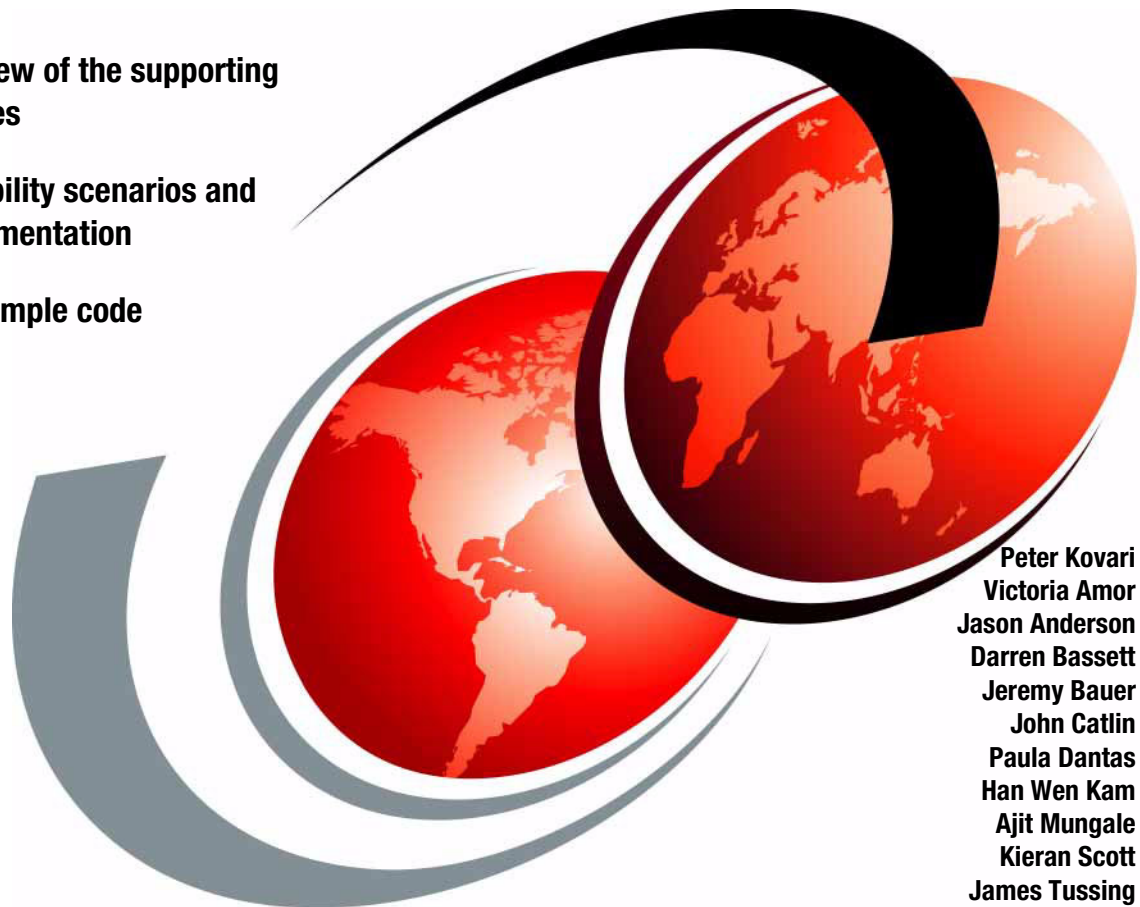


WebSphere and .NET Coexistence

In-depth view of the supporting
technologies

Interoperability scenarios and
their implementation

Working sample code



Peter Kovari
Victoria Amor
Jason Anderson
Darren Bassett
Jeremy Bauer
John Catlin
Paula Dantas
Han Wen Kam
Ajit Mungale
Kieran Scott
James Tussing



International Technical Support Organization

WebSphere and .NET Coexistence

March 2004

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (March 2004)

This edition applies to WebSphere Application Server V5.02 on AIX, Linux, Windows 2000 Server, Windows 2003 Server.

