a Java application to invoke services in a CICS region. The Java application can be an applet, a servlet, an Enterprise Java Bean (EJB), or any other type of Java application.

It is possible to use WebSphere Application Server in conjunction with a remote CICS Transaction Gateway daemon on z/OS or OS/390, as shown in Figure 7-1 on page 67. In this case, the ECI request flows from the servlet to the CICS Transaction Gateway on z/OS or OS/390 using one of the CICS Transaction Gateway network protocols.

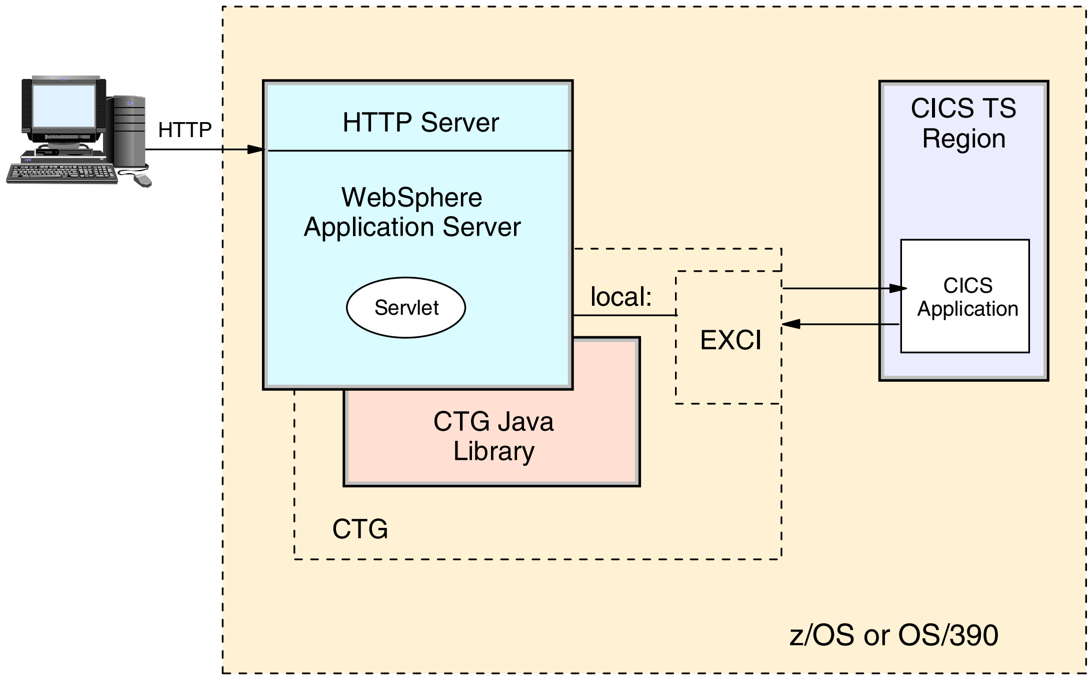


Figure 7-1 WebSphere Application Server and CICS Transaction Gateway daemon on z/OS

On z/OS or OS/390, the servlet environment is provided by WebSphere Application Server. This provides both the IBM HTTP Server and the servlet engine.

The CICS Transaction Gateway daemon is not usually required when running servlets within the WebSphere Application Server, since the CICS Transaction Gateway local: protocol can be used to directly invoke the native functions in the underlying EXCI. The EXCI passes the request onto the attached CICS region, which is not aware that the request comes from a Java application.

Since the CICS Transaction Gateway uses the CICS EXCI protocol, the CICS region and WebSphere Application Server for z/OS or OS/390 can be situated on any machine in a Parallel Sysplex. However, the servlet is limited to using the subset of the ECI methods supported by the EXCI; it cannot use the external presentation interface (EPI) or external security interface (ESI) and it must communicate with a CICS TS region V1.2 or above.

7.2.1 External Call Interface (ECI)

The ECI enables a non-CICS client application to call a CICS program synchronously or asynchronously, as a subroutine. The client application communicates with the server CICS program, using a data area called a COMMAREA. The COMMAREA is passed to the CICS server on the call, and the CICS program typically populates it with data accessed from files or databases. The data is then returned to the client for manipulation or display.

The CICS Transaction Gateway, which includes the function of the CICS Universal Client, supports concurrent ECI calls to a CICS server with no restrictions on communication protocols or functions, whether the calls are to the same or different CICS systems.

The number of concurrent ECI calls is controlled by the RDO session definition in CICS, and the number of concurrent connections allowed to the WebSphere address space (this limit is currently set to 100 per CICS region in z/OS V1.1).

ECI is the recommended interface for developing new client/server applications. Its call structure easily divides the presentation logic (usually in the client) from the business logic in the CICS application, offering application designers maximum flexibility.

CTG uses Java ECI classes that map onto the native CICS Universal Client ECI support for your platform. The Java classes are platform-independent. Therefore, you can write your Java classes on one platform and run them on another.

7.2.2 Common Connector Framework and Enterprise Access Builder

Aside from coding the basic CICS Transaction Gateway Java methods, you can develop a CICS Transaction Gateway application using the IBM Common Connector Framework Java Beans. The IBM Common Connector Framework provides the following:

- A common client programming model for connectors. These interfaces allow VisualAge for Java Enterprise Access Builder for transactions to easily build applets or servlets to access programs or transactions in a CICS region.
- A common infrastructure programming model for connectors, which gives a component environment, such as WebSphere, a standard view of a connector, and vice versa.

When developing an applet or servlet using the Common Connector Framework CICS connector, the `CICSConnectionSpec`, and `ECIInteractionSpec` or `EPIInteractionSpec` classes are used. These classes can be specified in an Enterprise Access Builder command with an input and output COMMAREA to invoke a CICS program.

For more information on developing Java applications using Common Connector Framework, refer to *CCF Connectors and Database Connections Using WebSphere Advanced Edition,* SG24-5514.

IBM VisualAge for Java is usually used for developing Java programs. VisualAge for Java supports the CCF and also provides the Enterprise Access Builder and the Java Record Framework. This provides a means to analyze CICS COBOL COMMAREA structures or BMS maps, and generates the Common Connector Framework Java Beans to represent input and output records. VisualAge for Java can then be used to visually develop applications and applets.

In the Java Record Framework, a record wraps a data structure, such as a COBOL copybook. A record is a logical collection of application data elements that combines the actual record bytes together with its type. A record is used to implement the input and output of an Enterprise Access Builder command. The Enterprise Access Builder supplies the Record Type Editor tool that allows you to create a record type, or modify existing records as shown in Figure 7-2 on page 69.

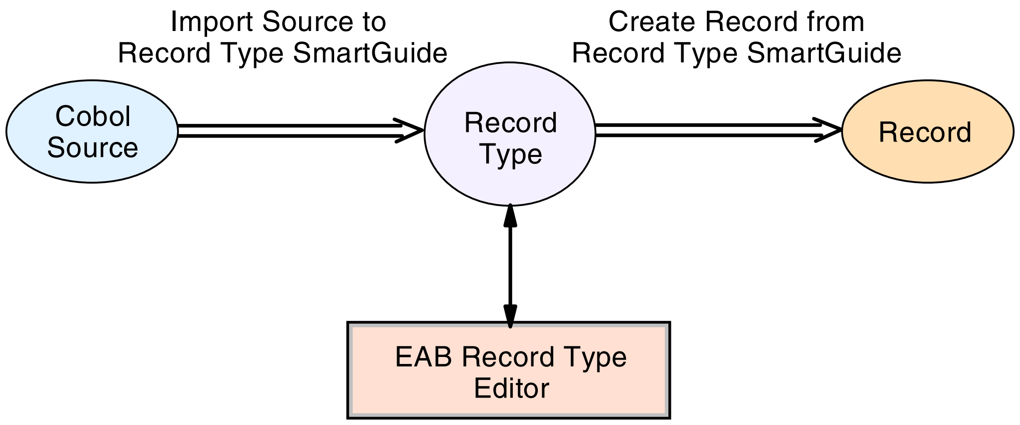

Import Source to Record Type SmartGuide
Create Record from Record Type SmartGuide
Cobol Source
Record Type
Record
EAB Record Type Editor

Figure 7-2 The Enterprise Access Builder record type editor

The Record COBOL Typeset extends the Record Framework to support COBOL structures; this is particularly useful for managing CICS COMMAREAs, which are usually handled as COBOL structures. You can use the Enterprise Access Builder tools to import a COBOL structure to produce a matching record type. Then, from this record type, a record can be produced. This record provides accessors for the fields of the COBOL structure and data conversion of the fields from Java types to COBOL types.

7.3 CICS TS support for Java

CICS TS 1.3 supports the Java programming language. It provides a library of Java classes for access to the important CICS services. A CICS program written in Java can be run under a JVM, or compiled to machine code and executed as a Language Environment (LE) run unit. DB2 can also be accessed from compiled Java programs through the SQLJ interface.

The CICS TS support for Java:

- Allows a CICS program to be written in Java
- Allows CICS Java programs to interoperate with CICS programs written in any other CICS-supported languages
- Allows a CICS application written in Java to access CICS services
- Provides a Java version of the CICS API known as the JCICS class library
- Supports Java development tools such as VisualAge for Java

The CICS translator is not required for Java. The application program is developed and compiled using a Java compiler on a workstation or under z/OS or OS/390 UNIX System Services. The bytecode output of this process may be interpreted by a JVM running in CICS, or it may be further compiled to S/390 machine code by the HPJ compiler provided by the ET/390 component of VisualAge for Java

7.3.1 CICS support for compiled Java code

The HPJ compiler runs under z/OS or OS/390 UNIX System Services and creates a program object that is stored in a partitioned data set extended (PDSE) library. Run-time support for Java programs compiled to machine code is similar to the CICS language support for COBOL or C++.

CICS loads the program from the PDSE on first invocation and executes it as an LE run unit. The normal CICS program execution model is used, in place of a long-lived JVM.

To be eligible for execution under CICS without the JVM, the program must not invoke z/OS UNIX system services directly. It must use JCICS classes for such purposes.

DB2 access is also enabled by using SQLJ, a standard API for static access to DB2 from a Java program. SQLJ requires a preprocessor and uses the same binding mechanism as SQL. It is supported by HPJ and runs under CICS in the same way as standard SQL embedded in a COBOL program. It is compatible with the current DB2 support in CICS, using the same CICS DB2 attachment facility and sharing the same database if required.

7.3.2 CICS support for the JVM

The JVM that runs a CICS Java program is the standard z/OS JVM. Transactions invoking CICS Java programs that run with the JVM are executed on their own z/OS or OS/390 TCB. When such a program uses CICS services, the CICS dispatcher switches, if necessary, to the CICS quasi-reentrant TCB to ensure appropriate serialization so there is no interference with non-Java CICS Transactions.

Programs that use Java classes that directly invoke system services, such as I/O operations, must use JVM execution mode.

7.4 XML support in CICS

There are two ways to use XML with CICS:

- Parsing XML within Java Application running as CICS transaction in CICS Transaction Server
- Parsing XML within Java Application running as servlet in WebSphere Application Server

7.4.1 Parsing XML in CICS TS

There is a Java parser for XML available that can be used with CICS TS V1.3 and above. The Java XML parser can be run in the CICS environment using the integrated Java Virtual Machine (JVM) in CICS Transaction Server. The application should not be compiled using the High Performance Java Option of Visual Age for Java.

Consider CICS TS Version 2 for its significant performance improvements in support of Java.

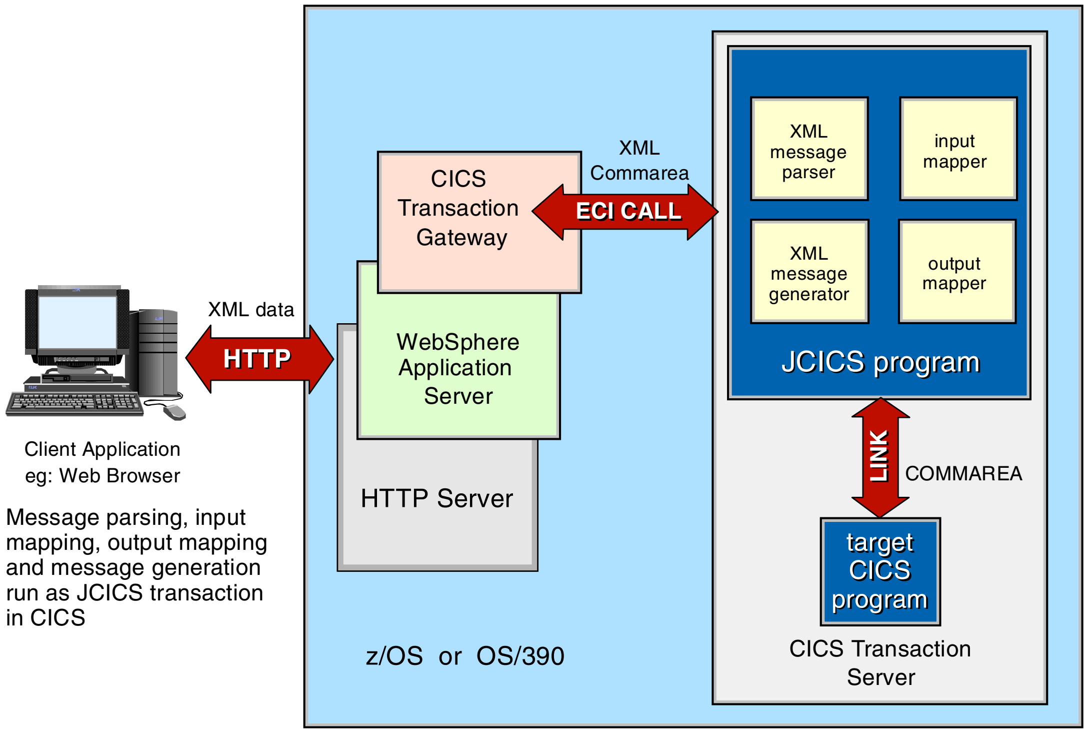


Figure 7-3 Parsing XML in CICS Transaction Server

Scenario 1 - parsing inside CICS transaction

Following is the flow of this scenario:

- The XML source is passed through an HTTP request to the Web server.
- The servlet invoked by the HTTP request runs within the JVM of the WebSphere Application server servlet engine.
- The Java CICS application is invoked using Common Connector Framework CICS connectors by passing the XML document as a COMMAREA. A Java Gateway connection is established and the ECI request is flowed to the CICS Transaction Gateway.
- The CICS Transaction Gateway builds an EXCI request and calls the CICS External Communications Interface (EXCI) to send the request to the CICS Transaction Server. The EXCI sends the request with the COMMAREA to CICS through an XCF or MRO connection.
- The message parser in the Java CICS transaction parses the XML source to get the required data.
- The input mapper populates the parsed data into the input COMMAREA record of the target CICS transaction.
- A LINK call is made to the target CICS application with the populated input COMMAREA.
- The output mapper gets the data from the COMMAREA and the message generator and builds the XML message to be returned to the user.
- The EXCI returns the response to the CICS Transaction Gateway, which then prepares the response and sends it back to the middle-tier Java application.

At this time, there is no parsing capability using native COBOL. It is possible to invoke a C++ parser from a COBOL program, but the C++ parser does not run in the CICS TS environment. Also, if you need to invoke the C++ parser from a COBOL application outside CICS, you should set up a glue routine to do the invocation; otherwise, errors from the parser cannot be surfaced to the COBOL application.

Benefits

- This architecture is suitable in a B2B scenario in which an industry conforms to a standard XML message format for communication and data transfer. In that case, every service provider adheres to this format and hence the application running in the transaction server can be designed to accept this XML message as input, parse it to get the required data, and perform the transaction.
- The middle-tier application in the Web server is independent of the transaction. It can be purely an XML-based application and hence can easily be extended to support many transactions.

Disadvantages

- While doing an ECI call, the size of the COMMAREA is restricted to 32 K and hence the XML document size cannot exceed this limit.
- An alternative is to use IIOP to invoke the CICS transaction. CICS TS 1.3 supports IIOP for only HPJ-compiled applications, which currently doesn't support XML parsers. In CICS TS 2.1, the Java application in CICS should run in the integrated JVM in order to support IIOP.
- If the middle tier is on z/OS or OS/390, data conversion is not required. But if the middle tier is on platforms other than z/OS or OS/390, code page translation might be required for the XML data passed.

7.4.2 Parsing outside CICS TS

Parsing and conversion of an XML message to COMMAREA can be done outside the CICS environment using either the Java or the C++ parser.

A recommended way of doing this is to use a Java Server Page (JSP) or a servlet running under WebSphere Application Server, which passes the COMMAREA to a CICS transaction using the CICS Transaction Gateway. Responses can be sent out via XML using the reverse path.

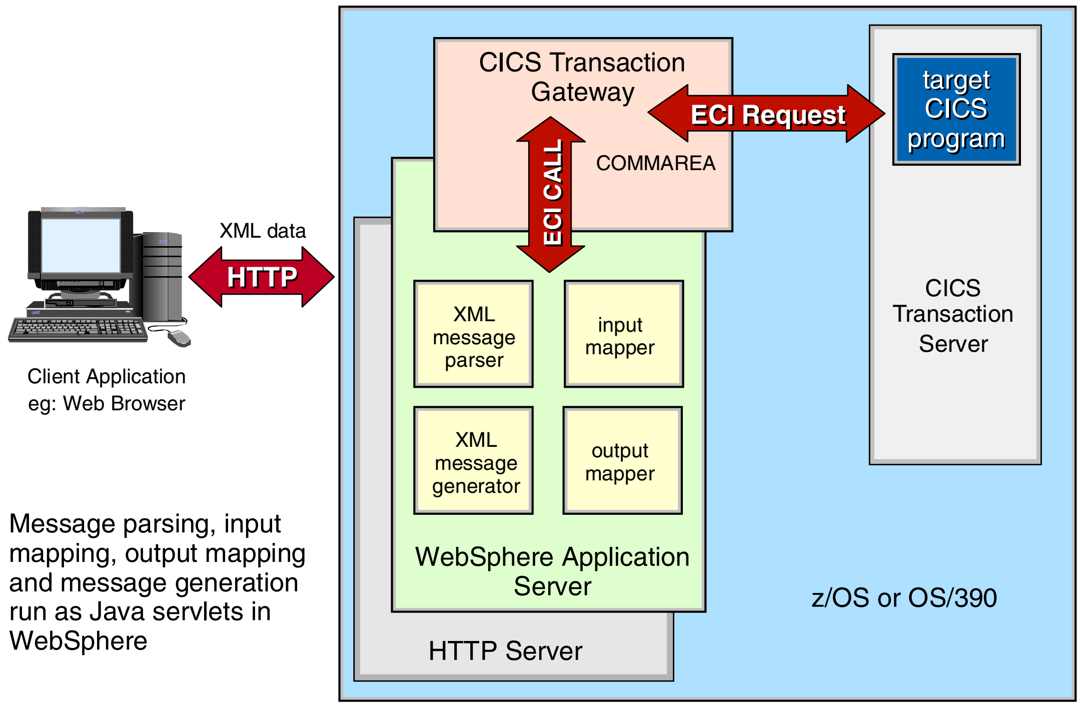


Figure 7-4 XML parsing using a JSP or a Servlet with WebSphere Application Server

Scenario 2 - Parser outside CICS application

This scenario has the following flow:

- The XML source is passed through an HTTP request to the Web server.
- The servlet invoked by the HTTP request runs within the JVM of the WebSphere Application server servlet engine.
- The message parser parses the XML source to get the required data.
- The input mapper populates the parsed data into the input COMMAREA record of the transaction.
- The CICS application is invoked using CCF CICS connectors. This is done by executing the command created for the transaction with appropriate connection and interaction specifications.
- A connection is established and the ECI request is flowed to the CTG.
- The CTG builds an EXCI request, and calls the CICS External Communications Interface to send the request to the CICS Transaction Server. The External Communications Interface sends the request with the COMMAREA to CICS through an XCF or MRO connection. The External Communications Interface returns the response to the CICS Transaction Gateway, which prepares the response and sends it back to the middle-tier Java application.
- The output mapper gets the data from the COMMAREA and the message generator, which can be even a Java Server Page, and builds the XML message to be returned to the user.

Benefits

- Quick development and implementation.

- Reusability of existing Web-enabled application since they can be easily extended to accept the XML data to the middle tier.

Drawbacks

- The middle tier needs to have the intelligence of the data format expected by the backend application.
- Mappers are specific to the interface for each transaction. Every time the backend application changes, the corresponding mapper needs to be adjusted to the change.
- Since ECI can support a COMMAREA with a maximum size of 32 K, the size of data extracted from the XML document is restricted to 32 K.

7.4.3 XML Parser Java Edition and code page conversion

Java programs executing on z/OS or OS/390 and using the CICS Transaction Gateway ECI methods deserve a special mention because of the code page considerations on z/OS or OS/390.

The most common encodings (character encoding schemes) use a single byte per character and they are called single-byte character sets (SBCS). They are all limited to 256 characters. Because of this, none of them can even cover all of the accented letters for the Western European languages.

ASCII, the American Standard Code for Information Interchange, is a character set using 7-bit units, with a trivial encoding designed for 7-bit bytes. It is the most important character set out there, despite its limitation to very few characters. ASCII provides only 128 numeric values, and 33 of those are reserved for special functions.

EBCDIC, the Extended Binary-Coded Decimal Interchange Code, is an encoding format using 8-bit bytes. Many different encodings were created over time to fulfill the needs of different Western languages. In z/OS and OS/390, the default is EBCDIC code page Cp1047.

037	USA, Canada
273	Germany, Austria
277	Denmark, Norway
278	Finland, Sweden
280	Italy
284	Spain, Latin America
285	United Kingdom
297	France
500	International
871	Iceland
1047	Latin 1, used by z/OS and OS/390

The most widely used SBCS encoding today, after ASCII, is ISO-8859-1. It is an 8-bit superset of ASCII and provides most of the characters necessary for Western Europe. A modernized version, ISO-8859-15, also has the euro symbol and additional French and Finnish letters.

The Unicode standard is an industry standard and parallels ISO 10646-1. Both standards have the same character repertoire and the same encoding forms and schemes.

The standards differ as to the kind of information they provide: the Unicode standard provides more character properties and describes algorithms etc., while the ISO standard defines collections, subsets and so on. The standards are synchronized and the respective committees work together to add new characters and assign code point values.

Unicode provides a single character set that covers the languages of the world, and a small number of machine-friendly encoding forms and schemes to fit the needs of existing applications and protocols. It is designed for best interoperability with both ASCII and ISO-8859-1, the most widely used character sets, to make it easier for Unicode to be used in applications and protocols.

For internal processing, the Unicode standard defines three encoding forms, optimized for a variety of different uses:

- UTF-16, the default encoding form, maps a character code point to either one or two 16-bit integers.
- UTF-8 is a byte-based encoding that offers backward compatibility with ASCII-based, byte-oriented APIs and protocols. A character is stored with 1, 2, 3, or 4 bytes.

  UTF-8 is used to meet the requirements of byte-oriented, ASCII-based systems. It maintains transparency for all of the ASCII code values (0..127). Thus, ASCII text is also UTF-8 text.
- UTF-32 is the simplest but most memory-intensive encoding form, it uses one 32-bit integer per Unicode character.

For more information on Unicode, refer to the following Web site:

`http://www.unicode.org`

Java, XML parsers and most other modern systems provide Internationalization solutions based on Unicode.

Java strings are stored in Unicode, while Java byte arrays are stored in the native code page of the operating system (EBCDIC code page Cp1047 on z/OS and OS/390), unless a specific code page is explicitly specified.

The XML standard defines strict encoding rules. If the document is not in UTF-8 or UTF-16, the encoding of the document must be specified via the `encoding=` attribute on the XML processing instruction.

```
<?xml version="1.0" encoding="ebcdic-cp-us" ?>
```

The COMMAREA passed to CICS in an `ECIRequest` is a Java byte array, but is often converted to or from a string, for handling within the Java code. It is common that CICS applications expect an encoding other than the default Cp1047. You may have to specify another EBCDIC encoding, such asCp037 or Cp500, when the CICS application does not use the default operating system encoding.

On any platform, including z/OS or OS/390:

- The parser is capable of handling XML documents in many different encodings, as specified in the XML processing instruction. When the XML document is not encoded in UTF-8 or UTF-16, you must specify the correct encoding in the XML processing instructions.
- The XML parser always translates XML data to Unicode when parsing.

- You have to convert strings to byte arrays to set values in the COMMAREA, and byte arrays to strings to get values from the COMMAREA.

The Java parser parses, and it is up to the Java program to decide how to handle the parsed document, either by responding to SAX events like "element start tag" or by traversing the DOM tree.

If the program wants to use the contents of an element, then these contents must be converted into a form that can be used by whatever function receives the data. The method for doing this is dependent upon the Java function used. The conversion of the parsed document's content to a specific code page is a Java function, not an XML function.

Samples shipped with the Java parser for z/OS and OS/390, DOMWriter and SAXWriter, illustrate how to print an XML document once it has been parsed. For the Java parser, the samples files are:

```
/usr/lpp/ixm/IBM/XML4J-3_1_1/samples/dom/DOMWriter.java
/usr/lpp/ixm/IBM/XML4J-3_1_1/samples/sax/SAXWriter.java
```

When the Java program executes in a Java Virtual Machine (JVM) on z/OS or OS/390, conversion of a string to a byte array may require the encoding to be specified. Therefore, Java applications on z/OS or OS/390 should be code page-aware, specifying the encoding of the COMMAREA byte array in the Java code.

On z/OS or OS/390, avoiding conversion prior to calling the XML parser is the most efficient and least error-prone solution. If the XML document is converted before the parser is invoked:

- There will be a performance penalty because two conversions will actually occur: once from the original document code page, and once to Unicode within the parser.
- You must ensure that any conversion made prior to calling the parser does not create a discrepancy between the resulting data encoding and the “`encoding`” attribute implicitly or explicitly specified within the XML processing instruction.

The `String.getBytes()` method converts a string to a byte array, and the `String` constructor converts a byte array to a string; both use the default platform encoding unless otherwise specified. Example 7-1 shows an example of Java code for string and array manipulation. Using this technique, the COMMAREA flows to CICS in EBCDIC with the correct code page.

Example 7-1 EBCDIC COMMAREA with code page Cp500

```
// Setting the COMMAREA
byte inputCommarea []=new byte [];
xmldoc ="<?xmlversion=\"1.0 \"
encoding=\"ebcdic-cp-us \"?><TXReq><txname>WDDF</txname></TXReq>";
inputCommarea =xmldoc.getBytes("Cp500");     //any platform to Cp500

// Getting results from the COMMAREA
String outData =new String(outputCommarea.value,"Cp500");
```

For more information on encoding and XML on z/OS and OS/390, see the following Web site:

http://www.ibm.com/servers/eserver/zseries/software/xml/usage/#encode

8

A CICS SampleApp with XML

In this chapter, we introduce our sample application, SampleApp. In this application we use the XML Toolkit for z/OS and OS/390, and we demonstrate how to combine it with other e-business technologies such as servlets, Java Server Pages (JSPs), and WebSphere or VisualAge for Java.

The purpose is to provide an example of how to use the XML Toolkit for z/OS and OS/390 in an e-business application.

8.1 Application overview

The sample application, SampleApp, performs inventory management of a warehouse. The sample shows the implementation of two possible functions in an inventory management system: Price Change, and Price Quote. They are implemented as two different transactions in the CICS Transaction server.

The Price Change module involves input of ItemId , New Price, and Old Price, and the system returns the status of update to the user.

In the case of Price Quote, the user inputs the ItemId and Quantity. The system returns the Item Name, Item Price, and total price for the specified quantity.

Let's take a look of the Web-enabled application before integrating with XML.

The user enters the data required for running the transaction:

- An HTTP request having the user data and the transaction identifier is sent from the Web browser to the Web server.
- A servlet is invoked by the HTTP request through the form. The servlet runs within the JVM of the WebSphere Application Server servlet engine.
- Depending on the transaction identifier, the servlet transfers the data received from the request to the input commarea record of the transaction, and flows the ECI request to run the transaction through JavaGateWay using the CCF CICS connector classes.
- The data returned by the output commarea of the transaction is displayed in the browser using Java Server Pages.

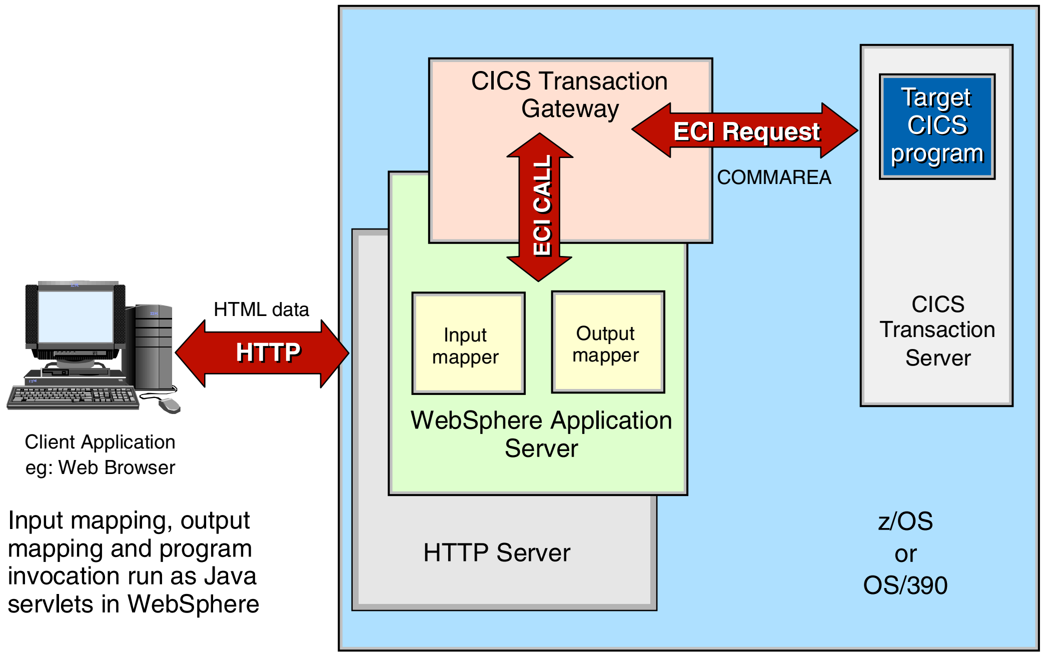


Figure 8-1 CICS SampleApp

8.1.1 Price Change module

UserInterface

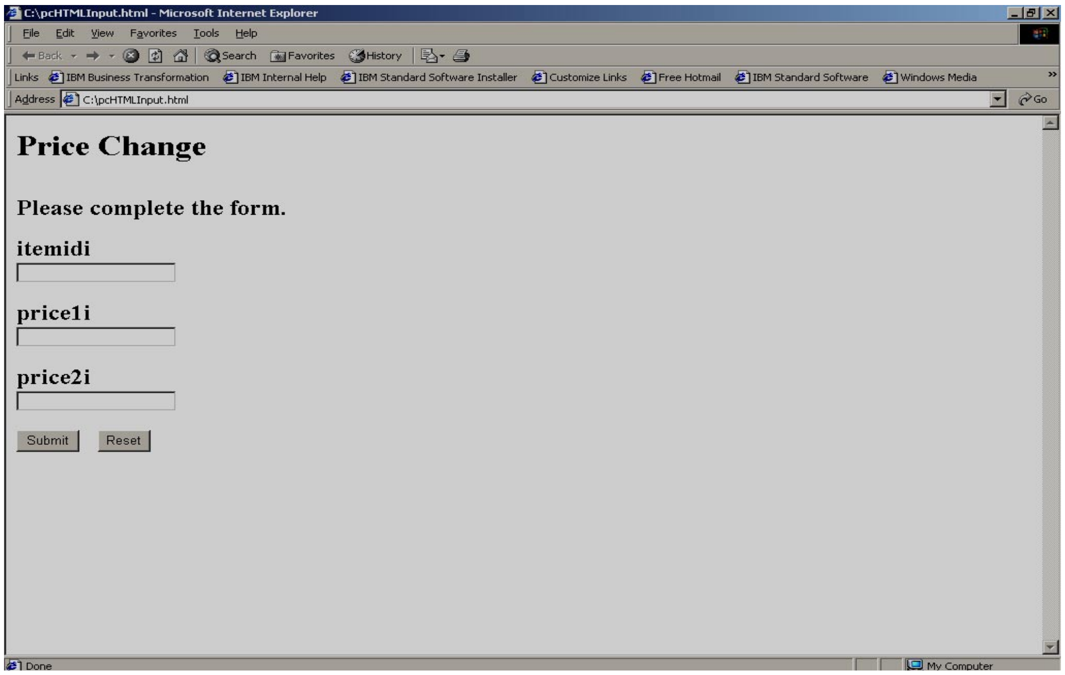

C:\pcHTMLInput.html - Microsoft Internet Explorer
Price Change
Please complete the form.
itemidi
price1i
price2i
Submit
Reset

Figure 8-2 Price Change transaction (PC)

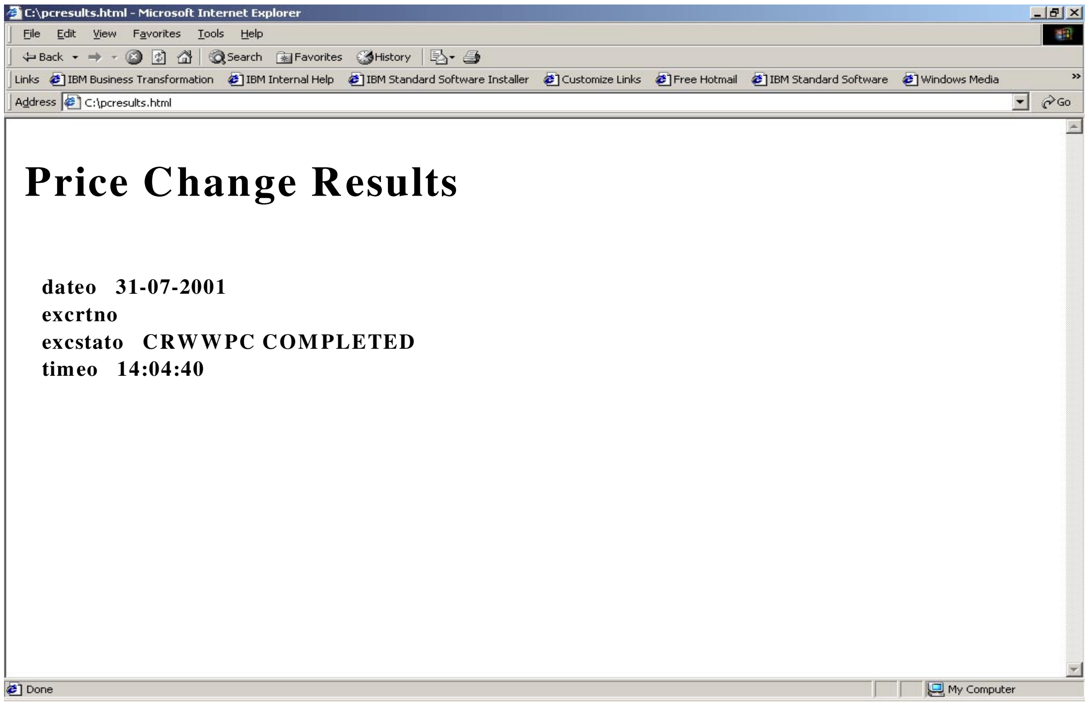
The user can enter ItemIdentifier(itemidi) , old price(price1i) and new price(price2i) for the item. On submitting this form, the item data along with the transaction identifier is passed as HTTPRequest to the Web server. A servlet is invoked by the HTTPRequest to handle the request.

Servlet logic

The servlet identifies the transaction requested and calls the specific driver to handle the request. The driver transfers the data from the request to the respective command object created from the transaction. In the case of a Price Change transaction, `PC_command` is instantiated and the properties `itemidi`, `price1i` and `price2i` are set. The transaction is triggered by calling the command as follows:

```
PC_command command = new PC_command();        // instantiate new command
command.setItemidi( req.getParameter("itemidi"));
command.execute();
```

On successful execution, the command object is put in the `HTTPSession` and a call is made to a Java Server Page to display the result. The JSP gets the command object from the `HTTPSession`, and hence the data in it, to display the status information of the transaction to the user.


C:\pcresults.html - Microsoft Internet Explorer
Address C:\pcresults.html
Done
My Computer

Price Change Results

```
dateo    31-07-2001
excrtno
excstato   CRWWPC COMPLETED
timeo   14:04:40
```

Figure 8-3 Price Change output page

8.1.2 Price Quote module

UserInterface

The user is prompted to enter the Item Identifier (`itemidi0, itemidi1...`) and the quantity (`qntyi0, qntyi1`..) required. On submitting this form, the item data along with the transaction identifier is passed as `HTTPRequest` to the Web server. A servlet is invoked by the `HTTPRequest` to handle the request.

Servlet logic

The servlet identifies the transaction requested and calls the specific transaction driver to handle the request. The driver transfers the data from the request to the respective command object created from the transaction. In the case of the Price Quote transaction, `PQ_command` is instantiated and the properties `itemidi0, itemidi1`...and `qntyi0, qntyi1`.. are set.

The transaction is triggered by calling the `execute()` method of the command:

```
PQ_command command = new PQ_command();       // instantiate new command
command.setItemidi0( req.getParameter("itemidi0"));
command.setQntyi0(req.getParameter("qntyi0");
command.execute();
```

Upon successful execution, the command object is put in the `HTTPSession` and a call to a Java Server Page is made to display the results. The JSP gets the command object from the `HTTPSession`, and hence the data in it, to display the ItemName, ItemPrice and Amount of the items specified.

8.2 SampleApp with XML

Now let’s see how XML can be integrated with the Web-enabled application. One way of doing this is to pass transaction data in an XML document through an HTTP request to the Application Server, where the data can be parsed to retrieve the required information and the target CICS application can be invoked passing the populated commarea through the CICS Transaction Gateway.

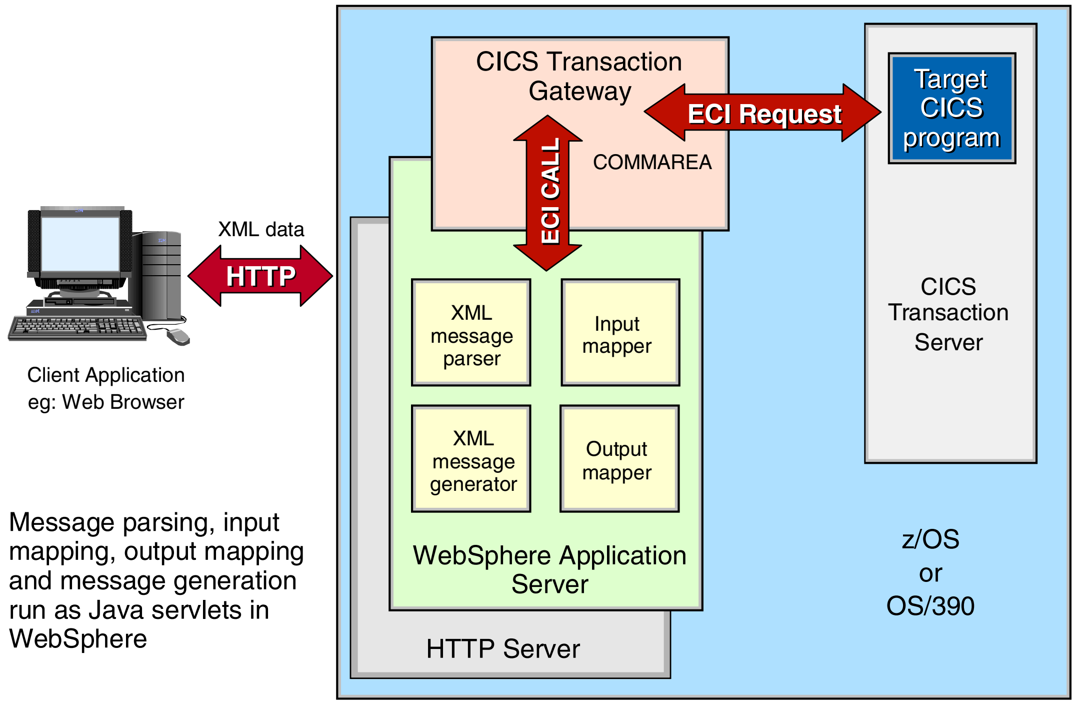


Figure 8-4 CICS SampleAPP with XML

8.2.1 Defining the XML Request format

The XML document passed to the Web server needs to have information for identifying the transaction to be invoked and the data required for running the transaction. A simple XML request format used in the sample application for the Price Change transaction is as follows:

```
<?xml version="1.0"?>
<TXReq>
<txname>WDPC</txname>
<itemidi>1</itemidi>
<price1i>1</price1i>
<price2i>2</price2i>
</TXReq>
```

The details of the request are encapsulated in a single element, `TXReq`. The transaction name is contained in the element txname. TXreq and txname are common for XML Requests passed for different transactions. Apart from these two elements, others are specific to the transaction invoked.

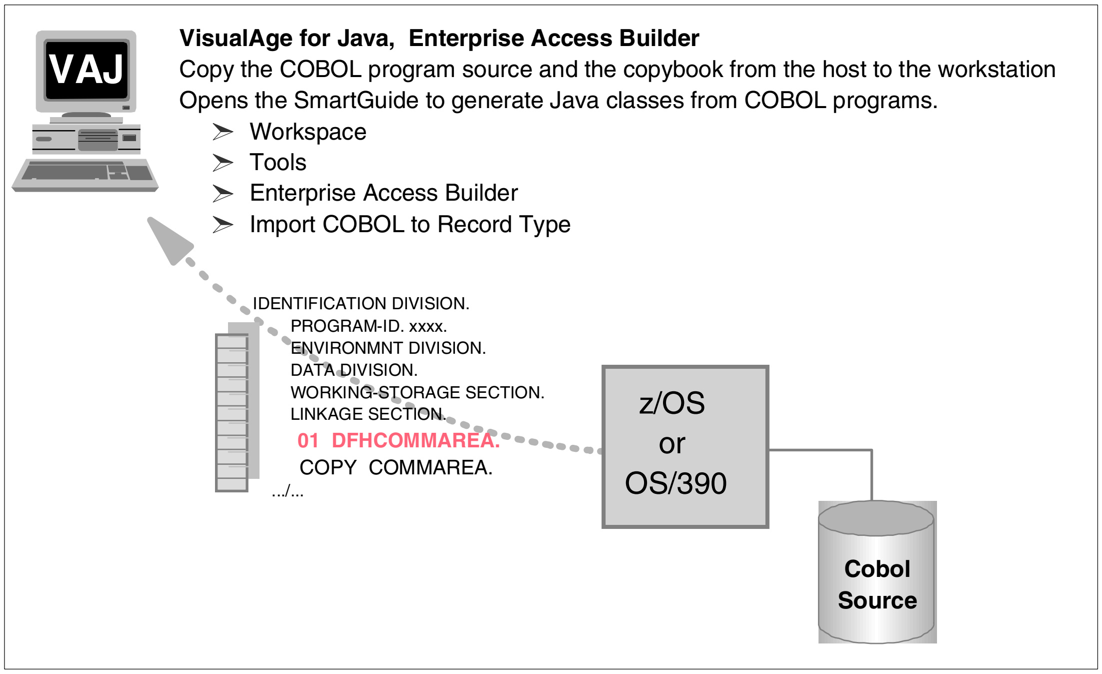
There is a one-to-one correspondence between an XML element and a property in the input commarea record of the target transaction. Only those properties which require data input for the tranaction need to have a corresponding tag in the XML document.