© Copyright International Business Machines Corporation 2004. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks	x
Preface	xi
The team that wrote this redbook	xii
Become a published author	xv
Comments welcome	xv
Part 1. Introduction	1
Chapter 1. J2EE introduction	3
1.1 Architecture	5
1.1.1 Overall architecture	5
1.1.2 Layer technologies (application architecture)	6
1.1.3 Standard support	8
1.1.4 Platform support	8
1.1.5 Programming languages	9
1.1.6 Deployment units	9
1.1.7 Runtime execution environment	11
1.1.8 Life cycle management	13
1.1.9 Remote object discovery	14
1.1.10 Remote Method Invocation	14
1.1.11 Web Services	15
1.1.12 Transaction management	16
1.1.13 Security	17
1.1.14 Load balancing and failover	20
1.1.15 Application logging	20
1.2 Development	21
1.2.1 Writing a Java application using a text editor	22
1.2.2 WebSphere Studio Application Developer (IDE)	24
1.3 Testing	29
1.3.1 WebSphere Studio Application Developer	29
1.4 Deployment	33
1.4.1 Packaging J2EE applications	33
1.4.2 Deploying the packaged applications	35
1.5 Runtime	37
1.5.1 WebSphere Application Server	38
1.6 Administration	44

Chapter 2. .NET introduction	49
2.1 Architecture	50
2.1.1 Overall architecture	50
2.1.2 Layered services (application architecture)	55
2.1.3 Standard support	59
2.1.4 Platform support	59
2.1.5 Programming languages	60
2.1.6 Deployment units	60
2.1.7 Runtime execution environment	63
2.1.8 Life cycle management	65
2.1.9 Remote object discovery	68
2.1.10 Remote invocation	68
2.1.11 Web Services	69
2.1.12 Transaction management	69
2.1.13 Security	70
2.1.14 Load balancing and failover	72
2.1.15 Application logging	73
2.1.16 Versioning	74
2.2 Development	75
2.2.1 Writing a C# application using text editor	75
2.2.2 Microsoft Visual Studio .NET (IDE)	77
2.2.3 Source code management	81
2.3 Testing	82
2.3.1 Debugging and unit testing	82
2.3.2 Performance and load testing	85
2.4 Deployment	88
2.5 Runtime	90
2.6 Administration	90
Chapter 3. An architectural model for coexistent applications	93
3.1 Coexisting heterogeneous technologies	94
3.1.1 Layered application model	95
3.1.2 Concentric layered application model	98
3.1.3 Bridging layers and address spaces	99
3.1.4 Interoperation layer abstraction	101
3.1.5 Summary	105
Part 2. Scenarios	107
Chapter 4. Technical coexistence scenarios	109
4.1 Introduction	110
4.2 Fundamental interaction classifications	111
4.2.1 Stateful synchronous interaction	112
4.2.2 Stateless synchronous interaction	115

4.2.3 Stateless asynchronous interaction	117
4.2.4 Stateful asynchronous interaction	120
4.2.5 RPC interface style	123
4.2.6 Document interface style	124
4.2.7 Argument by value paradigm	125
4.2.8 Argument by reference paradigm	127
4.2.9 Distributed object architecture	129
4.2.10 Message Oriented Architecture	130
4.2.11 Service-oriented architecture	131
4.2.12 Conclusions and recommendations	131
4.3 Layer interaction classifications	132
4.3.1 Interaction case a: client logic to client logic	138
4.3.2 Interaction case b: client logic to presentation logic	145
4.3.3 Interaction case c: client logic to business logic	152
4.3.4 Interaction case d: presentation logic to presentation logic	160
4.3.5 Interaction case e: presentation logic to business logic	167
4.3.6 Interaction case f: business logic to business logic	174
4.3.7 Interaction case g: business logic to resource	182
4.3.8 Interaction case h: resource to resource	195
4.3.9 Conclusion and recommendations	201
4.4 Technical solution mapping	202
4.4.1 Stateful synchronous integration solution candidates	203
4.4.2 Stateless synchronous integration solution candidates	214
4.4.3 Stateful asynchronous integration solution candidates	219
4.4.4 Other potential candidate technical solutions (to be proven)	223
4.4.5 Some last resource integration technologies	226
Chapter 5. Scenario: Asynchronous	229
5.1 Problem definition	230
5.1.1 Description of the problem	234
5.1.2 Considerations	234
5.2 Solution model	237
5.2.1 A solution to the problem	238
5.2.2 Simple scenario details	240
5.2.3 .NET consumer to WebSphere service provider	242
5.2.4 WebSphere consumer to .NET service provider	254
Chapter 6. Scenario: Synchronous stateful	261
6.1 Problem definition	262
6.1.1 Description of the problem	262
6.1.2 Considerations	266
6.1.3 Constraints	274
6.1.4 Recommendations	276