8.2.2 Creating the EAB records and commands

IBM VisualAge for Java provides the EAB and the Java Record Framework to analyze CICS COBOL COMMAREA structures and generate the CCF JavaBeans to represent input and output records.

In the Java Record Framework, a record wraps a data structure, such as a COBOL copybook. A record is a logical collection of application data elements that combines the actual record bytes together with its type. A record is used to implement the input and output of an EAB command.

The Enterprise Access Builder for Transactions (EAB) supplies the Record Type Editor tool that allows you to create a record type, or modify existing records. The Record COBOL Typeset extends the Record Framework to support COBOL structures; this is particularly useful for managing CICS COMMAREAs, which are usually handled as COBOL structures. You can use the EAB tools to import a COBOL structure to produce a matching record type.


VAJ
VisualAge for Java, Enterprise Access Builder
Copy the COBOL program source and the copybook from the host to the workstation
Opens the SmartGuide to generate Java classes from COBOL programs.
Workspace
Tools
Enterprise Access Builder
Import COBOL to Record Type
z/OS
or
OS/390
Cobol
Source

```
IDENTIFICATION DIVISION.
    PROGRAM-ID. xxxx.
    ENVIRONMNT DIVISION.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    LINKAGE SECTION.
     01  DFHCOMMAREA.
     COPY  COMMAREA.
  .../...
```

Figure 8-5 EAB Tools

Then, from this record type, a record can be produced. This record provides accessors for the fields of the COBOL structure and data conversion of the fields from Java types to COBOL types.

An EAB Command represents a transaction with its input and output COMMAREA. The Create Command and Command Editor in VisualAge helps you to create a command by specifying the input COMMAREA record and output COMMAREA record and the program name specified in the PPT entry. The properties of the input and output COMMAREA to be accessed from the client application can be promoted using the Promote Property option in the Command Editor.

In the sample application, PC_record represents the input and output COMMAREA of the Price Change **(PC)** transaction, and PC_command is the command object created for executing the transaction.

PQ_record represents the input and output COMMAREA of the Price Quote (PQ) transaction, and PQ_command the command object created for executing the transaction.

8.2.3 Servlet logic

The HTTPRequest invokes a servlet, CICSRouterServlet, which identifies the transaction by parsing the XML document passed, calls the specific transaction driver for executing the transaction, and returns the results to the user in an XML document in the form of a Java Server Page.

XML parsing considerations

It is important to be aware of some tradeoffs (not unique to XML) between encapsulation and performance or optimization. In general, when working with existing application code, one tends to wrapper it as above with a bean or other wrappering technology. This has the advantage of encapsulation and minimizing change with its attendant costs.

However, when writing new applications that will be using XML, it may be more appropriate to integrate the XML as the primary data transport into the application. This has performance and optimization advantages, but it tends to blur the differences between presentation and business logic.

Similarly, placement of function is a design consideration that comes into play. Depending on exactly what needs to be done, where the facilities exist, and what performance is required in production, you should choose where the XML processing should be done. In some cases, it may be expedient to build the XML-awareness on another server platform, while in other cases, it may be appropriate to integrate the XML-awareness with the back office system.

Additional performance considerations can be reviewed at:

http://www.ibm.com/servers/eserver/zseries/software/xml/perform/

Reusing the initialized parser

When parsing a small document, most of the CPU time is spent on parser initialization. So it is better to avoid instantiating a new parser every time you parse. Instead, you can create the parser once, and reuse the parser instance. A pool of reusable parser instances is a useful idea if you have many threads parsing at the same time.

Since our sample application is running in the Web server, it is ideal to maintain a pool of parsers so that it can serve multiple users simultaneously. The implementation of object pooling used in our application is shown in Example 8-1:

Example 8-1   object pooling

```
public class ObjectPool {
  protected Object objects[];
  protected Class  cls;
  protected int head;

public ObjectPool(String className, int size)
  throws ClassNotFoundException, IllegalArgumentException {
  this(Class.forName(className), size);
}

private ObjectPool(Class p_class, int size)
```

```
    throws IllegalArgumentException {
    if ((null == p_class) || (size <= 0)) {
      throw new IllegalArgumentException("Invalid arguments");
    }
    head = 0;
    cls = p_class;
    objects = new Object[size];
  }

  public Object getObjectFromPool()
    throws InstantiationException, IllegalAccessException {
    Object obj = null;
    synchronized(this) {
      if (head > 0) {
        head--;
        obj = objects[head];
        objects[head] = null;
    }
    else {
      obj = (Object)cls.newInstance();
    }
  }
    return obj;
  }

  public void returnObjectToPool(Object p_object) {
    synchronized(this) {
    if (objects.length > head) {
      objects[head] = p_object;
      head++;
    }
    else {
      expandObjectPool();
      objects[head] = p_object;
      head++;
      }
    }
  }

  public synchronized void expandObjectPool() {
    Object newObjectPool[] = new Object[objects.length * 2];
    System.arraycopy(objects, 0, newObjectPool, 0, objects.length);
    objects = newObjectPool;
    }
  }
```

The pool is initialized in the `init()` of the servlet. When the document needs to be parsed, an instance of the parser can be retrieved from the pool using the call `parserPool.getObjectFromPool()`.

Once the document is parsed, the parser can be reset and returned to the pool:

```
  parser.reset();
  parserPool.returnObjectToPool(parser);
```

Using SAX instead of DOM

Parsing with SAX is more efficient and requires less memory. In the case of large documents, this will lead to better performance of the application. The sample application uses `org.apache.xerces.parsers.SAXParser` available in `xerces.jar` supplied with the XML Toolkit for z/OS and OS/390.

BuildHashMap to avoid multiple parsing

In order to identify the transaction, the servlet needs to parse the document to get the information from the `<txname>` element. Once the transaction is identified, the specific transaction driver is called to process the data and call the transaction with populated commarea. In order to get the data to populate the commarea, the document needs to parsed again. To avoid multiple parsing of the document, we can store the data in a HashMap.

`BuildHashMap` extends `HandlerBase` and contains `HashMap` object as an instance variable. The servlet instantiates this object and associates it to the parser retrieved from the pool:

```
BuildHashMap counter = new BuildHashMap();
org.apache.xerces.parsers.SAXParser parser =
(org.apache.xerces.parsers.SAXParser)parserPool.getObjectFromPool();
parser.setDocumentHandler(counter);
parser.setErrorHandler(counter);
```

The parser can be invoked in different ways. The most two common ways are to specify the URL where the XML document is located, or to pass the XML document itself. Because in our sample application the XML document is passed through the `HTTPRequest`, we use the second approach:

```
String[] values = request.getParameterValues("xmldoc");
String xmldoc = values[0].trim();
StringReader sr = new StringReader(xmldoc);
InputSource is = new InputSource(sr);
parser.parse(is);
```

`BuildHashMap` has three instance variables:

- `currentElement` - The element which is being parsed
- `currentElementValue` - The value of the element
- `hashMap` - The HashMap object that holds pairs of elements and their values

The constructor instantiates an instance of the `HashMap` object and the `startDocument()` initializes the instance variables `currentElement` and `currentElementName` to null:

```
public BuildHashMap() {

   hashMap = new HashMap();

   }
public void startDocument() {
   currentElement      = null;
   currentElementValue = null;
   }
```

When an element is started, the `currentElement` is updated with the name. This can be done in the `startElement()` method:

```
public void startElement(String name, AttributeList attrs) {
        currentElement = name;
    }
```

The `currentElementValue` is updated every time a text value is found for the current element being parsed. This is done in the `characters()` method:

```
    public void characters(char ch[], int start, int length) {

        String s = new String(ch, start , length);
        currentElementValue = s;
        }
```

Once an element has been parsed and a text value is associated with it, it can be placed in the `hashMap` object. Then these variables need to be reset for the next element to be parsed. The right place to do this is in the `endElement()` method:

```
    public void endElement(String name) {

            if(currentElement!=null & currentElementValue != null){

                hashMap.put(currentElement,currentElementValue);
        }
            currentElement = null;
            currentElementValue = null;
        }
```

Once the `HashMap` is built from the XML document, the transaction to be invoked can be identified by checking the value of the element `txname`. The specific transaction driver for the target transaction is invoked passing the `hashMap` object as a parameter.

8.2.4 Transaction driver logic

Each transaction driver code is specific to a transaction. The purpose of this module is to instantiate the command object for the transaction, get the data for the transaction from the `HashMap`, populate the COMMAREA, and invoke the command to execute the transaction.

The servlet instantiates the driver object and invokes the `performTask()` method of the driver object, which takes a `HashMap` object as parameter.

In the case of a Price Change transaction, the servlet instantiates a `PC_driver` object and invokes the `performTask()` of the `PC_driver`.

In the constructor of the `PC_driver`, the `PC_command` is instantiated. In the `performTask()`, the values for the properties `itemidi`, `price1i` and `price2i` are retrieved trom the `hashMap` and set to the corresponding properties in the command object. The transaction is triggered by calling the `execute()` method of the command:

```
    command.setItemidi((String)hashMap.get("itemidi"));
    command.execute();
```

8.2.5 Sending the response to the user

On successful execution of the transaction driver, the servlet puts the command object in the `HTTPSession` and the results of the transaction are displayed to the user as an XML document:

```
if (txname.equals("PC")) {

       PC_driver pcDriver = new PC_driver();
      if (pcDriver.performTask(hash)) {
           request.getSession().setAttribute("pC_command", pcDriver.getCommand());
           nextPage = "pcXMLOutput.jsp";
         }
       }
// Call the output page.
   RequestDispatcher rd = getServletContext().getRequestDispatcher(nextPage);

   rd.forward(request, response);
```

In case of any exception, the servlet calls the report problem method, which displays a standard error page to the user and prints a stack trace of the exception in the server log:

```
private void reportProblem(HttpServletResponse response, HttpServletRequest request,
String message) {

   try{

    RequestDispatcher rd = getServletContext().getRequestDispatcher("error.jsp");
         rd.forward(request, response);

   }catch( IOException e){
      e.printStackTrace();
   }
   catch( ServletException e){
      e.printStackTrace();
}
```

8.2.6 Defining the XML Response format

The result of the transaction is returned to the client in the form of an XML document. This can be done in one of three ways (as detailed in Chapter 3, “IBM WebSphere Application Server and XML” on page 27):

- 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

Let’s choose the third option for the sample application. A Java Server Page can be designed to represent an XML document. Some simple directives define the content type as `text/xml`, and special tags can be used to insert dynamically generated values.

The following JSP 1.1 example generates an XML document for displaying the results of a Price Change transaction with dynamically generated values from the PC_command:

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

<%@ page contentType="text/xml; charset=utf-8" %>
<%@ page import="pc_tran.base.PC_command" %>
<jsp:useBean id="pC_command" class="pc_tran.base.PC_command" scope="session">
</jsp:useBean>
```

```
<TXRes>
<dateo><%= pC_command.getDateo() %></dateo>
<excrtno><%= pC_command.getExcrtno() %></excrtno>
<excstato><%= pC_command.getExcstato() %></excstato>
<timeo><%= pC_command.getTimeo() %></timeo>
</TXRes>
```

The content type is specified as `text/xml` and the character encoding is specified as `utf-8`.

The command object of the Price Change transaction, `PC_command`, is defined to the Java Server Page by the `page import` directive.

The instance of the command object stored in the `HTTPSession` can be accessed from the server page for processing by using the `jsp:useBean` syntax. The `id` represents the name given to the object, the `class` attribute represents the name of the class, while the `scope` represents whether the object is available in the session/request/page.

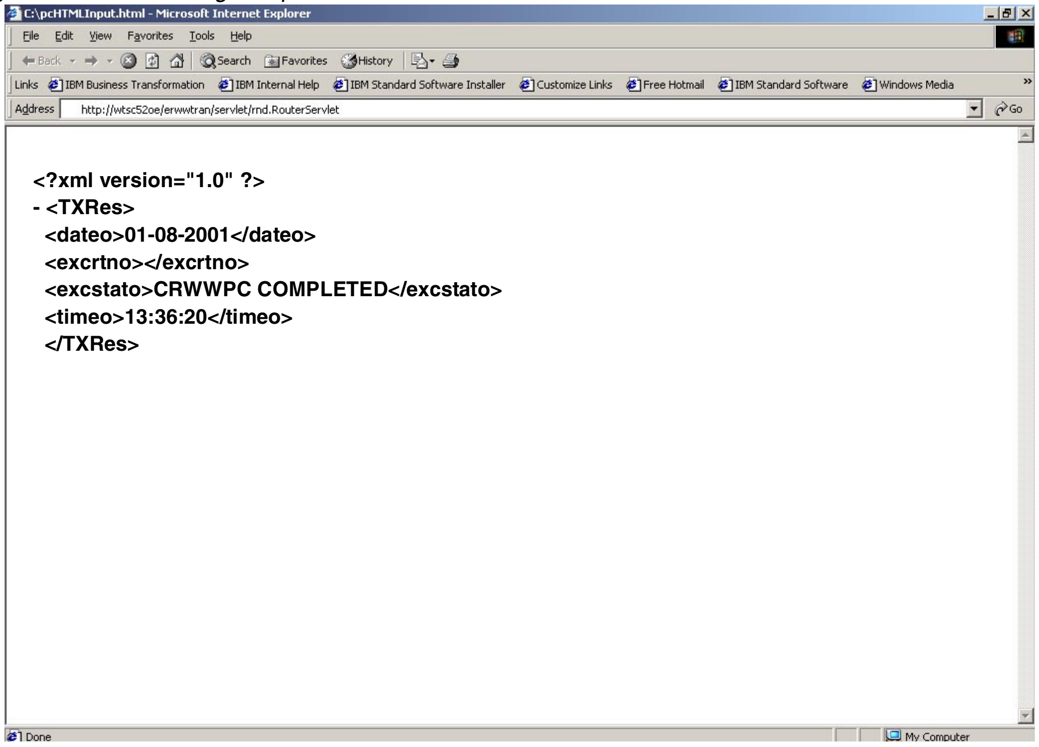
The results of the transaction are encapsulated in a single element, `TXRes`. The properties of the output COMMAREA record of the transaction are represented as child elements of this. Each property forms a tag with the value retrieved using the property's accessor in the command object.

For example, the status of successful completion of the Price Change transaction is represented by the property `excstato` of the COMMAREA. This is represented as an element in the XML document as:

```
<excstato><%= pC_command.getExcstato() %></excstato>
```

At runtime, the call to the command's accessor is replaced by the value of the property, thus displaying an XML document in the browser as in Figure 8-6.

Figure 8-6 Price change output


C:\pcHTMLInput.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Links IBM Business Transformation IBM Internal Help IBM Standard Software Installer Customize Links Free Hotmail IBM Standard Software Windows Media
Address http://wtsc52oe/erwwtran/servlet/rnd.RouterServlet
Go
Done
My Computer

```
<?xml version="1.0" ?>
- <TXRes>
  <dateo>01-08-2001</dateo>
  <excrtno></excrtno>
  <excstato>CRWWPC COMPLETED</excstato>
  <timeo>13:36:20</timeo>
  </TXRes>
```

9

XML Parser Java Edition in CICS

In Chapter 8, “A CICS SampleApp with XML” on page 77, we describe a SampleApp using the Java edition of the XML Toolkit for z/OS and OS/390 using WebSphere and CICS connectors.

This chapter illustrates how to use the Java and the XML parser directly from CICS. The purpose is to provide examples of how the Java edition of the XML Toolkit for z/OS and OS/390 can be used from within a CICS application.

9.1 JCICS application programming

JCICS application programming classes enable you to invoke CICS services from Java. All you have to do to invoke a CICS service is call a Java method. Methods provided by JCICS use Java exception classes to return error messages to the caller. The exception handling mechanism is internally based on the CICS API response codes, which are set after execution of the CICS commands.

The JCICS classes are built around the implementation of the C++ CICS Foundation classes. Most of the Java methods are based on native implementations in C. The DFJCZDTC dynamic link library contains the CICS command implementations, which are called from JAVA through the JNI.

The JCICS classes are packaged in the following .jar files:

`dfjcics.jar`	Contains all run time CICS classes
`dfjwrap.jar`	Contains the wrapper classes
`dfjcidl.jar`	Contains the IDL compiler
`dfjcorb.jar`	Contains the CORBA run time classes

These Java jar files have to be installed on the OS/390 server, and on the IDE workstation. CICS Java programs that use the JCICS classes are compiled just like other Java programs; no translation step is required. CICS Java programs invoke a CICS-supplied Java run time that provides the CICS functions. The JCICS classes represent CICS resources. You invoke methods on the classes to manipulate or set attributes of the resources

9.2 JCICS and VisualAge for Java

It is possible to develop CICS applications in Java using VisualAge for Java. Because you will compile your programs on the workstation in VisualAge for Java environment, you have to import the JCICS classes into VisualAge for Java. These classes are contained in the dfjcics.jar file.

Use the following procedure to import the classes:

1. Transfer the dfjcics.jar file to your workstation in binary format. This .jar file was installed into your MVS UNIX System Services environment when you installed CICS TS 1.3 or above. It can be found in the /usr/lpp/cicsts/<user_id>/classes directory.
2. Start VisualAge for Java and go to the workbench.
3. Select **Import** from the File menu.
4. Select the **Jar File** radio button in the SmartGuide Import dialog and click **Next**.
5. In the SmartGuide Import from a jar/zip file dialog, click the **Filename Browse** button and navigate to the directory containing the dfjcics.jar file.
6. Select the dfjcics.jar file and click **Open** to return to the SmartGuide Import from jar/zip file dialog.
7. Ensure that the check boxes for class and resource are checked.
8. In the Project field, type in: `JCICS Classes`.
9. Click **Finish**. A question box appears stating that the JCICS Classes project does not exist and asking if you want to create it. Click **Yes**.
10. Click **New Category** and type in: `JCICS` as the name for the new category.

11. Ensure that all beans under Available Beans are selected and that the new category, JCICS, is selected.
12. Click **Add to Category**.
13. Click **OK**.
14. Click **Finish**.

9.3 Setting up CICS, Java, and the XML Parser

The CICS TS 1.3 product base includes support for Java application programs. Variables that control the initialization of the JVM are put in a JVM environment PDS member.

To include the Java XML parser in our CICS region, we add the jar files provided with the Java parser to the CLASSPATH statement in the PDS member describing the JVM environment.

We followed these steps to tailor our CICS region to run with Java and the XML Java parser:

1. Updated the CICS region's startup JCL:
   a. Included SDFJLOAD library within the DFHRPL
   b. Included DD statement for JVM environment member
   c. Included DD dummy for JVM input
2. Updated the Java environment PDS member to include the Java XML parser within the CICS region's JVM environment CLASSPATH statement.

We defined a SDFHENV PDS to have characteristics using the CICS TS 1.3 distribution SDFHENV PDS. We customized "JVM environment" PDS member "CICS1" to include our Java application directory and the XML Java parser in the CLASSPATH statement.

See Figure 9-1 for a list of CICS region's CLASSPATH entries.

```
CLASSPATH=/u/cics1:/usr/lpp/cicsts/cicsts13/classes/dfjwrap.jar:/usr/lpp/cicsts/cicst
s13/classes/dfjcics.jar:/usr/lpp/cicsts/cicsts13/classes/dfjcorb.jar:/usr/lpp/java18/
J1.1/lib/classes.zip:/usr/lpp/ixm/IBM/XML4J-3_1_1/xerces.jar:/usr/lpp/ixm/IBM/XML4J-3
_1_1/xercesSamples.jar
Application java class file directory
/u/cics1
CICS entries required to be in the CLASSPATH
/usr/lpp/cicsts/cicsts13/classes/dfjwrap.jar
/usr/lpp/cicsts/cicsts13/classes/dfjcics.jar
/usr/lpp/cicsts/cicsts13/classes/dfjcorb.jar
Java entry required to be in the CLASSPATH
/usr/lpp/java18/J1.1/lib/classes.zip
XML Java parser entries to be included in the CLASSPATH
/usr/lpp/ixm/IBM/XML4J-3_1_1/xerces.jar
/usr/lpp/ixm/IBM/XML4J-3_1_1/xercesSamples.jar
```

Figure 9-1 CICS SDFHENV CLASSPATH

On our first attempt to run a Java program within our CICS region, the transaction abended. The error messages for our problem are shown in Figure 9-2 on page 92.

The missing member `libjava.a` shown there implies that the LIBPATH statement in our CICS region's SDFHENV member is missing an entry.

```
JESMSGLG output:
+DFHME0116
 (Module:DFHMEME) CICS symptom string for message DFHAP0001 is
 PIDS/565501800 LVLS/530 MS/DFHAP0001 RIDS/DFHSRP PTFS/ESA530 AB/U4038
 RIDS/DFHAPLJ1 ADRS/FFFFFFFF

CEEDMP output:
CEE3DMP V2 R10.0: Condition processing resulted in the unhandled condition.