6.2	Solution model using the ActiveX Bridge	279
6.2.1	A solution to the problem	279
6.2.2	Simple scenario details	279
6.3	Solution model using the Interface Tool for Java	289
Chapter 7. Scenario: Synchronous stateless (WebSphere producer and .NET consumer)		
7.1	Problem definition	299
7.1.1	Description of the problem	299
7.1.2	Considerations	300
7.2	Solution model.	303
7.2.1	A solution to the problem	303
7.2.2	Service provider	303
7.2.3	Service consumer	313
7.3	Extended solution	321
7.4	Recommendations	326
Chapter 8. Scenario: Synchronous stateless (WebSphere consumer and .NET producer)		
8.1	Solution model.	330
8.1.1	A solution to the problem	330
8.1.2	Service provider	331
8.1.3	Service consumer	346
8.1.4	Test	358
8.2	Extended solution model.	359
Chapter 9. Scenario: Web interoperability		
9.1	Introduction	368
9.2	Shared presentation components	368
9.2.1	Configuring Microsoft IIS for shared presentation.	370
9.3	Session state interoperability	376
9.3.1	Problem definition	376
9.3.2	WebSphere Application Server session management	377
9.3.3	Microsoft .NET session management	381
9.3.4	Considerations	385
9.3.5	Recommendations	395
9.4	Data propagation.	395
9.4.1	Problem definition	395
9.4.2	Description of the problem	396
9.4.3	Considerations	397
9.4.4	Solution model	402
9.4.5	URL redirection implementation	405
9.4.6	Form-based propagation implementation	412
9.4.7	Recommendations	418

9.5 Integrated security	418
9.5.1 WebSphere security	420
9.5.2 .NET security	420
9.5.3 Integrating authentication	421
9.5.4 Integrating authorization	425
Part 3. Guidelines	427
Chapter 10. Supporting technologies	429
10.1 Web Services	430
10.1.1 Technologies for Web Services	430
10.2 Client applications	439
10.2.1 Web browser	440
10.2.2 J2EE clients	441
10.2.3 Windows .NET clients	443
10.3 Server pages	443
10.3.1 Servlets and JSPs	443
10.3.2 ASP.NET	446
10.4 Distributed components	451
10.4.1 EJBs	452
10.4.2 .NET Remoting	454
10.5 Database access	456
10.5.1 EJBs	458
10.5.2 JDBC	459
10.5.3 ADO.NET	460
10.6 Messaging middleware	463
10.7 Back-end integration	466
10.7.1 J2C	466
10.7.2 .NET	466
10.8 Other integration technologies	467
10.8.1 ActiveX Bridge	468
10.8.2 IBM Interface Tool for Java	468
Chapter 11. Quality of service considerations	471
11.1 Scalability	472
11.1.1 WebSphere	472
11.1.2 .NET	475
11.2 Performance	478
11.2.1 WebSphere	478
11.2.2 .NET	480
11.3 Availability	482
11.3.1 WebSphere	483
11.3.2 .NET	483
11.4 Security	484

11.4.1 WebSphere	485
11.4.2 .NET	487
11.5 Transactionality	489
11.5.1 WebSphere	490
11.5.2 .NET	491
11.6 Manageability	493
11.6.1 WebSphere	493
11.6.2 .NET	496
11.7 Maintainability	498
11.7.1 WebSphere	498
11.7.2 .NET	499
11.8 Portability	499
11.8.1 WebSphere	500
11.8.2 .NET	500
11.9 Web Services	501
Part 4. Appendixes	511
Appendix A. Lotus Domino and .NET coexistence	513
A.1 Web Services integration	514
A.1.1 Domino provider, .NET consumer	515
A.1.2 .NET service provider, Domino service consumer	537
A.2 Using the COM interface.	538
A.2.1 Domino as a COM server, .NET as a client	540
Appendix B. Additional material	567
Locating the Web material	567
Using the Web material	568
System requirements for downloading the Web material	568
How to use the Web material	568
Abbreviations and acronyms	571
Related publications	573
IBM Redbooks	573
Other publications	573
Online resources	574
How to get IBM Redbooks	578
Help from IBM	578
Index	579

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Screen shot(s) reprinted by permission from Microsoft Corporation.