Information for enclave main

  Information for thread 0D10800000000000

  Traceback:
    DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E  Offs
    19BDE430  CEEHDSP       197B5618  +000030BE  CEEHDSP       197B5618  +000030
    19BDD8D0  CEEHSGLT      197E5A10  +0000005C  CEEHSGLT      197E5A10  +000000
    19BDD558  CEEPTLOR      1985CE90  +000001FA  CEEPTLOR      1985CE90  +000001
    19BDD480                19A43CD0  -000001FE  @@TRGLOC      19A43A38  +000000
    19BDD360                0D20DF98  +00000230  Set_vm_args   0D20DF98  +000002
    19BDD1F8                0D20B730  +00000894  main          0D20B730  +000008
    19BDD0E0                19ACA156  +000000B4  EDCZMINV      19ACA156  +000000
    19BDD018  CEEBBEXT      0055E310  +000001A6  CEEBBEXT      0055E310  +000001

  Condition Information for Active Routines
    Condition Information for CEEHSGLT (DSA address 19BDD8D0)
      CIB Address: 19BDEAA8
      Current Condition:
     CEE3501S The module libjava.a was not found.
```

Figure 9-2 CEE3501S missing libjava.a error message

To correct our problem, we updated our CICS region's LIBPATH statement to include the directory /usr/lpp/java18/J1.1/lib/mvs/native_threads.

The SDFHENV member of our CICS region is listed in Figure 9-3 on page 93.

On our system, the Java home directory is /usr/lpp/java18/J1.1. It is important to note that, if the location of the Java HFS directories changes, CICS SDFHENV members need to be updated as well.

```
CHECKSOURCE=NO
CICS_HOME=.
CLASSPATH=/u/cics1:/usr/lpp/cicsts/cicsts13/classes/dfjwrap.jar:/usr/lpp/cicsts/cicsts13/classes/dfjc
ics.jar:/usr/lpp/cicsts/cicsts13/classes/dfjcorb.jar:/usr/lpp/java18/J1.1/lib/classes.zip:/usr/lpp/ix
m/IBM/XML4J-3_1_1/xerces.jar:/usr/lpp/ixm/IBM/XML4J-3_1_1/xercesSamples.jar
DISABLEASYNCGC=NO
ENABLECLASSGC=YES
ENABLEVERBOSEGC=NO
INVOKE_DFHJVMAT=NO
JAVASTACKSIZE=409600
JAVA_COMPILER=OFF
#JAVA_HOME=/usr/lpp/java18/J1.1
JAVA_HOME=/usr/lpp/java18/J1.1
/usr/lpp/java18/J1.1/lib/mvs/native_threads
MAXHEAPSIZE=8000000
MINHEAPSIZE=1000000
NATIVESTACKSIZE=262144
STDERR=dfhjvmerr
STDIN=dfhjvmin
STDOUT=dfhjvmout
VERBOSE=NO
VERIFYMODE=NO
```

Figure 9-3 CICS1 Java environment

9.4 SAXCount390 sample program

We used as a starting point the sample SAXCount390.java program provided in the samples subdirectory of the XML Parser for z/OS and OS/390, Java Edition Toolkit.

We modified the SAXCount390.java sample so that the parser input becomes the COMMAREA. First we had to define a Java class, SAXCount390, and code the required import statements including the following:

```
import com.ibm.cics.server.*;
class SAXCount390 extends HandlerBase()
```

The `import com.ibm.cics.server.*` statement points to the dfjcics.jar file, which is provided by CICS TS. The main() method of the program is shown in Figure 9-4:

```
public static void main(CommAreaHolder ca) {
              // is there anything to do?
String  parserName = DEFAULT_PARSER_NAME;
        boolean canonical  = false;
      SAXCount390 sax390 = new SAXCount390();
              // print uri
         sax390.print(parserName, ca);
    }         // main(String[])
```

Figure 9-4 Developing SAXCount390 Parser Program

The COMMAREA in the code uses a new class, `CommAreaHolder`, for COMMAREA handling. To overcome the limitation that Java passes parameters by value, a simple technique is introduced that uses a so-called Holder class, which consists of the code shown in Figure 9-5 on page 94.

```
public synchronized class CommAreaHolder
{
 public byte value[];
 public CommAreaHolder()
 {
    this(null);
 }
 public CommAreaHolder(byte ab[])
 {
value = ab;
 }
}
```

Figure 9-5 CommAreaHolder class

A byte array is defined as a public attribute value. You can set the value from outside, using the CommAreaHolder constructor with parameters. Because the attribute value is defined as public, you can use `instanceobj.value` to get the value to verify that the wrapper can be invoked from outside the CICS region using EXCI.

A main() of a sample client program to invoke the transaction from OMVS using `ECIRequest` is shown in Figure 9-6.

```
public static void main(String[] args) {

try{

String xmldoc = args[1];

byte[] abCommarea = xmldoc.getBytes();
JavaGateway jgaConnection = new JavaGateway();
jgaConnection.open();
ECIRequest eciRequest = new ECIRequest(
ECIRequest.ECI_SYNC,   // Call type
"  ",                  // CICS Server
"  ",                  // CICS userid
" ",                   // Password
"",                    // Program name
"",                    //Transaction name
abCommarea);           // Commarea byte array
jgaConnection.flow(eciRequest);
String strCommarea = new String(abCommarea);
System.out.println(strCommarea);

   }catch(Exception e){
      e.printStackTrace();
   }

   }
}
```

Figure 9-6 Sample client program with ECIRequest

9.4.1 Defining the SAXCount390 Java Program to CICS

Programs are defined in CICS using the CEDA transaction. CICS program names cannot be longer than 8 characters. Java byte code program names can be longer than 8 characters.

To accommodate long Java program names, the Resource Definition Online (RDO) program resource definition includes attributes for indicating if a particular RDO program is a Java program, as follows:

- JVM=YES for byte code java programs
- Hotpool=YES for HPJ compiled Java program objects
- JVM and Hotpool attributes set to NO for programs that are not Java

The RDO program resources that specify a JVM=YES attribute must also specify a JVMClass attribute. The JVMClass attribute is the case-sensitive Java program class name, without the .class suffix. The HFS directories specified in the SDFHENV member's CLASSPATH statement will be searched to load and run the Java program.

Our CICS region is configured to translate all terminal input to upper case. Entering the command **`CECI SET TERMINAL(XXXX) NOUCTRAN`** turns off upper case translation for our terminal ID: XXXX, without changing the region's upper case translation default.

Figure 9-7 shows the CLASSPATH, PATH and JAVA_HOME settings we used to compile the SAXCount390.java.

```
CLASSPATH=
/usr/lpp/ixm/IBM/XML4J-3_1_1/xerces.jar:/usr/lpp/ixm/IBM/XML4J-3_1_1/xercesSampl
es.jar:/usr/lpp/cicsts/cicsts13/classes/dfjcics.jar:/usr/lpp/java18/J1.1/lib/cla
sses.zip:.:
PATH=
/usr/lpp/java18/J1.1/bin:/bin
JAVA_HOME=
/usr/lpp/java18/J1.1
java version "1.1.8"
javac SAXCount390.java
Note: SAXCount390.java uses a deprecated API.  Recompile with "-deprecation" for
 details.
1 warning
```

Figure 9-7 CLASSPATH and PATH export, compile SAXCount390.java

Before running the Java SAXCount390 program, we define the program to CICS using the CEDA transaction:

1. To begin, we use the CEDA RDO PROGRAM definition panel to define programs to CICS. We enter the following command:

   ```
   CEDA DEFINE PROGRAM(SAX390) GROUP(JVATEST)
   ```

   If the JVATEST group does not exist, it will be created.

2. Within the RDO PROGRAM definition panel we define the attributes of our program entry. We press PF11 to display the section of the program definition panel that allows us to input JVM attributes.

   ```
   Set JVM = YES and JVMClass = SAXCount390
   ```

   Figure 9-8 on page 96 shows the Java attributes for our SAX390 RDO program definition. The JVMClass attribute is case sensitive. It must match the SAXCount390.class file in our /u/CICS1 directory.

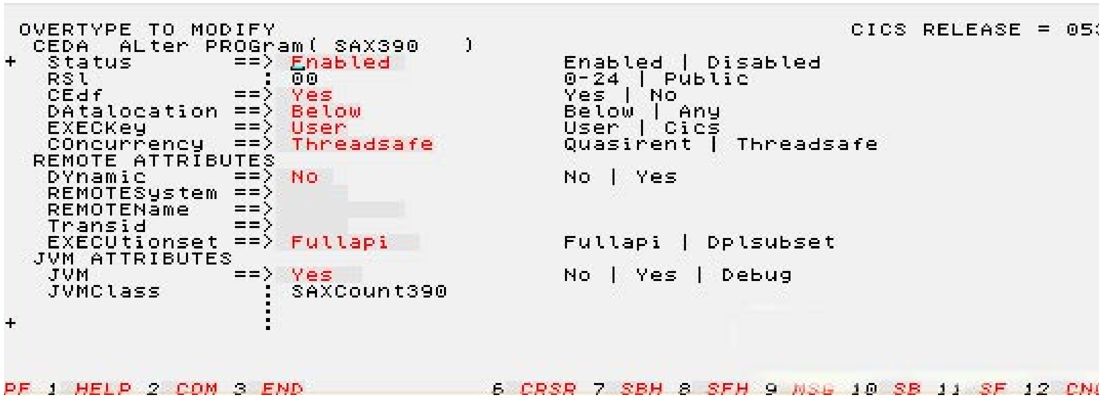
3. We press Enter.

   The RDO program SAXCount390 will be defined to CICS. A warning message will display indicating that the CONCURRENCY attribute was changed to THREADSAFE. This is expected. All Java programs must be defined as THREADSAFE.

   ```
   W JVM(YES) IMPLIES CONCURRENCY(THREADSAFE).
   ```

4. We exit from the Current CEDA session and install the JVATEST group:

   ```
   Press PF3
   CEDA INSTALL GROUP(JVATEST)
   ```



```
 OVERTYPE TO MODIFY                                                  CICS RELEASE = 05
  CEDA  ALter PROGram( SAX390    )
+   Status         ==> Enabled            Enabled | Disabled
    RSl            : 00                   0-24 | Public
    CEdf           ==> Yes                Yes | No
    DAtalocation   ==> Below              Below | Any
    EXECKey        ==> User               User | Cics
    COncurrency    ==> Threadsafe         Quasirent | Threadsafe
  REMOTE ATTRIBUTES
    DYnamic        ==> No                 No | Yes
    REMOTESystem   ==>
    REMOTEName     ==>
    Transid        ==>
    EXECUtionset   ==> Fullapi            Fullapi | Dplsubset
  JVM ATTRIBUTES
    JVM            ==> Yes                No | Yes | Debug
    JVMClass       : SAXCount390
                   :
+                  :

PF 1 HELP 2 COM 3 END          6 CRSR 7 SBH 8 SFH 9 MSG 10 SB 11 SF 12 CN
```

Figure 9-8 CEDA RDO program resource Java attributes

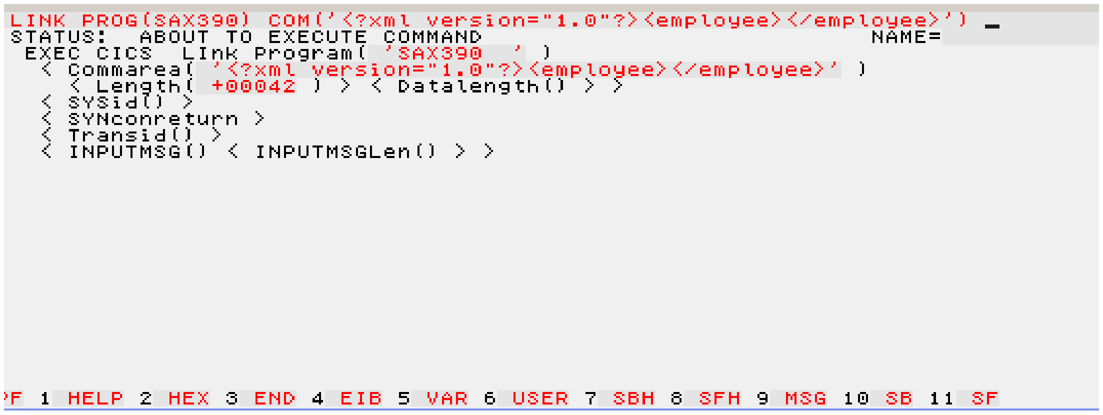
9.4.2 Linking to the SAXCount390 Parser program from CICS

The CECI command allows us to test calling our SAXCount java program, passing the program a COMMAREA that contains a well-formed XML document.

1. We enter the following command:

   ```
   CECI LINK PROG(SAX390) COM('<?xml version="1.0"?><employee></employee>')
   ```

   The CECI display screen, as shown in Figure 9-9, gives a status of: `ABOUT TO EXECUTE`. The program name we use for the link is the RDO program name that contains the JVMClass attribute `SAXCount390`. Note that the COMMAREA contains a well-formed XML document.



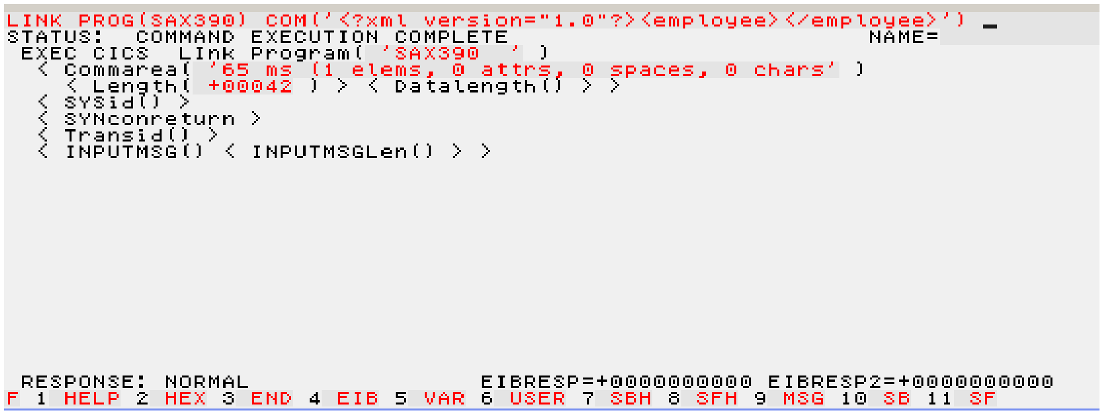
```
LINK PROG(SAX390) COM('<?xml version="1.0"?><employee></employee>')  _
STATUS:  ABOUT TO EXECUTE COMMAND                          NAME=
 EXEC CICS  LInk Program( 'SAX390   ' )
  < Commarea( '<?xml version="1.0"?><employee></employee>' )
    < Length( +00042 ) > < Datalength() > >
  < SYsid() >
  < SYNconreturn >
  < Transid() >
  < INPUTMSG() < INPUTMSGLen() > >

PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

Figure 9-9 About to Link to XML parser program

2. We press Enter.

We will now link to program SAX390. Because we specified a JVMClass attribute for the SAX390 program, a JVM will be established for the execution of ourJava program.

Figure 9-10 shows the results after returning from the program link. The COMMAREA returned contains the parsing results.



```
LINK PROG(SAX390) COM('<?xml version="1.0"?><employee></employee>')
STATUS:  COMMAND EXECUTION COMPLETE                          NAME=
 EXEC CICS  LInk Program( 'SAX390   ' )
  < Commarea( '65 ms (1 elems, 0 attrs, 0 spaces, 0 chars' )
    < Length( +00042 ) > < Datalength() > >
  < SYSid() >
  < SYNconreturn >
  < Transid() >
  < INPUTMSG() < INPUTMSGLen() > >









 RESPONSE: NORMAL                  EIBRESP=+0000000000 EIBRESP2=+0000000000
F 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF
```

Figure 9-10 Return from Parser program link

When you are used to working on a workstation with upper case translation on, it can be confusing to use a terminal with upper case translation off. Terminal input is now mixed case. Typing `ceci` is no longer the same as typing `CECI`.

After updating our mixed-case JVMClass attributes, we set our terminal back to upper case translation:

```
CECI SET TERMINAL(XXXX)
```

9.5 Sample link to a CICS program with COMMAREA

A LINK is a way for one CICS program to invoke another CICS program, with the invoked program returning control to the invoking program when it is done. With the LINK, CICS establishes a new environment for the LINKed program. The LINKed program gets data passed to it through a COMMAREA that was established by the invoking program.

The invoking program and the invoked program do not have to use the same programming language. For example, the invoking program can be a Java program that was started through an IIOP request or ECIRequest, and the invoked program can be a COBOL program containing the actual business logic.

Both the invoking and invoked programs must be defined to CICS, and they are not statically linked. The invoked program is still part of the same UOW. Therefore, any work done to recoverable resources by both programs is backed out if the task does not complete successfully.

Before a Java program can link to another program, at a minimum a program object must be created, and the program name to be called must be specified. Note that the COMMAREA that is passed is an array of bytes.

From a traditional CICS perspective, you would pass your data as an array of bytes. This method is required if the LINKed-to program is in a language other than Java.

However, if the LINKing and LINKed-to program are in Java, you can pass a serialized object. The `link()` method is part of the program object and is used to LINK to other CICS programs.

Example 9-1 shows the minimum code required to call the Price Change CICS transaction passing the COMMAREA.

Example 9-1 Call the CICS transaction passing the COMMAREA

```
try
 {
Program p = new Program();
  p.setName("WDPC");
  p.link(ca.value);
  }
catch (Throwable th) {
System.out.println("Problem calling backend program");

  }
```

10

IMS TM applications and XML

In this chapter, we discuss integrating IMS TM applications and XML.

10.1 IMS overview

IMS is the world's premier e-business database and transaction management system, offering unsurpassed performance in database availability and processing speed. Operating within the demands of an evolving e-business environment and a marketplace working in Web time, IMS delivers the integrity, capability, and performance expected by users.

IMS provides the means whereby you can most quickly and safely extend the reach of your existing applications and data via the Internet, while maintaining the enterprise computing qualities of service our customers and their users have come to expect.

IMS Version 7 builds upon this world-class product by providing enhancements to both its Database Manager (IMS DB) and Transaction Manager (IMS TM) systems.

10.2 Java and XML support in IMS

IMS Java application development support enables IMS applications to be written in Java and executed in IMS dependent regions. VisualAge workstation tools and host tools can be used for development and testing. IMS Java applications have access to IMS message queues and databases through the use of IMS Java classes. Additionally, the JDBC interface can be used to access both IMS databases and DB2 data. The IMS Java support enhances the ability of customers and business partners to provide integrated e-business application development with IMS.

The Extensible Markup Language (XML) is supported for IMS through interoperation with the OS/390 XML Parser, Java Edition. The XML parser Application Programming Interface can be used with the High Performance Java Compiler, shipped as part of VisualAge for Java, Enterprise Edition for OS/390, to develop new IMS Java programs running in IMS V7. Java applications programmers can thus invoke the APIs of the OS/390 XML Java Parser to convert an XML document from stream to parsed form for reading, editing, or updating an XML document.

Existing IMS transaction applications can be driven by sending and receiving XML documents. Parsing and data transformation occur outside of IMS applications. In addition to invoking the parser (which can validate the XML document), there is a need to transform the parsed XML stream to and from an IMS transaction message data structure.

Additionally, new or modified IMS transaction applications can directly receive, create, edit and transform XML documents by invoking the parser and transformers. The Java XML parser runs in the IMS Transaction Manager V7 environment. However, it should be compiled using the High Performance Compiler option in VisualAge for Java. Transformation to and from the IMS message format is the responsibility of the application.

10.3 IMS Connector for Java

IMS Connector for Java provides a way to create Java applications that can access IMS transactions using IMS Connect. In conjunction with the VisualAge for Java development environment, IMS Connector for Java lets you rapidly develop Java applications that run your transactions. With additional support from the IBM WebSphere Studio and IBM WebSphere Application Server, you can build and run Java servlets that access your transactions from Web sites.

IMS Connector for Java provides a Common Connector Framework-compliant Java interface to IMS Connect. IMS Connector for Java is a class library that consists of two packages: `com.ibm.connector.imstoc` and `com.ibm.imstoc`.

An EAB command for a typical IMS transaction includes Java beans that represent the IMS transaction input and output messages. These beans are constructed by VisualAge for Java's Enterprise Access Builder by first parsing the COBOL source of the IMS transaction's application program. The Enterprise Access Builder tool then uses this information to create the transaction input and output beans that match the input and output messages received and sent by that IMS application program

10.4 IMS Connect

The Open Transaction Manager Access (OTMA) Facility Callable Interface function improves IMS connectivity by providing a high-level interface for access to IMS applications from the OS/390 subsystems. It presents an API to application programs to enable access and execution of IMS transactions through IMS OTMA facilities. With this simple and easy-to-use interface, the invoker of the API can submit a transaction or command to IMS from within the z/OS or OS/390 environment without the necessity of understanding the technical protocols of the Cross Coupling Facility (XCF) or IMS OTMA.

A prerequisite to using IMS Connector for Java is IMS Connect. It is a separately provided and separately installed component that runs on the host IMS machine, through which a Java application or servlet accesses IMS OTMA.

IMS Connect is a program product that replaces IMS TCP/IP OTMA Connection, or simply ITOC. IMS Connect is SMP/E installable and maintainable, and provides several new functions and enhancements not available with ITOC. IMS Connect allows client applications to send messages to IMS TM through the IMS Open Transaction Manager Access (OTMA) interface, providing access to IMS transactions from a variety of platforms, including both workstation and mainframe products.

IMS supports Unicode (provided through the service process), letting you send Unicode application data to an IMS host application capable of handling Unicode, such as a Java application running in IMS Transaction Manager.

10.5 Accessing IMS using the IMS Connector for Java

IMS Connect, used with the VisualAge for Java development environment and IMS Connector for Java, can significantly ease development of e-business solutions that access IMS TM transactions. Java programs developed with these products can be deployed in WebSphere Application Server, allowing you to use Java servlets, Java server pages, and XML to quickly transform static Web sites into virtual sources of dynamic Web content.

Parsing and conversion of an XML message to IMS message can be done outside the IMS environment using the Java parser. A recommended way of doing this is to use a Java Server Page (JSP) or a servlet running under WebSphere Application Server, which passes the IMS message to a IMS transaction using the IMS Connector for Java and IMS Connect. Responses can be sent out via XML using the reverse path.

IMS Connect, which runs alongside IMS, allows TCP/IP clients to send messages to IMS. IMS Connect interfaces to IMS using the MVS Cross Coupling Facility (XCF) and the IMS Open Transaction Manager Access (OTMA) interface. Thus it provides connectivity to IMS Transaction Manager (TM) from any TCP/IP client application.

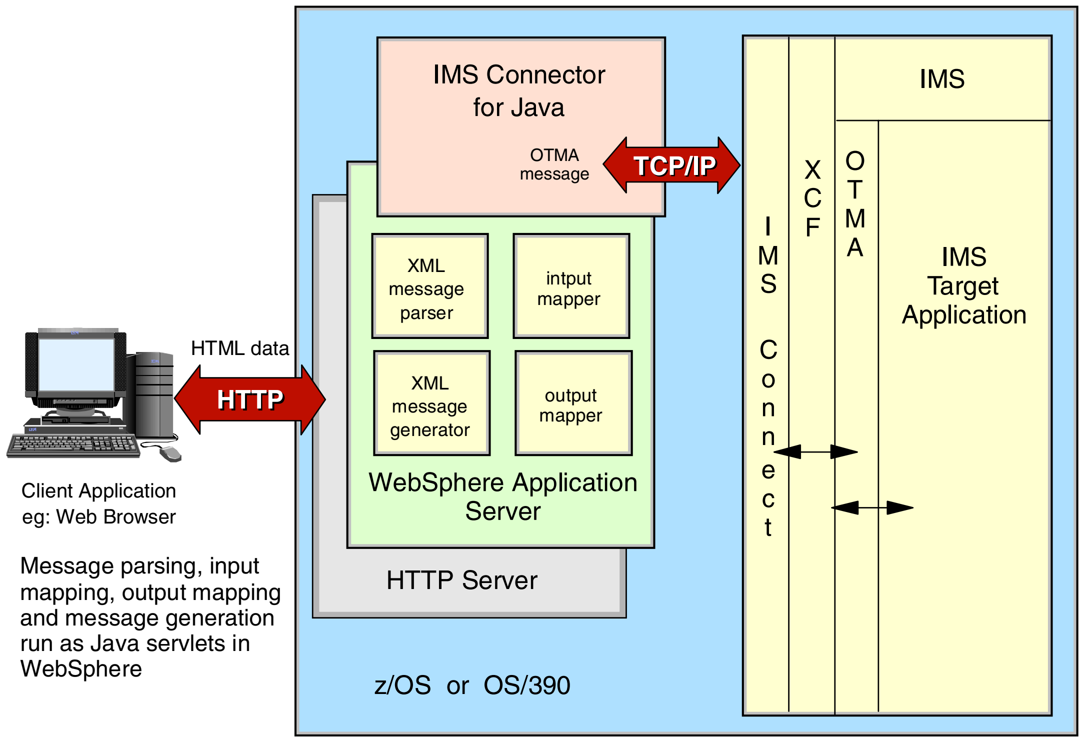


Figure 10-1 IMS Connector

Let's look at the scenario when parsing of an XML document happens in WebSphere Application Server, outside the IMS TM.

From the Web browser, a client enters the transaction input data on an HTML data.

1. An XML document containing the transaction input data is passed by the client (e.g. Web browser) through an HTTP request to the Web server.
2. The servlet invoked by the HTTP request runs within the JVM of the WebSphere Application servlet engine and acts as a TCP/IP client to IMS Connect.
3. The message parser parses the XML source to get the required data.
4. The input mapper populates the parsed data into the input commarea record of the transaction
5. The servlet executes the EAB command, which in turn invokes IMS Connector for Java. If the version of WebSphere Application Server that is running the Java servlet does not include the class libraries used by the servlet, these class libraries can be deployed to WebSphere Application Server in the form of JAR files.
6. IMS Connector for Java builds an OTMA message and sends it to IMS Connect, which in turn sends it to OTMA using XCF. OTMA sends the IMS transaction input message to IMS and receives the IMS transaction output message from IMS. The IMS transaction output message is sent back to IMS Connect by OTMA. IMS Connect then sends it to the servlet using IMS Connector for Java.
7. The output of the IMS transaction is displayed on the Web browser by the Java servlet using a JSP file.

11

An IMS SampleApp with XML

In this chapter, we introduce our sample application SampleApp. In this application we use XML Toolkit for z/OS and OS/390, and demonstrate how to combine it with other e-business technologies such as servlets, Java Server Pages (JSPs), and WebSphere or VisualAge for Java. The purpose is to provide an example of how to use XML Toolkit for z/OS and OS/390 in an e-business application.

11.1 Application overview

The sample application that was implemented for this redbook, SampleApp, performs inventory management of a warehouse. The sample shows the implementation of two of the possible functions in an inventory management system: Price Change, and Price Quote. They are implemented as two different transactions in the IMS Transaction server.

The Price Change module involves input of ItemId, New Price, and Old price, and the system returns the status of update to the user. In the case of Price Quote, the user enters the ItemId and Quantity, and the system returns the details of each item, such as Item Name, Item Price, and total amount for the specified quantity.

Let's take a look at the Web-enabled application before integration with XML; see Figure 11-1.

1. The user enters the data required for running the transaction.
2. An HTTP request having the user data and the transaction identifier is sent from the Web browser to the Web server.
3. A servlet is invoked by the HTTP request through the form. The servlet runs within the JVM of the WebSphere Application Server servlet engine.
4. Depending on the transaction identifier, the servlet transfers the data obtained from the request to the input message record of the transaction and invokes the **EAB** command from the TCP/IP OTMA connection to run the IMS transaction. The transaction request is routed to the IMS Transaction Manager using the IMS Connector for Java.
5. The output data returned by the transaction is displayed in the browser using Java Server Pages.

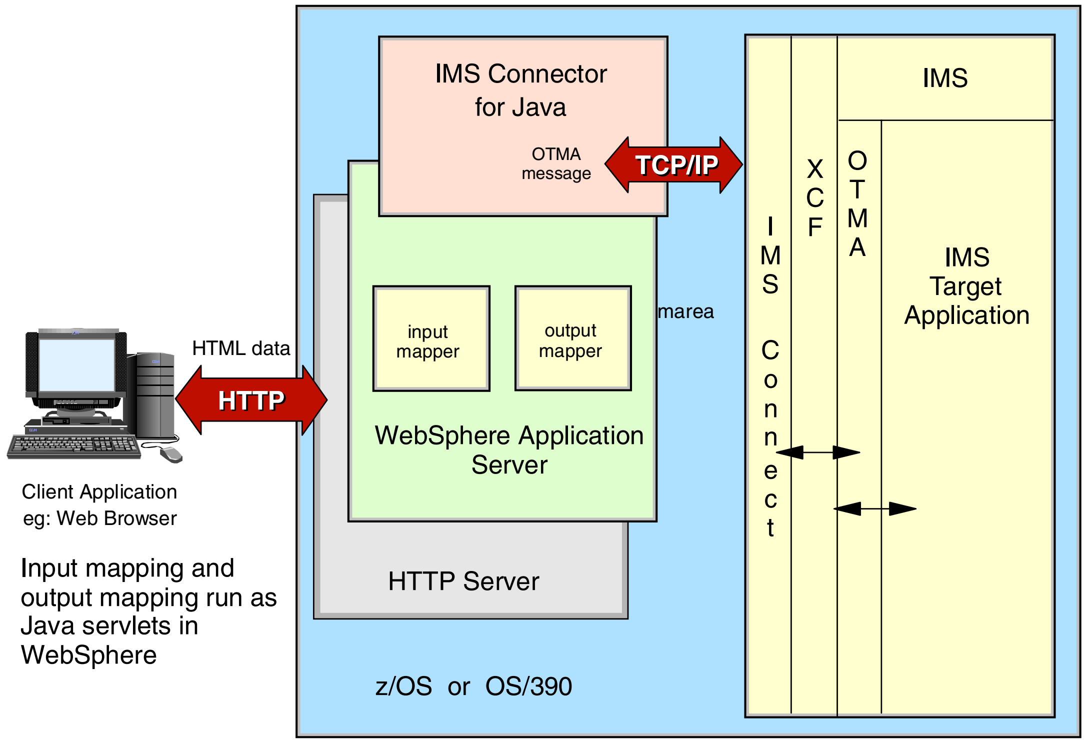
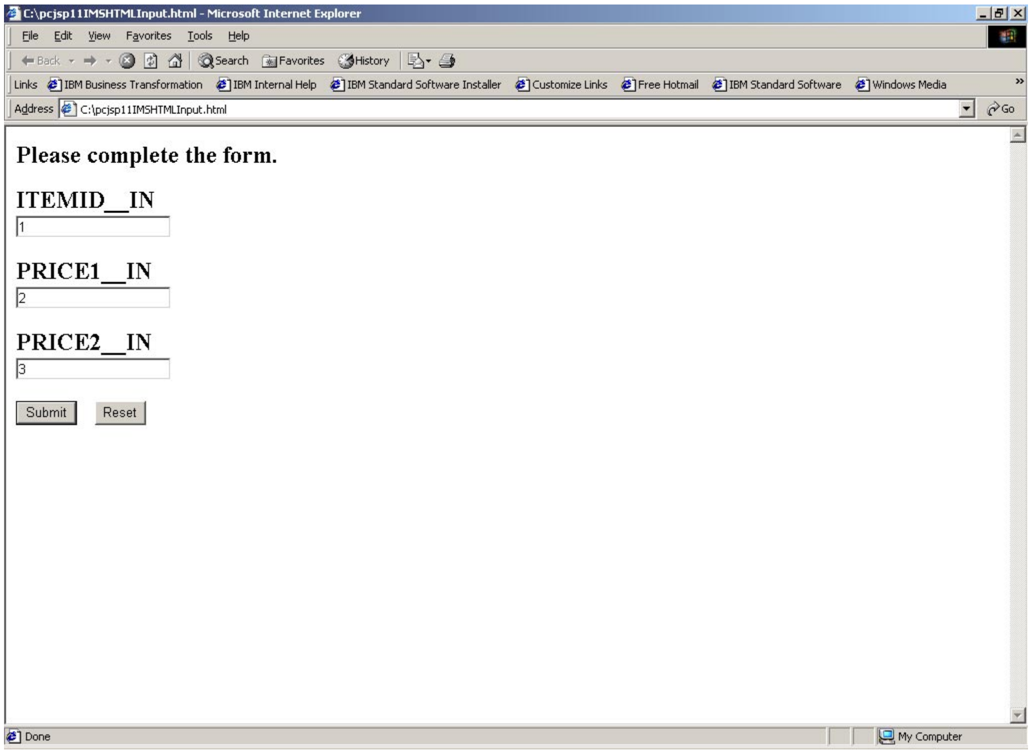


Figure 11-1 IMS SampleApp

11.1.1 Price Change module

UserInterface

The user can enter ItemIdentifier(ITEMID_IN), old price(PRICE1_IN) and new price(PRICE2_IN) for the item. On submitting this form, the item data along with the transaction identifier is passed as HTTPRequest to the Web server. A servlet is invoked by the HTTPRequest to handle the request; see Figure 11-2:


C:\pcjsp11IMSHTMLInput.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Links IBM Business Transformation IBM Internal Help IBM Standard Software Installer Customize Links Free Hotmail IBM Standard Software Windows Media
Address C:\pcjsp11IMSHTMLInput.html
Go
Submit
Reset
Done
My Computer

Please complete the form.

ITEMID__IN
1

PRICE1__IN
2

PRICE2__IN
3

Figure 11-2 Price Change form

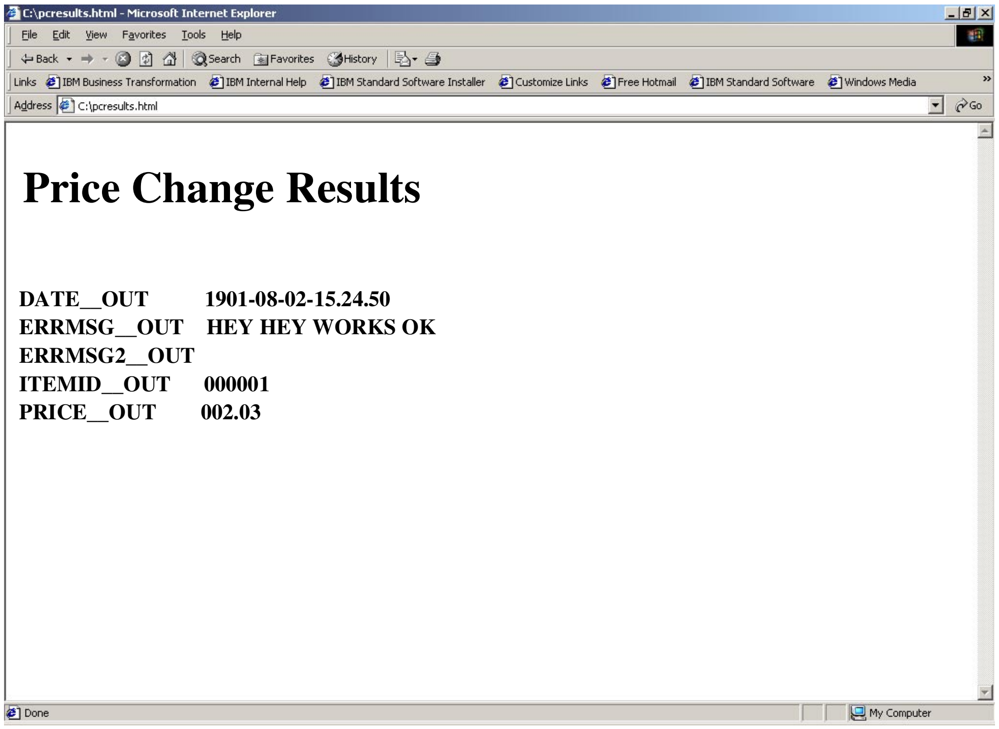
Servlet logic

The servlet identifies the transaction requested and calls the specific driver to handle the request. The driver transfers the data from the request to the respective command object created from the transaction. In the case of the Price Change transaction, pcIMSCommand is instantiated and the properties ITEMID_IN, PRICE1_IN and PRICE2_IN are set.

The transaction is triggered by calling the execute() method of the command. The length of the message and the transaction identifier are also set into the message, as shown:

```
ims.pcIMSCommand pcIMSCommand = new ims.pcIMSCommand();
    pcIMSCommand.setITEMID__IN(req.getParameter("ITEMID__IN));
   ............................
    pcIMSCommand.setLL__IN((short)23); //message length
    pcIMSCommand.setTRANID__IN("PCTIA"); //Transaction name
    pcIMSCommand.execute();
```

On successful execution, the command object is put in the HTTPSession and a call to a Java Server Page is made to display the result. The JSP gets the command object from the HTTPSession, and hence the data in it, to display the status information of the transaction to the user; see Figure 11-3 on page 106.


C:\pcresults.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Links IBM Business Transformation IBM Internal Help IBM Standard Software Installer Customize Links Free Hotmail IBM Standard Software Windows Media
Address C:\pcresults.html
Go
Done
My Computer

Price Change Results

```
DATE__OUT       1901-08-02-15.24.50
ERRMSG__OUT     HEY HEY WORKS OK
ERRMSG2__OUT
ITEMID__OUT     000001
PRICE__OUT      002.03
```

Figure 11-3 Price Change output display

11.1.2 Price Quote module

UserInterface

The user is prompted to enter the ItemIdentifier (`ITEMID_IN1, ITEMID_IN2...`) and the quantity (`QNTY_IN1,QNTY_IN2...`) required. On submitting this form, the item data along with the transaction identifier is passed as `HTTPRequest` to the Web server. A servlet is invoked by the `HTTPRequest` to handle the request

Servlet logic

The servlet identifies the transaction requested and calls the specific driver to handle the request. The driver transfers the data from the request to the respective command object created from the transaction. In the case of the Price Quote transaction, pqIMSCommand is instantiated and the properties `ITEMID_IN1, ITEMID_IN2` and `QNTY_IN1, QNTY_IN2` are set.

The transaction is triggered by calling the execute() method of the command. The message length and transaction name are set into the command, as shown:

```
ims.pqIMSCommand pqIMSCommand = new ims.pqIMSCommand();
    pqIMSCommand.setITEMID__IN1(req.getParameter("ITEMID__IN1"));
      .........................................
    pqIMSCommand.setQNTY__IN1(req.getParameter("QNTY__IN1"));
      ........................ ................
    pqIMSCommand.setLL__IN((short)132); //message length
    pqIMSCommand.setTRANID__IN("PQTIA"); //transaction name
    pqIMSCommand.execute();
```

On successful execution, the command object is put in the HTTPSession and a call to a Java Server Page is made to display the results. The JSP gets the command object from the HTTPSession, and hence the data in it, to display the ItemName, ItemPrice and Amount of the items specified.

11.2 SampleApp using XML

Now let's see how XML can be integrated with the Web-enabled application. One way of doing this is to pass transaction data in an XML document through an HTTP request to the Application Server. Here, the data can be parsed to retrieve the required information and the target IMS application can be invoked, passing the populated OTMA message to IMS using IMS Connector for Java API; refer to Figure 11-4:

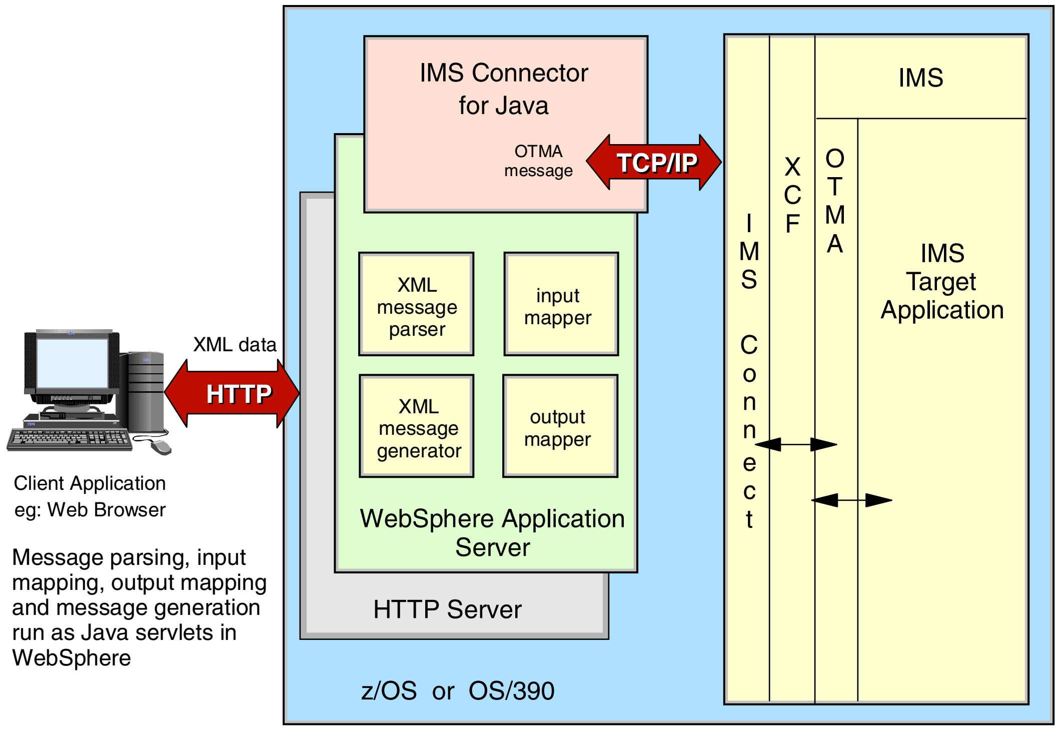


Figure 11-4 IMS SampleApp with XML

11.2.1 Defining the XML request format

The XML document passed to the Web server needs to have information for identifying the transaction to be invoked and the data required for running the transaction. A simple XML request format used in the sample application for the Price Change transaction is shown in Example 11-1.

Example 11-1 XML used for Price Change

```
<?xml version="1.0"?>
<TXReq>
<txname>PCTIA</txname>
<ITEMID__IN>1</ITEMID__IN>
<PRICE1__IN>2</PRICE1__IN>
<PRICE2__IN>3</PRICE2__IN>
</TXReq>
```

The details of the request are encapsulated in a single element, TXReq. The transaction name is contained in the element txname. Txreq and txname are common for XML requests passed for different transactions. Apart from these two elements, others are specific to the transaction invoked.

There is a one-to-one correspondence between an XML element and a property in the input message record of the target transaction. Only those properties which require data input for the transaction need to have a corresponding tag in the XML document. The tag names have been kept the same as the property name, in order to keep the format simple and straightforward.