This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®	Notes®	Wave®
DB2®	OS/390®	WebSphere®
developerWorks®	OS/400®	XDE™
Domino Designer®	Redbooks (logo)  ™	z/OS®
Domino™	S/390®	zSeries®
ibm.com®	SAA®	alphaWorks®
IBM®	SecureWay®	HACMP™
Lotus Notes®	SupportPac™	IBM®
Lotus®	Tivoli®	Redbooks™
MQSeries®	 ™	IBM  ™

Rational is a registered trademark of International Business Machines Corporation and Rational Software Corporation, in the United States, other Countries or both.

Rational®

ClearCase®

The following terms are trademarks of other companies:

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, the Windows logo, Visual Studio, and the Visual Studio logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Preface

This IBM® Redbook explores the different coexistence scenarios for the WebSphere® and the .NET platforms. This book is a good source of information for solution designers and developers, application integrators and developers who wish to integrate solutions on the WebSphere and .NET platforms.

Part 1, “Introduction” on page 1 is a quick introduction to the J2EE (WebSphere) and .NET technologies. It also depicts a basic architectural model which can be used to represent both WebSphere and .NET applications.

Part 2, “Scenarios” on page 107 identifies several potential technical scenarios for coexistence via point-to-point integration between applications deployed in the IBM WebSphere Application Server and applications deployed in the Microsoft® .NET Framework. This part provides in-depth technical details on how to implement certain scenarios using today’s existing technologies. The implementations for stateless asynchronous, stateful and stateless synchronous presentation (Web) integration include technologies such as Web Services, messaging middleware, native interfaces, etc.

Part 3, “Guidelines” on page 427 provides general guidelines for solution developers. A list of supporting technologies can help with the solution implementation. Chapter 11, “Quality of service considerations” on page 471 is a collection of services available on both platforms.

The “Appendixes” on page 511 go further by showing other IBM technologies and describing two integration solutions between Lotus® Domino™ and .NET applications.

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, Raleigh Center.



The team who wrote the book (left to right): Ajit Mungale, Jeremy Bauer, Paula Dantas, Kieran Scott, Darren Bassett, John Catlin, James Tussing, Peter Kovari

Peter Kovari is a WebSphere Specialist at the International Technical Support Organization, Raleigh Center. He writes extensively about all areas of WebSphere. His areas of expertise include e-business, e-commerce, security, Internet technologies and mobile computing. Before joining the ITSO, he worked as an IT Specialist for IBM in Hungary.

Victoria Amor is an IT Specialist in IBM WebSphere and Lotus Domino at IBM Spain. Her areas of expertise include WebSphere support, particularly the areas of security and administration, and design and consultancy in Lotus Domino. She has worked within IBM for six years, participating in remarkable e-business projects such as the Sydney Olympic Games where she was responsible for all Lotus Domino Servers in the Games System Center. Currently, she is working for ITS department in Services Delivery. She has previously co-authored the *IBM WebSphere V4.0 Advanced Edition: Security* redbook, SG24-6520.

Jason Anderson is a Senior Software Design Engineer in IBM's Applied Web Services Design Center, focusing on distributed technologies and the next generation of IBM products. Prior to working at IBM, Jason was the Chief Technology Officer of Agital Software Corporation where he ran the Research and Development organization and created the company's flagship product The

XML Network Server. Before Agital, Jason worked in a variety of product teams at Microsoft, including Internet Explorer, MS's Java™ Virtual Machine, the Common Language Runtime, and most recently as an Architect for Microsoft's XML Services group.

Darren Bassett is an Advisory IT Specialist working for the Infrastructure and Systems Management group of IBM Global Services in the United Kingdom. Darren has five years of experience with IBM, specializing in Java, Java 2 Enterprise Edition and WebSphere Application Server. He has worked on a number of high-profile customer projects within the United Kingdom, delivering software development and infrastructure solutions for the WebSphere Application Server platform. Darren holds a first class bachelors degree in Computer Science from Kingston University.

Jeremy Bauer is a Staff Software Engineer at IBM in Rochester, Minnesota. In his seven years with IBM, he has worked on IBM DB2® database middleware and end user applications for the iSeries IBM @server. This includes development roles in the IBM iSeries Access for Windows® and IBM iSeries Access for Web products. His most recent work includes development of an ADO.NET driver for the iSeries server. Jeremy recently completed his Masters of Software Engineering at the University of Minnesota and is considering pursuing a doctorate in Computer Science.

John Catlin is an Application Architect for the IBM Software Group Services in the UK.

Paula Dantas is an IT Specialist in IBM Brazil working in the Software Group. She has been working with WebSphere products for two years and her main responsibility is to provide technical sales support to customers using WebSphere Application Server and WebSphere Portal Server, which includes delivery of Proof-of-Concepts, Proof-of-Technologies, Pre-sales Presentations, Demonstrations, Software installation and configuration, Skills transfer to Business Partners, Customers, SWG sales force, etc. Before working with WebSphere, she was responsible for developing Lotus Notes® applications for the Latin America SWG team. She works in Rio de Janeiro, Brazil.

Han Wen Kam is an Advisory IT Specialist for IBM Open Computing Centre, based in Singapore. He has experience in the design and development of Web Services and J2EE applications with core software product skills in WebSphere Application Server and WebSphere Studio. Working with business partners, Han has also spent time developing and conducting technical courses for developers in ASEAN. Han has contributed technical articles to WebSphere Developer Domain and was also the recipient of the "Asia Pacific WebSphere Field Technical Sales Specialist of the Year 2002" award at the Annual Asia Pacific Software e-business University held in Shanghai.

Ajit Mungale has a total of seven years of experience and has been working with IBM GSI as a Senior Software Engineer for the last four years. He has extensive experience with Microsoft technologies and has worked with almost all languages and technologies. He also has experience with IBM products, including IBM WebSphere and MQ.

Kieran Scott is a Solution Specialist working for the IBM WebSphere Platform Solution Test, part of System House, which focuses on improving ease of integration and interoperability of WebSphere Platform components. He specializes in the area of Web Services and makes use of his expertise in J2EE and WebSphere Studio to test functionality and performance of Web Services as part of wider cross-platform scenarios. In this role, Kieran works as a developer, tester, designer and author, and also has the opportunity to work with customers in an advisory capacity. His work includes the use of many different IBM products, and also involves interoperability testing between IBM and Microsoft technologies, such as the .NET Framework.

James Tussing is the President and Principle Architect of Axiom, Ltd., an IBM Business Partner and consulting services firm, specializing in EAI and Network Security and headquartered near Columbus, Ohio, USA. James is an IBM Certified Specialist, Developer and Solutions Expert for IBM WebSphere MQ. He specializes in EAI architecture, multi-platform middleware product development and integrating Microsoft .NET with the enterprise.

Thanks to the following remote participants for their contributions to this project:

Lawrence Lourduraj

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

Cecilia Bardy
Gail Christensen
Mark Endrei
Geert Van de Putte
Carla Sadtler
Margaret Ticknor
Jeanne Tucker

International Technical Support Organization, Raleigh Center

Denise Gabardo
Massimiliano Parlione
Carlo Randone
Rajasi Saha
Andre Tost

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our 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:

IBM® Corporation, International Technical Support Organization
Dept. HZ8 Building 662
P.O. Box 12195
Research Triangle Park, NC 27709-2195



Part 1

Introduction



J2EE introduction

The purpose of this chapter is to provide an introduction to and overview of the Java 2 Platform, Enterprise Edition (J2EE) and the IBM technologies that implement it. The intended audience for this chapter consists of readers who are familiar with the Microsoft .NET Framework and who wish to gain a better understanding of J2EE for comparison purposes.

This chapter discusses the following items:

- ▶ The J2EE architecture, including application components, packaging and deployment, the runtime environment, and the standard services provided by the J2EE platform.
- ▶ Design considerations for J2EE applications, including a discussion of the IBM Rational® XDE™ design tooling.
- ▶ Development tools for J2EE applications, including the IBM WebSphere Studio Application Developer V5.1 integrated development environment and other open source tools for automation of the build and deployment process.
- ▶ Testing tools available for J2EE applications.
- ▶ How J2EE applications are deployed, and the tools available to aid the process.

- ▶ IBM WebSphere Application Server V5, examining the packages available and the runtime architecture of IBM's J2EE compliant application server.
- ▶ Finally, this chapter discusses the administration options available for IBM WebSphere Application Server V5

1.1 Architecture

This section presents a discussion of the architecture of the Java 2 Enterprise Edition platform and the distributed applications that execute upon it. The discussion focuses on the following items:

- ▶ Java 2 Platform, Enterprise Edition architecture and logical application layers.
- ▶ J2EE support for standards, platforms and programming languages.
- ▶ J2EE application components and how they are packaged and deployed.
- ▶ The J2EE runtime environment and life cycle management.
- ▶ Standard services provided by J2EE application servers, including quality of service considerations.