11.2.2 Creating the EAB records and commands

IBM's VisualAge for Java provides the Enterprise Access Builder for Transactions (or EAB) and the Java Record Framework to analyze IMS transaction message structures and generate the CCF JavaBeans to represent input and output records.

In the Java Record Framework, a record wraps a data structure, such as a COBOL copybook. A record is a logical collection of application data elements that combines the actual record bytes together with its type. A record is used to implement the input and output of an EAB command.

However, if your IMS application program is not written in COBOL, you can represent the input and output messages as COBOL data structures and provide them to the Enterprise Access Builder tool.

An EAB command can be created to represent the IMS transaction with its input and output message. The Create Command of the Command Editor in VisualAge helps you to create a command by specifying the input message record and output message record. The properties of the input and output message to be accessed from the client application can be promoted using the Promote Property option in the Command Editor.

In the sample application, `pcIMSInputRecord` represents the input message and `pcIMSOutputRecord` the output message of the Price Change (PC) transaction, while `pcIMSCommand` is the command object created for executing the transaction.

Similarly, `pqIMSInputRecord` represents the input message and `pqIMSOutputRecord` the output message of the Price Quote (PQ) transaction, while `pqIMSCommand` is the command object created to execute the transaction.

11.2.3 Servlet logic

The `HTTPRequest` invokes a servlet, `IMSRouterServlet`, which identifies the transaction by parsing the XML document passed, calls the specific driver for executing the transaction, and returns the results as an XML document in the form of a Java Server Page.

Parsing of XML document

It is important to be aware of some tradeoffs (not unique to XML) between encapsulation and performance or optimization. In general, when working with existing application code, one tends to wrapper it as above with a bean or other wrappering technology. This has the advantage of encapsulation and minimizing change, with its attendant costs.

However, when writing new applications that will be using XML, it may be more appropriate to integrate the XML as the primary data transport into the application. This has some performance and optimization advantages, but tends to blur the differences between presentation and business logic.

Similarly, placement of function is a design consideration that comes into play. Depending on exactly what needs to be done, where the facilities exist, and what performance is required in production, you should choose where the XML processing should be done.

In some cases, it may be expedient to build the XML-awareness on another server platform, while in other cases, it may be desirable to integrate the XML-awareness with the back office system.

There are performance considerations that should be taken into account when using XML. Some of these can be found at the following site:

http://www.ibm.com/servers/eserver/zseries/software/xml/perform/

Reusing the initialized parser

When parsing a small document, most of the CPU time is spent on parser initialization. So it is better to avoid instantiating a new parser every time you parse. Instead, you can create the parser once, and reuse the parser instance. Having a pool of reusable parser instances is a useful idea if you have many threads parsing at the same time.

Since our sample application is running in the Web server, it is ideal to maintain a pool of parsers so that it can serve multiple users simultaneously. The implementation of object pooling used in our application is shown in Example 11-2.

Example 11-2 object pooling

```
public class ObjectPool {
  protected Object objects[];
  protected Class  cls;
  protected int head;

public ObjectPool(String className, int size)
  throws ClassNotFoundException, IllegalArgumentException {
  this(Class.forName(className), size);
}

private ObjectPool(Class p_class, int size)
  throws IllegalArgumentException {
  if ((null == p_class) || (size <= 0)) {
    throw new IllegalArgumentException("Invalid arguments");
  }
  head = 0;
  cls = p_class;
  objects = new Object[size];
}

public Object getObjectFromPool()
  throws InstantiationException, IllegalAccessException {
  Object obj = null;
  synchronized(this) {
    if (head > 0) {
      head--;
      obj = objects[head];
      objects[head] = null;
  }
  else {
    obj = (Object)cls.newInstance();
  }
}
  return obj;
}

public void returnObjectToPool(Object p_object) {
  synchronized(this) {
  if (objects.length > head) {
    objects[head] = p_object;
    head++;
  }
  else {
    expandObjectPool();
    objects[head] = p_object;
    head++;
    }
  }
}

public synchronized void expandObjectPool() {
  Object newObjectPool[] = new Object[objects.length * 2];
  System.arraycopy(objects, 0, newObjectPool, 0, objects.length);
  objects = newObjectPool;
  }
}
```

The pool is initialized in the `init()` of the servlet. When the document needs to be parsed, an instance of the parser can be retrieved from the pool using the call `parserPool.getObjectFromPool()`.

Once the document is parsed, the parser can be reset and returned to the pool:

```
parser.reset();
parserPool.returnObjectToPool(parser);
```

Using SAX instead of DOM

Parsing with SAX is more efficient and requires less memory. In the case of large documents, this will help to improve performance of the application. The sample application is using `org.apache.xerces.parsers.SAXParser` available in `xerces.jar` supplied with the XML Toolkit for z/OS and OS/390.

BuildHashMap to avoid multiple parsing

In order to identify the transaction, the servlet needs to parse the document to get the information from the `<txname>` element. Once the transaction is identified, the specific driver is called to process the data and call the transaction with populated OTMA message. In order to get the data to populate the input message, the document needs to be parsed again. To avoid multiple parsing of the document, we can store the data in a HashMap.

`BuildHashMap` extends `HandlerBase` and contains a HashMap object as an instance variable. The servlet instantiates this object and associates to the parser obtained from the pool; see Example 11-3:

Example 11-3 `BuildHashMap`

```
BuildHashMap counter = new BuildHashMap();
```

```
org.apache.xerces.parsers.SAXParser parser =
(org.apache.xerces.parsers.SAXParser) parserPool.getObjectFromPool();
parser.setDocumentHandler(counter);
parser.setErrorHandler(counter);
```

The parser can be invoked in different ways. The two most common ways are to specify the URL where the XML document is located, or to pass the XML document itself.

Because in our sample application the XML document is passed through the `HTTPRequest`, we use the second approach:

```
String[] values = request.getParameterValues("xmldoc");
String xmldoc = values[0].trim();
StringReader sr = new StringReader(xmldoc);
InputSource is = new InputSource(sr);
parser.parse(is);
```

BuildHashMap has three instance variables:

`currentElement`	The element that is being parsed
`currentElementValue`	The value of the element
`hashMap`	The HashMap object that holds pairs of elements and their values

The constructor instantiates an instance of the `HashMap` object and the `startDocument()` initializes the instance variables `currentElement` and `currentElementName` to null:

```
public BuildHashMap() {
   hashMap = new HashMap();
   }
public void startDocument() {
currentElement      = null;
      currentElementValue = null;
    }
```

When an element is started, the currentElement is updated with the name. This can be done in the `startElement()` method, as follows:

```
public void startElement(String name, AttributeList attrs)
   {
   currentElement = name;
   }
```

The currentElementValue is updated every time a text value is found for the current element being parsed. This is done in the `characters()` method; see Example 11-4:

Example 11-4 characters Method

```
public void characters(char ch[], int start, int length) {
   String s = new String(ch, start , length);
   currentElementValue = s;
   }
```

Once an element has been parsed and a text value is associated with it, it can be placed in the hashMap object. Then these variables need to be reset for the next element to be parsed. The right place to do this is in the `endElement()` method; see Example 11-5:

Example 11-5 EndElement Method

```
public void endElement(String name) {
   if(currentElement!=null & currentElementValue != null){
   hashMap.put(currentElement,currentElementValue);
   }
   currentElement = null;
   currentElementValue = null;
   }
```

Once the `HashMap` is built from the XML document, the transaction to be invoked can be identified by checking the value of the element `txname`. The specific driver for the target transaction is invoked passing the `HashMap` object as a parameter.

Transaction driver logic

Each transaction driver code is specific to a transaction. The purpose of this module is to instantiate the command object for the transaction, get the data for the transaction from the `HashMap`, populate the message, and invoke the command to execute the transaction.

The servlet instantiates the driver object and invokes the `performTask()` method of the driver object which takes a `HashMap` object as parameter.

In the case of the Price Change transaction, the servlet instantiates a `pcxmlIMS` object and invokes the `performTask()` of the `pcxmlIMS` object.

In the constructor of the PC_driver, `pcIMSCommand` is instantiated. In the `performTask()`, the values for the properties `ITEMID_IN`, `PRICE_IN1`, and `PRICE_IN2` are retrieved from the hash map and set to the corresponding properties in the command object. The transaction is triggered by calling the `execute()` method of the command, as shown:

```
command.setITEMID__IN((String)hashMap.get("ITEMID__IN"));
command.setLL__IN((short)23);
command.setTRANID__IN("PCTIA");
...........................
command.execute();
```

Sending the response to the user

On successful execution of the driver, the servlet puts the command object in the `HTTPSession` and the result page is displayed to the user as an XML document:

```
if (txname.equals("PQTIA")) {
        pqxmlIMS pqxmlims = new pqxmlIMS();
        if (pqxmlims.performTask(hash)) {
            request.getSession().setAttribute("pqIMSCommand", pqxmlims.getCommand());
            nextPage = "pqXMLIMSOutput.jsp";
          }
}
.....................

// Call the output page.
  RequestDispatcher rd = getServletContext().getRequestDispatcher(nextPage);

  rd.forward(request, response);
```

In case of any exception, the servlet calls the report problem method, which displays a standard error page to the user and prints a stack trace of the exception in the server log.

```
private void reportProblem(HttpServletResponse response, HttpServletRequest request,
String message) {

   try{

     RequestDispatcher rd = getServletContext().getRequestDispatcher("error.jsp");

        rd.forward(request, response);

   }catch( IOException e){
      e.printStackTrace();
   }
   catch( ServletException e){
      e.printStackTrace();
}
```

Defining the XML response format

The result of the transaction is returned to the client in the form of an XML document. This can be done in one of the following three ways, as detailed in Chapter 3, “IBM WebSphere Application Server and XML” on page 27:

- By using `println` statements and string manipulation to generate and output text strings from servlets
- By 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
- By using XML format JSPs to set elements of the XML document based on Java data

Let's choose the third option for the sample application. A Java Server Page can be designed to represent an XML document. Some simple directives define the content type as text/xml, and special tags can be used to insert dynamically generated values.

The JSP 1.1 example shown in Example 11-6 generates an XML document for displaying the results of a Price Change transaction with dynamically generated values from the `pcIMSCommand`:

Example 11-6 Java Server page for Price Change transaction

```
<?xml version="1.0"?>
<%@ page contentType="text/xml; charset=utf-8" %>
  <%@ page import="ims.pcIMSCommand" %>
  <jsp:useBean id="pcIMSCommand" class="ims.pcIMSCommand" scope="session"></jsp:useBean>
<TXRes>
<DATE__OUT>
  <%= pcIMSCommand.getDATE__OUT() %>
</DATE__OUT>
<ERRMSG__OUT>
  <%= pcIMSCommand.getERRMSG__OUT() %>
</ERRMSG__OUT>
<ERRMSG2__OUT>
  <%= pcIMSCommand.getERRMSG2__OUT() %>
</ERRMSG2__OUT>
<ITEMID__OUT>
  <%= pcIMSCommand.getITEMID__OUT() %>
</ITEMID__OUT>
<PRICE__OUT>
```

```
  <%= pcIMSCommand.getPRICE__OUT() %>
</PRICE__OUT>
</TXRes>
```

The content type is specified as `text/xml` and the character encoding is specified as `utf-8`. The command object of the Price Change transaction, `pcIMSCommand`, is defined to the Java Server Page by the page import directive.

The instance of the command object stored in the HTTPSession can be accessed from the server page for processing by using the jsp:useBean syntax. The id represents the name given to the object, the class attribute represents the name of the class, while the scope represents whether the object is available in session/request/page.

The results of the transaction are encapsulated in a single element, TXRes. The properties of the output message record of the transaction are represented as child elements of this. Each property forms a tag with the value retrieved using the property's accessor in the command object. For example, the status of successful completion of the Price Change transaction is represented by the property ERRMSG__OUT of the output message. This is represented as an element in the XML document as:

```
<ERRMSG__OUT>
  <%= pcIMSCommand.getERRMSG__OUT() %>
</ERRMSG__OUT>
```

At runtime, the call to the command accessor is replaced by the value of the property, thus displaying an XML document in the browser.

12

IMS Java IVP Sample using XML

An IMS Java program can process an XML document directly by invoking the XML Parser and interpreting the tagged data. In this chapter, we use the familiar IMS INSTALL/IVP Sample Application (IVP) which ships with IMS as an example to show how this is done.

12.1 Overview

The IVP is a very simple application that manages an IMS database representing a phone book. It has been implemented in every language that supports IMS, including Assembler, COBOL, PLI, REXX, and Java.

The IVP accepts the following commands for managing the phone book:

Add:	Adds an entry into the database
Delete:	Deletes an entry from the database
Display:	Displays a certain entry in the database
Update:	Updates an entry in the database
End:	Ends the IVP transaction

A phone book entry consists of the following parts or fields in the IMS database:

First Name:	Defined as 10 bytes of character data
Last Name:	Defined as 10 bytes of character data
Extension:	Defined as 10 bytes of character data
ZipCode:	Defined as 7 bytes of character data

Refer to *IMS V7 Installation Volume 1: Installation and Verification,* GC26-9429, for more information about the IVP.

The IVP is a Message Processing Program (MPP) which runs in a Message Processing Region (MPR), and as such its processing is based on the notion of reading input messages from the IMS TM message queue, processing them, and writing output messages to the message queue.

The MPR waits for messages for the IVP before loading and starting the IVP. The IVP continues to run until there are no more messages in the queue. Typically, an MPP is used to execute online transactions that must process and reply to messages quickly.

For the traditional IVP, an input message has the following format and sample contents:

.

LL	**ZZ**	IVTCC	ADD	Smith	John	5551234567	1234567

The first and second fields in the message, LL and ZZ, are IMS descriptor fields with length and other information for IMS's use. The third field, TRANCODE, is the IMS TM transaction code, which in the case of the IVP is IVTCC. The fourth field is the IVP command to be performed. In the sample, the command is ADD.

The remaining four fields constitute the phone book entry to be added by using the `ADD` command.

12.2 The XML

This section shows how IMS Java users can include XML-tagged data in the data portion of an IMS TM message. IMS Java applications can then be used to process this XML-tagged data.

The following shows an XML-tagged input message:

LL	**ZZ**	TRANCODE	<?xml version="1.0"?><input >...</input>

The first three fields are required for all IMS applications. IMS specifies that these fields must be in EBCDIC and may not be tagged.

The fourth field contains the XML-tagged data. The first tag, `<?xml version="1.0"?>`, is delimited by the first pair of angle brackets and identifies the XML version. Because this XML tag must be in lower case, the EDIT=(ULC) parameter must be coded in the IVP application's sysgen.

Upon receiving an XML-tagged TM message, the IMS Java transaction invokes the XML Parser to parse the XML message and extract the required input information, execute the required business logic, and generate the appropriate replies.

In the case of our IVP sample, two types of XML-tagged data are used in addition to the IVP root tag: commands, and phone book entries. Following are tags for these two types of data.

1. Commands describe which of the commands to execute:
   - ADD, DELETE, DISPLAY, UPDATE, or END.
2. Phone book entries describe fields in the database:
   - <First Name >
   - <Last Name >
   - <Extension>
   - <Zipcode>

Using XML-tagged data in the data portion of the input message, our sample input message now looks as shown in Example 12-1, using `<IVPInput>` as the root tag:

Example 12-1 Sample XML input message

```
<?xml version="1.0"? encoding="ebcdic-cp-us"?>
<IVPInput>
<ProcessCode>ADD</ProcessCode>
<FirstName>John</FirstName>
<LastName>Smith</LastName>
<Extension>5551234567</Extension>
<ZipCode>1234567</ZipCode>
</IVPInput>
```

Note: If your document is in UTF-8 or ASCII, you may leave out the encoding statement.

12.3 IMS Java Programming Model

IMS Java User Guide, SC27-0832, describes the basic programming model for IMS Java. Generally speaking, the steps to follow are these:

- Subclass the `IMSApplication` base class and implement the entire application in the `doBegin` method of this subclass.

  The `doBegin` method is an abstract method in the base class, `IMSApplication`. The main method of the subclass must call the begin method of the base class `IMSApplication`.

  The `begin` method of the base class then performs initialization and calls the abstract `doBegin` method that was implemented in the subclass.

- Subclass `IMSFieldMessage` which is used to receive and send messages. In our example, the `IMSFieldMessage` is subclassed to define our application's input message and output message formats and to make the fields in a message available to our Java program.

In the process, we identify the field name, data type, position, and length of each individual field within the byte array. This allows our application to use the accessor functions within the `IMSFieldMessage` base class to convert each data field automatically from its format in the message to a Java type that our application can process. In addition to the message-specific fields, `IMSFieldMessage` provides accessor functions that determine the transaction code and the length of the message.

- Perform a commit before processing the next message or terminating the application by calling `IMSTransaction.getTransaction().commit()`.

12.3.1 Putting it together

Our approach to processing an XML document in an IMS TM message is to read in the XML tagged input message, extract the IMS input using the XML parser, transform the XML input message to look like the usual input message expected by the transaction, and then process the reformatted message.

The output message will also be formatted for XML.

With this approach in mind, let's look at the various classes that make up the XML-enabled sample application.

IVP	Extends the IMSApplication base class and contains all of the business logic.
InputMessage	Used to describe input messages from the message queue.
OutputMessage	Used to describe output messages to the message queue.
XMLOutput	Used to describe XML tagged output messages to the message queue.
ParseXMLInput	Parses and extracts input data from the XML.
PhoneBookSegment	Maps the A1111111 segment of the DFSIVD2 database. This class is independent of any XML considerations.
IVPDatabaseView	Contains the segment information for the database DFSIVD2. This class is independent of any XML considerations.
SPAMessage	Used to store data between conversations. This class is independent of any XML considerations.

Let's begin by looking at `InputMessage`, as shown in Example 12-1:

```
public class InputMessage extends com.ibm.ims.application.IMSFieldMessage
{
    static DLITypeInfo[] fieldInfo = {
        new DLITypeInfo("Reserved",DLITypeInfo.CHAR,1,4),           // line 1
        new DLITypeInfo("ProcessCode",DLITypeInfo.CHAR,5,8),
        new DLITypeInfo("LastName",DLITypeInfo.CHAR,13,10),
        new DLITypeInfo("FirstName",DLITypeInfo.CHAR,23,10),
        new DLITypeInfo("Extension",DLITypeInfo.CHAR,33,10),
        new DLITypeInfo("ZipCode",DLITypeInfo.CHAR,43,7),
        new DLITypeInfo("XMLRequest", DLITypeInfo.CHAR,   5, 1800)   // line 2
    }
};
```

Figure 12-1 InputMessage

`IMSFieldMessage` and `DLITypeInfo` are IMS Java classes which describe the message segment layout and its data characteristics.

`IMSFieldMessage` has accessor methods for the named fields of type `DLITypeInfo`. The code in Example 12-1 on page 117 shows that our program can support either a traditionally-defined IVP input message or an XML document.

The traditional non-XML IVP message is defined in the example beginning with the line denoted by `line 1`. It indicates that the `ProcessCode` is found starting at byte 5 with a length of 8 bytes, and is followed by additional information as defined in the subsequent five lines.

The XML document, on the other hand, is defined in the line denoted by `line 2`, starts on byte 5, and has a length of 1800 bytes.

The net result is that we now a “bilingual” transaction message, in the sense that it can accept both XML-tagged data and non-XML-tagged data.

In the `doBegin` method of the IVP class, we code a test for XML input and do the XML parse.

```
MsgQ.getNextMessage(inputMessage)
    ....
    procCode = inputMessage.getString("ProcessCode").trim();
    if (procCode.indexOf("<?xml") != -1) {
          if (xmlParser==null) {
                xmlParser = new ParseXMLInput();
          }
          isXMLOutput = true;
          try {
                xmlParser.parse(inputMessage);
          } catch (Exception e) {}
          inputMessage = xmlParser.getConvertedInput();
     }
```

The real workhorse for this XML example is the class `ParseXMLInput`. This class allows our XML-enabled IVP application to detect when the input message contains an XML document. When an XML document is found, this class remaps the XML document to the traditional IVP message format. This remapped message can then be processed as before.

`ParseXMLInput` extends `HandlerBase` in order to do a SAX parse. We implement the following methods to track the start and end tags and extract the XML data:

```
public void characters(char[] arg1, int start, int length)
public void startDocument()
public void startElement(String element, AttributeList attrlist)
public void endElement(String element)
public void endDocument()
```

Note the following definitions:

`characters`	is a method to copy the data into the InputMessage class based on the current tag
`startDocument`	is used to create a new InputMessage
`startElement`	is used to identify which current tag is being processed
`endElement`	is used to identify which tag is no longer to be processed
`endDocument`	is used to signal completion

Two additional methods of `ParseXMLInput` get a new instance of the parser, do the SAX parse, and return the converted input message.

```
public InputMessage getConvertedInput ( )
public void parse(InputMessage xmlInput)
```

Note the following definitions:

`getConvertedInput`	is used to return a traditionally-formatted IVP input message built from the XML document
`parse`	is used to parse the XML document. The actual code is shown in Example 12-2:

```
public void parse(InputMessage xmlInput) throws Exception {

    try {
        String xmlDocument = xmlInput.getString("XMLRequest").trim();
        SAXParser parser = (SAXParser)
        Class.forName(DEFAULT_PARSER_NAME).newInstance();
        parser.setDocumentHandler(this);
        StringReader xmlReader = new StringReader(xmlDocument);
        parser.parse(new InputSource(xmlReader));

        } finally {   }
}
```

Figure 12-2 Parse

Note: You will receive a warning message about deprecated methods after you compile this class. This is because we are using SAX 1.0 APIs. The replacements for `HandlerBase` and `AttributeList` are SAX 2.0 APIs that are part of the XML Toolkit for z/OS and OS/390 but are not supported.