1.1.1 Overall architecture

The J2EE specification defines a standard application programming model and platform for distributed applications in the J2EE environment. The architecture for the J2EE platform is illustrated in Figure 1-1.

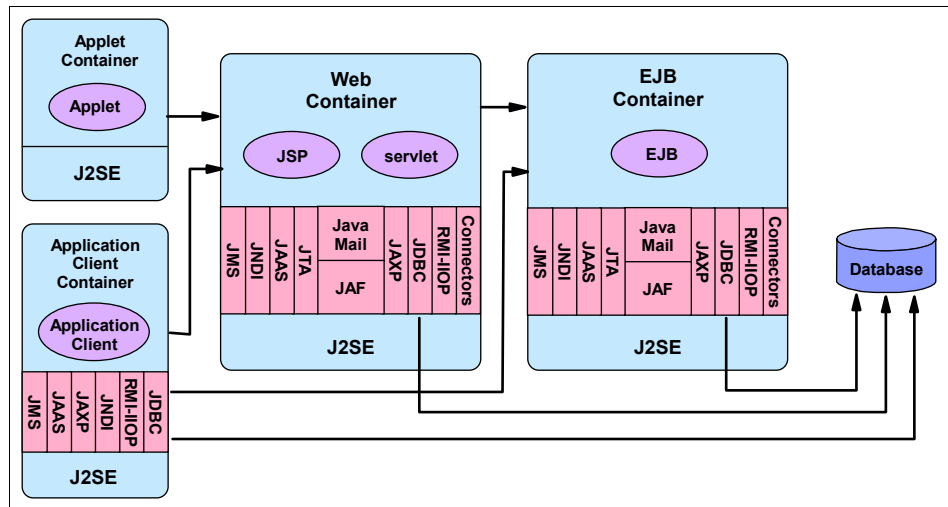


Figure 1-1 J2EE components, containers and services

The J2EE programming model defines four types of application components:

- ▶ Application clients

These components execute in a container in a client process that is distinct from the server processes, though not necessarily remote. They provide a rich set of functionality and for the remainder of this book are referred to as fat

clients. They have full access to the J2EE server side components and services.

- ▶ Applets

These are lightweight client components that execute in the container with restricted access to system resources. Typically, the container executes in a Web browser or handheld device.

- ▶ Web components

Web components comprise Java Server Pages and Java servlets. They execute in the Web container of a J2EE application server. These Web components are typically concerned with presentation and control logic in a distributed J2EE application.

- ▶ Enterprise Java Beans

Enterprise Java Beans (EJB) are server side components that execute in the EJB container of an application server. Their purpose is to implement business logic and model business data.

The containers themselves perform functions on behalf of, and provide services to the applications that execute within them. For a more detailed overview of the containers please refer to “Containers” on page 12.

1.1.2 Layer technologies (application architecture)

Throughout this book, a five-layer model is used to represent the logical layers of a distributed enterprise application. The mapping of J2EE application components, containers and other artifacts onto the five-layer model are illustrated in Figure 1-2 on page 7.