12.3.2 Building your application

To build your application, the first step is to set up your classpath for IMS Java. You will need to issue the following in OMVS in the directory that contains the imssam directory.

```
export CLASSPATH=$CLASSPATH:.:/usr/lpp/ims/imsjava71/classes.zip:/usr/lpp/hpj/lib
```

Note: This assumes that your classpath already includes the `xerces.jar` and `xercesSample.jar` files that were discussed in earlier chapters.

To compile your Java source code, use the **`javac`** command:

```
javac imssam/*.java
```

Next, to compile the class files for execution with IMS, you will need to invoke the hpj compiler as follows:

```
hpj imssam.IVP -main=imssam.IVP \
 -o="//'itso3.imsjava.lib(DFSIVP32)'" -alias=DFSIVP32 \
 -target=IMS \

-lerunopts="ENVAR(CLASSPATH=/usr/lpp/ims/imsjava71:/usr/lpp/ims/imsjav71/imsjava.jll:/us
r/lpp/hpj/lib"  \
 -include=org.apache.xerces.msg.XMLMessages \
 -include=imssam.IVPDatabaseView \
 -exclude=com.ibm.ims.db.* \
 -exclude=com.ibm.ims.db.* \
 -exclude=com.ibm.ims.application.* \
 -exclude=com.ibm.ims.base.*
```

This command compiles all classes into .o files, creates an executable, and places that executable into the PDSE itso3.imsjava.lib(DFSIVP32) for use in an MPP region.

While the **hpj** command can determine most of the files it will need to compile, it may miss some. In the case of the IVP sample, the class `IVPDatabaseView` and the `xerces` class `XMLMessages` need to be explicitly included.

You won't need any of the .o files that are created as a result of the hpj process, so you may delete them afterwards.

To run the IVP sample you will need to add the PDSE to the STEPLIB of the JCL used to bring up the MP region where the IVP will run. You will also need to include datasets for the IMS Java class libraries and the HPJ class libraries; see Figure 12-3:

```
//STEPLIB  DD DSN=ITSO3.IMSJAVA.LIB,DISP=SHR
//         DD DSN=hlq.SDFSJLIB,DISP=SHR
//         DD DSN=IMS710E.&SYS2.PGMLIB,DISP=SHR
//         DD DSN=IMS710E.&SYS2.SDFSRESL,DISP=SHR
//         DD DSN=HPJ.SHPJMOD,DISP=SHR
//         DD DSN=HPJ.SHPOMOD,DISP=SHR
```

Figure 12-3 STEPLIB JCL statements

Note: The PDSE with the new IVP must be listed *first* so as not to pick up the default IVP executable supplied by IBM.

To ensure that IVP can accept lower case, you will need to do a sysgen using the code shown in Example 12-2:

Example 12-2 SYSGEN to accept lower case

```
APPLCTN PSB=DFSIVP32,PGMTYPE=TP,SCHDTYP=PARALLEL           HDAM/VSAM-C
  TRANSACT CODE=IVTCC,SPA=(80,),MODE=SNGL,EDIT=(ULC),                X
             MSGTYPE=(SNGLSEG,NONRESPONSE,1)
```

In addition, you will also need to do a PSBGEN to add TELEPCB as a PCBNAME. You can use the sample shown in Example 12-3:

Example 12-3 PSBGEN

```
TELEPCB  PCB    TYPE=DB,DBDNAME=IVPDB2,PROCOPT=A,KEYLEN=10
       SENSEG NAME=A1111111,PARENT=0,PROCOPT=A
       PSBGEN LANG=ASSEM,PSBNAME=DFSIVP32
       END
```

A

GNU gmake

In this appendix we describe GNU and the gmake utility (gmake). We also tell you how to obtain gmake from the Web and how to install it.

What is GNU

GNU is a recursive acronym that means “GNU’s Not Unix.” The initial announcement of GNU was written by Richard Stallman in 1983. In that announcement, Richard declared his intention to develop a UNIX-comparable software system and give it away free.

For a detailed history of GNU, visit the GNU Web site:

```
http://www.gnu.org/
```

The gmake configuration utility

The gmake utility is an open software version of the make utility. Like make, gmake uses the commands specified in its input file (default input file: makefile) to help you build and manage a product. Although the commands of gmake and make are similar, there are differences. Therefore, the choice of using gmake or make is based on the utility for which the makefile was written.

Detailed information about the gmake utility can be found at the following Web site:

```
http://plan9.btv.ibm.com/gnu/make.html
```

The gmake utility is one of many open source software utilities that have been ported to OS/390. Refer to *Open Source Software for OS/390 UNIX,* SG24-5944 for additional information about gmake and other open source software utilities for OS/390.

Open Source Software for OS/390 UNIX, SG24-5944 is available online at:

```
http://www.ibm.com/redbooks
```

Obtaining gmake

The gmake utility is included in the Tools & Toys section of the z/OS UNIX System Services home page:

http://www.ibm.com/servers/eserver/zseries/zos/unix/

From that page, select the link **Tools and Toys**. There are several ways of downloading the gmake utility. Click the link **downloading packages** for a description of the downloading methods.

To download via the browser:

1. Click **ported tools**.
2. Scroll down the page and right-click the link **gmake**.

   To download the gmake file, select **save as..**

Installing gmake

Open Source Software for OS/390 UNIX, SG24-5944, recommends that open source software be installed in the /usr/local directory—and notes that if it is created as a separate HFS, then this file system will not be affected when OS/390 is upgraded.

To perform this installation, do the following:

1. Upload the gmake.pax file to a UNIX system services directory.
2. From the TSO command environment, invoke the OS390 shell and enter the command:

   ```
   OMVS
   ```

3. Switch to the directory that contains the gmake.pax file upload. Extract the files in the gmake.pax file and then enter the following command:

   ```
   pax -rf gmake.pax
   ```

   The extract will be put in the gmake subdirectory of the current directory.
4. Change to the gmake subdirectory:

   ```
   cd gmake
   ```

The README file indicates that the gmake.install file can be used to install the gmake utility. The gmake.install file is listed in Figure 12-4 on page 125, and it includes line numbers that we reference here.

Lines 25 and 26 contain copy file commands.(Note that the gmake.install file does not check for the existence of the destination directories, nor does it attempt to create the directories.)

Before using gmake.install to install gmake, verify that the destination directories are already defined. If they are not already defined, do so now.

- Define directories that gmake.install expects to exist:

  ```
  mkdir /usr/local/bin
  mkdir /usr/local/man
  mkdir/usr/local/man/$LANG
  mkdir /usr/local/man/$LANG/cat1
  ```

- Run the install procedure:

  ```
  gmake.install make-3.76.1.os390.tar.Z /usr/local
  ```

Messages indicating that some GNU tools are missing (aclocal, autoconf, automake, autoheader, makeinfo) can be ignored. They are not needed for running gmake.

The gmake.install script copies the GNU gmake program to the /usr/local/bin directory as program name make. The gmake.install creates gmake as a symbolic link to the make file.

The symbolic link will be in the /usr/local/bin directory. Therefore, if you add /usr/local/bin ahead of /bin in your PATH, either gmake or make will invoke GNU gmake. If you need to execute the standard OS/390 make program, you can reference it explicitly with command /bin/make.

```
# gmake install file
scriptName=`basename $0`
if [ $# != 2 ]; then
  echo "Usage: $scriptName archive install_dir"
  exit
fi
archive=$1
installDir=$2

# unwind and build
echo "$archive was installed by $scriptName on `date`"
echo "unwinding via the command 'pax -o from=ISO8859-1,to=IBM-1047 -rzf
pax -o from=ISO8859-1,to=IBM-1047 -rzf $archive
ln -s make-3.76.1 make
cd make
chmod +w stamp-h.in
echo "creating makefile via the command './configure'"
CC=cc LDFLAGS=-Wl,EDIT=NO ./configure --prefix=$installDir
echo "building via the command 'make'"
make
rm CEE*

# set up symbolic links to bin
echo "copying executables and man pages to $installDir/bin and man ..."
cp make $installDir/bin
cp make.man $installDir/man/$LANG/cat1/make.1
echo "setting up symbolic links from gmake to make"
cd $installDir/bin
if [ -f make ]; then
  ln -s make gmake
fi
cd $installDir/man/$LANG/cat1
if [ -f make.1 ]; then
  ln -s make.1 gmake.1
fi
```

Figure 12-4 gmake.install

B

Sample IMS IVP with XML

This appendix lists the Java code for the IMS Java IVP, with the additions to use XML. The code can also be downloaded from the IBM Redbooks Web site as described in Appendix C, “Additional material” on page 137.

IVP

```
package imssam;

//
// CONV TITLE 'INSTALLATION VERIFICATION PROCEDURE - CONVERSATIONAL'
//-------------------------------------------------------------------
//
//  APPLICATION  :  IMS INSTALLATION VERIFICATION PROCEDURE
//                  CONVERSATIONAL MPP PROGRAM HDAM/VSAM
//  TRANSACTION  :  IVTCC
//  PSB          :  DFSIVP32
//  DATABASE     :  DFSIVD2
//
//-------------------------------------------------------------------
//
//        Licensed Materials - Property of IBM
//        "Restricted Materials of IBM"
//        5655-158 (C) Copyright IBM Corp. 1999
//
//-------------------------------------------------------------------
//
//  INPUT:
//         TELEPHONE DIRECTORY SYSTEM
//         PROCESS CODE : CCCCCCCC
//         LAST NAME    : XXXXXXXXXX
//         FIRST NAME   : XXXXXXXXXX
//         EXTENTION#   : N-NNN-NNNN
//         INTERNAL ZIP : XXX/XXX
//
//  CCCCCCCC = COMMAND
//         ADD     = INSERT ENTRY IN DB
//         TADD    = NOT SUPPORTED. DEFAULT TO ADD
//         DELETE  = DELETE ENTRY FROM DB
//         UPDATE  = UPDATE ENTRY FROM DB
//         DISPLAY = DISPLAY ENTRY
//
//-------------------------------------------------------------------

import com.ibm.ims.base.*;
import com.ibm.ims.application.*;
import com.ibm.ims.db.*;
import java.sql.*;
```

```
/**
 * The IVP class is the Installation Verification Program which is
 * a conversational program that runs in the IMS MPP region.
 * This program works with the DFSIVD2 database, which defines
 * the phone book directory.
 **/

public class IVP extends IMSApplication {
    private IMSMessageQueue msgQ = null;
    private Connection connection = null;
    static boolean appTraceOn = true;
    boolean isXMLOutput = false;
    static IMSTrace trace;
    static {
        IMSTrace.libTraceLevel=IMSTrace.TRACE_DATA3;
        IMSTrace.traceOn=true;
        trace=IMSTrace.currentTrace();
    }

/** Constructs a new IVP object. */
    public IVP()
    {
    }
/**
 * This is the IVP application program.
 * It first reads the SPA message and determines the state of the
 * conversation program. It then reads the user input request and
 * process the database command and send a response back to the
 * user according to the status of the processing.
 */
    public void doBegin() throws IMSException
    {
        if (IVP.appTraceOn) IMSTrace.currentTrace().logEntry("IVP.doBegin");

        msgQ = new IMSMessageQueue();
        InputMessage inputMessage = new InputMessage();
        OutputMessage outputMessage = new OutputMessage();
        SPAMessage spaMessage = new SPAMessage();
        boolean msgReceived = false;
        String procCode = null;
        String errorMessage = null;
        ParseXMLInput xmlParser = null;

        try {
            Class.forName("com.ibm.ims.db.DLIDriver");
            connection = DriverManager.getConnection("jdbc:dli:imssam.IVPDatabaseView");
            // Get the SPA data
            msgReceived = msgQ.getUniqueMessage(spaMessage);
            if (!msgReceived)
                errorMessage = "UNABLE TO READ SPA";
        } catch (IMSException e) {
            if (e.getStatusCode() != JavaToDLI.MESSAGE_QUEUED_PRIOR_TO_LAST_START)
                throw e;
        } catch (Exception e) {

            e.printStackTrace();
            errorMessage = "Could not load DLIDriver";
        }
        if (errorMessage == null) {

            if (!msgQ.getNextMessage(inputMessage)) {

                // No input message received
                errorMessage = "NO INPUT MESSAGE";
            } else {
                procCode = inputMessage.getString("ProcessCode").trim();

                if (procCode.indexOf("<?xml") != -1) {
                    xmlParser = new ParseXMLInput();
                    isXMLOutput = true;
                    try {
                        xmlParser.parse(inputMessage);
                    } catch (Exception e) {
                        e.printStackTrace();
                        outputMessage.setString("Message", "XML Parse error. " + e.toString());
                    }

                    inputMessage = xmlParser.getConvertedInput();

                }
                procCode = inputMessage.getString("ProcessCode").trim();
                if ((spaMessage.getShort("SessionNumber")==0)
                    && (!inputMessage.getString("ProcessCode").trim().equals("END"))
                    && (inputMessage.getString("LastName").trim().equals(""))) {
                    // New Conversation. User has to specify last name.
                    errorMessage = "LAST NAME WAS NOT SPECIFIED";
                } else {
                    try {

                        procCode = inputMessage.getString("ProcessCode").trim();
                        if (spaMessage.getShort("SessionNumber") != 0) {       // Check if process code is specified.
If not, use SPA data as default
                            if (procCode.equals(""))
                                inputMessage.setString("ProcessCode",spaMessage.getString("ProcessCode"));
                            // Check if last name is specified. If not, use SPA data as default
                            if (inputMessage.getString("LastName").equals(""))
                                inputMessage.setString("LastName",spaMessage.getString("LastName"));
                        }
                        // Determine which command was requested

                        if (procCode.equals("ADD") || procCode.equals("TADD"))
                            add(inputMessage, outputMessage);                        // Add the entry to the database
                        else if (procCode.equals("DELETE"))
                            delete(inputMessage, outputMessage);            // Delete the entry from the database
                        else if (procCode.equals("UPDATE"))
                            update(inputMessage, outputMessage);            // Update the entry in the database
                        else if (procCode.equals("DISPLAY"))
                            display(inputMessage, outputMessage);           // Display the data from the database
                        else
                            errorMessage = "PROCESS CODE WAS INVALID";
                    } catch (Exception e) {
                        outputMessage.setString("Message", "Request failed. " + e.toString());
                    }
                }
            }
        }

        // Update the spa and send a response back to the user.
        updateSPA(inputMessage, spaMessage, procCode);

        if (errorMessage != null)
            outputMessage.setString("Message", errorMessage);
        reply(outputMessage, procCode);
        // Close the connection
        try {
            connection.close();
        } catch (Exception e) {
            outputMessage.setString("Message", "Connection close failure" + e.toString());
        }

        // Commit this transaction.
        IMSTransaction.getTransaction().commit();
        if (IVP.appTraceOn) IMSTrace.currentTrace().logExit("IVP.doBegin");

    }
/** main entry point of the IVP program **/
    public static void main(String args[])
    {
        IVP ivp = new IVP();
        ivp.begin();
    }
/**
 * Send a response back to the user.
 * @param outputMessage - the output message to be sent to user
 * @param procCode - the process code for this transaction
 */
    public void reply(OutputMessage outputMessage, String procCode) throws IMSException
    {
        XMLOutput xmlOutputMessage;
        String xmlOutput;
        if (procCode.equals("END"))
            outputMessage.setString("Message","CONVERSATION HAS ENDED");
        if (isXMLOutput) {
            try {
                xmlOutput = outputMessage.toXML();
                xmlOutputMessage = new XMLOutput();
                xmlOutputMessage.clearBody();
                xmlOutputMessage.setString("Results", xmlOutput);
                msgQ.insertMessage(xmlOutputMessage);
            } catch (Exception e) {
                outputMessage.setString("Message","Can create XML output");

            }
        } else {
            msgQ.insertMessage(outputMessage);
        }
    }
/**
 * Update the output message.
 * @param result - the segment info from the database
 * @param outputMessage - the output message to be updated
 */
    public void setOutput(ResultSet result, OutputMessage outputMessage) throws SQLException, IMSException
    {
        // Set output data fields
        outputMessage.setString("LastName",result.getString("LastName"));
        outputMessage.setString("FirstName",result.getString("FirstName"));
        outputMessage.setString("Extension",result.getString("Extension"));
        outputMessage.setString("ZipCode",result.getString("ZipCode"));
    }
/**
 * Update the output message.
 * @param inputMessage - the input message from the user
 * @param outputMessage - the output message to be updated
 */
    public void setOutput(InputMessage inputMessage, OutputMessage outputMessage) throws IMSException
    {
        // Set output data fields
        outputMessage.setString("ProcessCode",inputMessage.getString("ProcessCode"));
        outputMessage.setString("LastName",inputMessage.getString("LastName"));
        outputMessage.setString("FirstName",inputMessage.getString("FirstName"));
        outputMessage.setString("Extension",inputMessage.getString("Extension"));
        outputMessage.setString("ZipCode",inputMessage.getString("ZipCode"));
    }
/**
 * Add an entry into the database.
 * All fields for the segment (Last Name, First Name, Extension Number,
 * and Internal Zip Code) should be entered for insert.
 * @param inputMessage - the input message from the user
 * @param outputMessage - the output message for this transaction
 */
    public void add(InputMessage inputMessage, OutputMessage outputMessage)  throws IMSException
    {
        String firstName = inputMessage.getString("FirstName").trim();
        String lastName = inputMessage.getString("LastName").trim();
        String extension = inputMessage.getString("Extension").trim();
        String zip = inputMessage.getString("ZipCode").trim();
        // Check for input fields
        if (firstName.equals("")
            || lastName.equals("")
            || extension.equals("")
            || zip.equals("")) {       // Not enough input data
            outputMessage.setString("Message","DATA IS NOT ENOUGH. PLEASE KEY IN MORE");
        } else {

            try {       // Insert the new segment into the database
                Statement statement = connection.createStatement();
                statement.executeUpdate("INSERT INTO PhoneBookSegment (FirstName, LastName, Extension, ZipCode) "
                                        + "VALUES ('" + firstName + "', '" + lastName + "', '" + extension + "', '" +
zip + "')");
                outputMessage.setString("Message","ENTRY WAS ADDED");
            } catch (Exception e) {       // Error in inserting
                e.printStackTrace();
                outputMessage.setString("Message","ADDITION OF ENTRY HAS FAILED");
            }
        }
        setOutput(inputMessage, outputMessage);
    }
/**
 * Delete an entry from the database.
 * @param inputMessage - the input message from the user
 * @param outputMessage - the output message for this transaction
 */
    public void delete(InputMessage inputMessage, OutputMessage outputMessage) throws IMSException
    {
        try {       // Delete the segment
            Statement statement = connection.createStatement();
            statement.executeUpdate("DELETE FROM PhoneBookSegment "
                                    + "WHERE LastName = '" + inputMessage.getString("LastName") + "'");
            outputMessage.setString("Message","ENTRY WAS DELETED");
        } catch (Exception e) {       // Error in deleting the data
            outputMessage.setString("Message","DELETION OF ENTRY HAS FAILED");
        }
        setOutput(inputMessage, outputMessage);
    }
/**
 * Display a certain entry from the database.
 * @param inputMessage - the input message from the user
 * @param outputMessage - the output message for this transaction
 */
    public void display(InputMessage inputMessage, OutputMessage outputMessage) throws IMSException
    {
        try {
            Statement statement = connection.createStatement();
            ResultSet result = statement.executeQuery("SELECT * FROM PhoneBookSegment "
                                                  + "WHERE LastName = '" + inputMessage.getString("LastName") +
"'");
            setOutput(result, outputMessage);
            outputMessage.setString("Message","ENTRY WAS DISPLAYED");
        } catch (Exception e) {       // Set output data fields
            setOutput(inputMessage, outputMessage);
            outputMessage.setString("Message","SPECIFIED PERSON IS NOT FOUND");
        }
    }

/**
 * Update an entry in the database.
 * @param inputMessage - the input message from the user
 * @param outputMessage - the output message for this transaction
 */
    public void update(InputMessage inputMessage, OutputMessage outputMessage) throws IMSException
    {
        String firstName = inputMessage.getString("FirstName").trim();
        String lastName = inputMessage.getString("LastName").trim();
        String extension = inputMessage.getString("Extension").trim();
        String zip = inputMessage.getString("ZipCode").trim();
        // Check for input fields
        if (firstName.equals("")
            || lastName.equals("")
            || extension.equals("")
            || zip.equals("")) {       // Requested input data not enough
            outputMessage.setString("Message","DATA IS NOT ENOUGH. PLEASE KEY IN MORE");
        } else {
            try {       // Update the new entry
                Statement statement = connection.createStatement();
                statement.executeQuery("UPDATE PhoneBookSegment SET FirstName = '" + firstName
                                + "', Extension = '" + extension + "', ZipCode = '" + zip + "' "
                                + "WHERE LastName = '" + inputMessage.getString("LastName") + "'");
                outputMessage.setString("Message","ENTRY WAS UPDATED");
            } catch (Exception e) {       // Error with the update
                outputMessage.setString("Message","UPDATE OF ENTRY HAS FAILED");
            }
        }
        // Update the output data
        setOutput(inputMessage, outputMessage);
    }
/**
 * Update the SPA message.
 * @param inputMessage - the input message from the user
 * @param spaMessage - the spa message to be updated
 */
    public void updateSPA(InputMessage inputMessage, SPAMessage spaMessage, String procCode) throws IMSException
    {
        if (!procCode.equals("END")) {
            // Set spa data fields
            spaMessage.setString("ProcessCode",inputMessage.getString("ProcessCode"));
            spaMessage.setString("LastName",inputMessage.getString("LastName"));
            spaMessage.setString("FirstName",inputMessage.getString("FirstName"));
            spaMessage.setString("Extension",inputMessage.getString("Extension"));
            spaMessage.setString("ZipCode",inputMessage.getString("ZipCode"));
            spaMessage.incrementSessionNumber();
            msgQ.insertMessage(spaMessage);
        } else
            msgQ.insertMessage(spaMessage, true);

    }
}
```

IVPDatabaseView

```
package imssam;

import com.ibm.ims.db.*;
/**
 *  TELEPCB contains the segment info for the database DFSIVD2
 */
public class IVPDatabaseView extends com.ibm.ims.db.DLIDatabaseView {
    static DLISegmentInfo[] segments = {new DLISegmentInfo(new imssam.PhoneBookSegment(),DLIDatabaseView.ROOT)};
/**
 * Constructs a new IVPDatabaseView object.
 */
    public IVPDatabaseView() {
        super("TELEPCB", segments);
    }
}
```

InputMessage

```
package imssam;

import com.ibm.ims.base.*;
/** This is the Input Message class for receiving message from the message queue */
public class InputMessage extends com.ibm.ims.application.IMSFieldMessage {
    static DLITypeInfo[] fieldInfo = {
        new DLITypeInfo("Reserved",DLITypeInfo.CHAR,1,4),
        new DLITypeInfo("ProcessCode",DLITypeInfo.CHAR,5,8),
        new DLITypeInfo("LastName",DLITypeInfo.CHAR,13,10),
        new DLITypeInfo("FirstName",DLITypeInfo.CHAR,23,10),
        new DLITypeInfo("Extension",DLITypeInfo.CHAR,33,10),
        new DLITypeInfo("ZipCode",DLITypeInfo.CHAR,43,7),
        new DLITypeInfo("XMLRequest", DLITypeInfo.CHAR,   5, 1800)
    };
/**
 * Constructs an InputMessage. It allocates the byte array
 * for the input message. Total data length is 49 bytes.
 * The input message data has the following layout format
 * as defined in the MID:
 * Reserved         ( 4 bytes)
 * Process Code     ( 8 bytes)
 * Input Data :     (37 bytes total)
 *       - Last Name         (10 bytes)
 *       - First Name        (10 bytes)
 *       - Extension Number  (10 bytes)
 *       - Internal Zip Code ( 7 bytes)
 * XMLRequest remaps the above storage starting at 5
 */
    public InputMessage()
    {
        super(fieldInfo,1804, false);
    }
}
```

OutputMessage

```
package imssam;

import com.ibm.ims.base.*;
/**
 * This is the OuputMessage class which the IVP program uses to send output
 * to the message queue.
 */
public class OutputMessage extends com.ibm.ims.application.IMSFieldMessage {
    static DLITypeInfo[] fieldInfo = {
        new DLITypeInfo("Message",DLITypeInfo.CHAR,1,40),
        new DLITypeInfo("ProcessCode",DLITypeInfo.CHAR,41,8),
        new DLITypeInfo("LastName",DLITypeInfo.CHAR,49,10),
        new DLITypeInfo("FirstName",DLITypeInfo.CHAR,59,10),
        new DLITypeInfo("Extension",DLITypeInfo.CHAR,69,10),
        new DLITypeInfo("ZipCode",DLITypeInfo.CHAR,79,7),
        new DLITypeInfo("SegmentNumber",DLITypeInfo.CHAR,86,4)
    };
/**
 * Constructs an OutputMessage. It allocates the byte array
 * for the output message. Total data length is 89 bytes.
 * The output message data has the following layout format:
 * Output Message (40 bytes)
 * Process Code   ( 8 bytes)
 * Output Data    (37 bytes)
 *       - Last Name         (10 bytes)
 *       - First Name        (10 bytes)
 *       - Extension Number  (10 bytes)
 *       - Internal Zip Code ( 7 bytes)
 * Segment Number ( 4 bytes)
 */
    public OutputMessage()
    {
        super(fieldInfo,89, false);
    }
    public StringBuffer getXMLData() throws Exception {
        StringBuffer xmlBuffer = new StringBuffer(1400);
        xmlBuffer.append("<Message>").append(getString("Message").trim()).append("</Message>");
        xmlBuffer.append("<ProcessCode>").append(getString("ProcessCode").trim()).append("</ProcessCode>");
        xmlBuffer.append("<LastName>").append(getString("LastName").trim()).append("</LastName>");
        xmlBuffer.append("<FirstName>").append(getString("FirstName").trim()).append("</FirstName>");
        xmlBuffer.append("<Extension>").append(getString("Extension").trim()).append("</Extension>");
        xmlBuffer.append("<ZipCode>").append(getString("ZipCode").trim()).append("</ZipCode>");
        xmlBuffer.append("<SegmentNumber>").append(getString("SegmentNumber").trim()).append("</SegmentNumber>");

        return(xmlBuffer);
    }
    public String toXML() throws Exception {
        StringBuffer xmlBuffer = new StringBuffer(1400);
        xmlBuffer.append("<?xml version=\"1.0\"?>");
        xmlBuffer.append("<output>");
        xmlBuffer.append(getXMLData());
        xmlBuffer.append("</output>");

        return new String(xmlBuffer);

    }

}
```

ParseXMLInput

```
package imssam;

/**
 * This class converts an XML document */

import java.lang.reflect.Constructor;
import java.io.StringReader;
import org.xml.sax.HandlerBase;
import org.xml.sax.AttributeList;
import org.xml.sax.InputSource;
import com.ibm.ims.base.IMSTrace;
import org.apache.xerces.parsers.SAXParser;

public class ParseXMLInput extends HandlerBase {
    boolean isProcessCode;
    boolean isFirstName;
    boolean isLastName;
    boolean isExtension;
    boolean isZipCode;
    InputMessage  xmlInput;
    InputMessage  parsedXML;

    /* Default parser */
    private static final String
    DEFAULT_PARSER_NAME = "org.apache.xerces.parsers.SAXParser";

    public void characters(char[] arg1, int start, int length) {


        if (IVP.appTraceOn) IMSTrace.currentTrace().logEntry("ParseXMLInput.characters");

        String data = new String(arg1, start, length);
        try {
            if (isProcessCode) {
                parsedXML.setString("ProcessCode", data);
            } else if (isFirstName) {
                parsedXML.setString("FirstName", data);
            } else if (isLastName) {
                parsedXML.setString("LastName", data);
            } else if (isExtension) {
                parsedXML.setString("Extension", data);
            } else if (isZipCode) {
                parsedXML.setString("ZipCode", data);
            }
        } catch (Exception e) {

            IMSTrace.currentTrace().logData(data, e.toString());
            e.printStackTrace();
            parsedXML = null;
        } finally {

            if (IVP.appTraceOn) IMSTrace.currentTrace().logExit("ParseXMLInput.characters");
        }
    }
    public void endElement(String element) {
        if (IVP.appTraceOn) {
            IMSTrace.currentTrace().logEntry("ParseXMLInput.endElement");
            IMSTrace.currentTrace().logData("element", element);
        }

        if (element.equals("ProcessCode")) {
            isProcessCode = false;
        } else if (element.equals("FirstName")) {
            isFirstName = false;
        } else if (element.equals("LastName")) {
            isLastName = false;
        } else if (element.equals("Extension")) {
            isExtension = false;
        } else if (element.equals("ZipCode")) {
            isZipCode = false;
        }

        if (IVP.appTraceOn) IMSTrace.currentTrace().logExit("ParseXMLInput.endElement");
    }
    public InputMessage getConvertedInput ( ) {

        return parsedXML;
    }
    public void parse(InputMessage xmlInput) throws Exception {

        if (IVP.appTraceOn) IMSTrace.currentTrace().logEntry("ParseXMLInput.parse");
        try {

            String xmlDocument = xmlInput.getString("XMLRequest").trim();
            SAXParser parser = (SAXParser) Class.forName(DEFAULT_PARSER_NAME).newInstance();
            parser.setDocumentHandler(this);
            StringReader xmlReader = new StringReader(xmlDocument);
            parser.parse(new InputSource(xmlReader));

        } finally {

            if (IVP.appTraceOn) IMSTrace.currentTrace().logExit("ParseXMLInput.parse");
        }


    }
    /** Start document. */
    public void startDocument() {

        if (IVP.appTraceOn) IMSTrace.currentTrace().logEntry("ParseXMLInput.startDocument");
        parsedXML = new  InputMessage();
        isProcessCode = false;
        isFirstName = false;
        isLastName = false;
        isExtension = false;
        isZipCode = false;
        if (IVP.appTraceOn) IMSTrace.currentTrace().logExit("ParseXMLInput.startDocument");
    }
    public void startElement(String element, AttributeList attrlist) {


        if (IVP.appTraceOn) {
            IMSTrace.currentTrace().logEntry("ParseXMLInput.startElement");
            IMSTrace.currentTrace().logData("element", element);
        }
        if (element.equals("ProcessCode")) {
            isProcessCode  = true;
        } else if (element.equals("FirstName")) {
            isFirstName = true;
        } else if (element.equals("LastName")) {
            isLastName = true;
        } else if (element.equals("Extension")) {
            isExtension = true;
        } else if (element.equals("ZipCode")) {
            isZipCode = true;
        }
        if (IVP.appTraceOn) IMSTrace.currentTrace().logExit("ParseXMLInput.startElement");
    }
    public void endDocument() {

        if (IVP.appTraceOn) IMSTrace.currentTrace().logEntry("ParseXMLInput.endDocument");
        isProcessCode = false;
        isFirstName = false;
        isLastName = false;
        isExtension = false;
        isZipCode = false;
        if (IVP.appTraceOn) IMSTrace.currentTrace().logExit("ParseXMLInput.endElement");
    }
}
```

PhoneBookSegment

```
package imssam;

import com.ibm.ims.base.*;
/**
 * This is the PhoneBookSegment class which represents the A1111111 segment
 * for the DFSIVD2 database which is a root only database having the
 * following segment layout:
 *        BYTES  1-10  LAST NAME (CHARACTER) - KEY
 *        BYTES  11-20 FIRST NAME (CHARACTER)
 *        BYTES  21-30 INTERNAL PHONE NUMBER (NUMERIC)
 *        BYTES  31-37 INTERNAL ZIP (CHARACTER)
 *        BYTES  38-40 RESERVED
 */
public class PhoneBookSegment extends com.ibm.ims.db.DLISegment {
    static  DLITypeInfo[] typeInfo = {
        new DLITypeInfo("LastName", DLITypeInfo.CHAR,1,10,"A1111111"),
        new DLITypeInfo("FirstName", DLITypeInfo.CHAR,11,10),
        new DLITypeInfo("Extension", DLITypeInfo.CHAR,21,10),
        new DLITypeInfo("ZipCode", DLITypeInfo.CHAR,31,7)
    };
/**
 * Constructs an PhoneBookSegment class. Allocates the byte array for the
 * segment data. Total segment length is 40 bytes.
 */
    protected PhoneBookSegment()
    {
        super("A1111111",typeInfo,40);
    }
}
```

SPAMessage

```
package imssam;

import com.ibm.ims.base.*;
/**
 * This is the SPAMessage class which the IVP program uses to save data
 * between the conversation.
 */
public class SPAMessage extends com.ibm.ims.application.IMSFieldMessage {
    static DLITypeInfo[] fieldInfo = {
        new DLITypeInfo("SessionNumber",DLITypeInfo.SMALLINT,1,2),
        new DLITypeInfo("ProcessCode",DLITypeInfo.CHAR,3,8),
        new DLITypeInfo("LastName",DLITypeInfo.CHAR,11,10),
        new DLITypeInfo("FirstName",DLITypeInfo.CHAR,21,10),
        new DLITypeInfo("Extension",DLITypeInfo.CHAR,31,10),
        new DLITypeInfo("ZipCode",DLITypeInfo.CHAR,41,7),
        new DLITypeInfo("Reserved",DLITypeInfo.CHAR,48,19)
    };
/**
 * Constructs a SPAMessage. It allocates the byte array
 * for the spa message. Total data length is 66 bytes.
 * The output message data has the following layout format:
 * Session Number (2 bytes)
 * Process Code        (8 bytes)
 * SPA Data           (37 bytes total)
 *       - Last Name         (10 bytes)
 *       - First Name        (10 bytes)
 *       - Extension Number  (10 bytes)
 *       - Internal Zip Code ( 7 bytes)
 * Reserved           (19 bytes)
 */
    public SPAMessage()
    {
        super(fieldInfo, 66, true);
    }
/**
 * Increment the session number for the conversation.
 */
    public void incrementSessionNumber() throws IMSException
    {
        setShort("SessionNumber",(short)(getShort("SessionNumber")+1));
    }
}
```

XMLOutput

```
package imssam;

import com.ibm.ims.base.*;
/**
 * This is the OuputMessage class which the IVP program uses to send output
 * to the message queue as XML .
 */
public class XMLOutput extends com.ibm.ims.application.IMSFieldMessage {
    static DLITypeInfo[] fieldInfo = {
        new DLITypeInfo ("Results",      DLITypeInfo.CHAR, 1, 1500)
    };
    public XMLOutput()
    {
        super(fieldInfo, 1500 , false);
    }

}
```

C

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG246285

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24-6285.

Using the Web material

The additional Web material that accompanies this redbook is packaged into compressed file SG246582.zip. The file will decompress into three subdirectories.

Directory name	*Description*
CICSSampleApp	Contains the SampleApp described in Chapter 8. “A CICS SampleApp with XML” on page 77.
IMSSampleApp	Contains the SampleApp described in Chapter 11, “An IMS SampleApp with XML” on page 103.
IMSXMLIVPSample	Contains the Sample IVP described in Chapter 12, “IMS Java IVP Sample using XML” on page 115.

Files in directory CICSSampleApp are:

cicsxmlsample.jar	The servlets code, parser and EAB java beans for the application.
pcXMLInput.html	HTML files are for the XML input for the PC transaction.
pcXMLOutput.jsp	Represents XML output file for the PC transaction.

pqXMLInput.html	HTML files are for the XML input for the PQ transaction.
pqXMLOutput.jsp	Represents XML output file for the PQ transaction.
error.jsp	Standard error page displayed by the sample application in case of system error.

Files in directory IMSSampleApp are:

imsxmlsample.jar	The servlets code, parser and EAB java beans for the application.
pcXMLIMSInput.html	HTML files are for the XML input for the PC transaction.
pcXMLIMSOutput.jsp	Represents the XML output file for the PC transaction.
pqXMLIMSInput.html	HTML files are for the XML input for the PQ transaction.
pqXMLIMSOutput.jsp	Represents the XML output file for the PQ transaction.
error.jsp	Standard error page displayed by the sample application in case of system error.

Files in directory IMSXMLIVPSample are:

IVP.java	Extends the IMSApplication base class and contains all of the business logic Java source.
InputMessage.java	Input message.
OutputMessage.java	Output message.
XMLOutput.java	XML tagged output message.
ParseXMLInput.java	Parses and extracts input data Java source.
PhoneBookSegment.java	Maps the A1111111 segment of the DFSIVD2 database.
IVPDatabaseView.java	Segment information for the database DFSIVD2.
SPAMessage.java	Message used to store data between conversations.

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	Approximately 0.4 MB
Operating System:	Windows 9x, NT or 2000
Processor:	Intel 386 or higher
Memory:	64 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder. Make sure that you specify the path info or folder name option. This will create three directories, CICSXMLSample, IMSXMLSample and IMSIVPSample and decompress the files into each folder. For CICSXMLSample and IMSXMLSample the Java code provided must be customized to match the specifications (transaction name and message area) of your own CICS or IMS application.

Special notices

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 142.

- *Integrating XML with DB2 XML Extender and DB2 Text Extender,* SG24-6130
- *The XML Files: Using XML for Business-to-Business and Business-to-Consumer Applications,* SG24-6104
- *IMS Version 7 and Java Application Programming,* SG24-6123
- *Revealed! Architecting Web Access to CICS,* SG24-54662
- *e-business Enablement Cookbook for OS/390 Volume I: Technology Introduction,* SG24-5664
- *e-business Enablement Cookbook for OS/390 Volume II: Infrastructure for Java-based Solutions,* SG24-5981
- *e-business Enablement Cookbook for OS/390 Volume III: Java Development,* SG24-5980
- *CICS Transaction Gateway V3.1 The WebSphere Connector for CICS,* SG24-6133
- *Programming with VisualAge for Java Version 3.5,* SG24-5264

IBM Publications

- *CICS Transaction Gateway Version 4.0, OS/390 Gateway Administration,* SC34-5935
- *CICS Transaction Gateway Version 4.0, Gateway Programming,* SC34-5938
- *IMS V7 Installation Vol. 1: Installation and Verification*, GC26-9429

Other resources

These publications are also relevant as further information sources:

- *XML in Plain English*, by Eddy, S.E., et al., published by IDG Books Worldwise, ISBN 0764570064
- *XML and Java Developing Web Applications*, by Maruyama, H., Tamura, K., and Uramoto, N., published by Addison-Wesley ISBN 0201485435
- *Java Performance and Scalability, Volume 1: Server-Side Programming Techniques*, Bulka, D., published by Addison-Wesley, ISBN 0201704293

Referenced Web sites

These Web sites are also relevant as further information sources:

- `http://www.ibm.com/xml/`

  XML Zone

- `http://alphaworks.ibm.com`

  The mission of alphaWorks is to provide developers with direct access to IBM emerging “alpha code” technologies

How to get IBM Redbooks

Search for additional Redbooks or redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also 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.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Abbreviations and acronyms

ANSI	American National Standards Institute
API	Application Programming Interface
CGI	Common Gateway Interface
CLI	Call Level Interface
CORBA	Common Objects Request Broker Architecture
DLL	Dynamic Link Library
DNS	Domain Name Server
DOM	Document Object Model
DTD	Document Type Definition
EAB	Enterprise Access Builder
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
IIOP	Internet Inter-Orb Protocol
IMAP	Internet Message Access Protocol
IPC	Inter-Process Communication
ITSO	International Technical Support Organization
JAR	Java archive
JDBC	Java Data Base Connection
JDK	Java Development Kit
JIT	Just In Time compiler (in Java)
JVM	Java Virtual Machine
JSP	Java Server Page
LDAP	Lightweight Directory Access Protocol
LE	Language Environment
MathML	Mathematical Markup Language
NLS	National Language Support
ODBC	Open Database Connectivity
OO	Object Oriented
ORB	Object Request Broker
UDDI	Universal Description, Discovery, and Integration
VIPA	Virtual IP Addressing
RDF	Resource Description Framework
SAX	Simple API for XML
SMIL	Synchronized Multimedia Integration Language
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
URI	Universal Resource Identifier
URL	Universal Resource Locator
USS	UNIX System Services
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	eXtensible Markup Language
XHTML	eXtensible Hyper Text Markup Language
XLink	XML Linking Language
XPointer	XML Pointer Language
XPath	XML Path Language
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations

Index

A
Apache XML project 44
ASCII 74

B
BuildHashMap 85, 110, 111

C
Cascading Style Sheets 21
CCF 68, 71, 73, 78, 82
character encoding schemes 74
CICS Transaction Gateway 28
COBOL structure 82
code page 2, 74
COMMAREA 68
CSS 21, 22

D
Document Object Model 38
Document Type Definition 14
DocumentHandler 36, 37
DOM 110
DOMParser 40
DTD 14, 16
DTDHandler 36

E
EAB 82
EBCDIC 74
ebXML 5
Electronic Data Interchange (EDI) 5
ErrorHandler 36
Experimental API's 44

I
IMS Connect 101
IMS Connector 28
IXM4C33 49

J
Java Bean components 29
Java Server Pages 29

M
MathML 26

N
Namespaces 18

O
OASIS 5
Object pooling 83, 109

P
ParserFactory 36
Parsing Considerations 83
Public API's 44

R
Redbooks Web site 142
    Contact us viii

S
SAX 110
Servlets 28
SMIL 26
SOAP 7
Strictly Conforming XHTML 24
SYS1.SIXMEXP 49, 51

U
UDDI 7
Unicode 75, 101
UTF-16 75
UTF-32 75
UTF-8 75

W
webapp classpath 53
WebSphere Studio Application Developer 53

X
XHTML 23, 24
XML Schema 19
XML4J 31, 32, 35, 44
XPath 20
XPointer 20
XSL 21, 23
XSLT 23

Redbooks

Using XML on z/OS and OS/390 for Application Integration

(0.2"spine)
0.17"<->0.473"
90<->249 pages

Using XML on z/OS and OS/390 for Application Integration

XML basics

Using the XML parser with z/OS and OS/390 applications

Sample Java code for CICS and IMS applications

This IBM Redbook will help you to understand how to integrate XML technology with business applications on z/OS. We start by providing a brief tutorial on XML, describing the advantages of its use. We explain its positioning, with emphasis on Java and the IBM WebSphere Application Server.

We then document XML implementation on z/OS and describe the XML Toolkit for z/OS and OS/390, from installation to usage.

We present examples of using XML on existing CICS or IMS applications, with a discussion of design alternatives, and explain how to install implementations in the WebSphere Application Server on z/OS. Finally, we provide sample Java code.

This redbook shows developers how XML can be used to communicate with existing CICS or IMS back office applications, as well as how it can be used to design new applications on z/OS.

SG24-6285-00 ISBN 0738423165