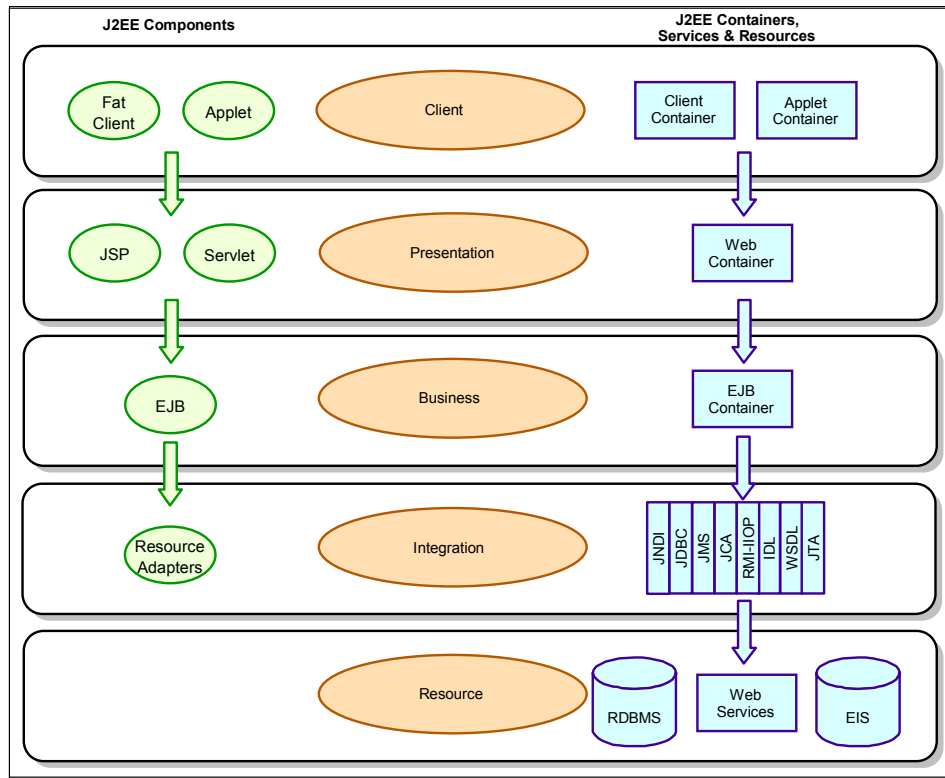


Figure 1-2 Logical Application layers

A layer can be considered as a logical grouping of components with common functionality; the function of each layer is briefly described below:

► Client

The Client layer contains client applications, these reside in a client process that can be remote from or local to the server processes. When considering this in the context of J2EE, this is the layer where fat clients and Java Applets reside.

► Presentation

The Presentation layer is where processing of presentation and control logic occurs. In the Model-View-Controller paradigm, this layer contains the view and control elements. For a J2EE application, this layer contains JSPs and servlets executing in a Web container.

► Business

The applications business logic executes in the business layer. The typical J2EE components that execute in this layer are Enterprise Java Beans and

associated helper classes. In the Model-View-Controller paradigm, this layer contains components that model the business and the data.

► Integration

The integration layer is where J2EE services and other integration middleware such as message queueing software and enterprise information system connectors reside.

Note: Our diagram illustrates the J2EE services in the logical hierarchy as entities separate from the containers. In the physical J2EE platform, the services are actually provided by the client, Web and EJB containers.

► Resource

The Resource layer represents any resource to which a distributed application may connect. For instance, this could be a relational database or an enterprise information system.

1.1.3 Standard support

Java 2 Platform, Enterprise Edition is a specification that defines a standard application programming model and runtime environment for the creation and execution of distributed enterprise applications. In terms of its relationship with other Java 2 platforms, Java 2 Enterprise Edition can be considered to be a superset of the Java 2 Platform Standard Edition (J2SE).

While Sun Microsystems invented and retains control over the Java language itself, the Java 2 Enterprise Edition is a result of the contributions of a number of enterprise software vendors including IBM.

Control over the evolution of the Java 2 Enterprise Edition and other Java technologies is retained by the Java Community Process (JCP). This is an open organization whose members' prime responsibility is to guide the development and approval of Java technical specifications. The Java Community Process also considers submissions for enhancements to the Java technologies from non-members. This collaboration of interested parties has been particularly successful in ensuring that Java technology has remained open.

The specification incorporates a compatibility test suite. Vendors can certify their J2EE implementations against this test suite.

1.1.4 Platform support

A feature of all Java applications is their independence from the underlying operating system on which they execute. This platform independence is

achieved by executing the code in a virtual machine, known as the Java Virtual Machine (JVM), which abstracts the code from the operating system.

Provided that a Java Virtual Machine exists for a given platform, then Java 2 Enterprise Edition applications can run on it.

The Java runtime is available on numerous platforms.

WebSphere is available in the following platforms:

- ▶ AIX® 4.3.3 4330-10 Maintenance level, 5.1 5100-02 or 5100-03 Maintenance level, 5.2
- ▶ Windows 2000 Advanced Server Service Pack 3, 2000 Server Service Pack 3, 2003 Server, Enterprise, 2003 Server, Standard, NT 4.0 Service Pack 6.0a
- ▶ Red Hat Linux 8, Enterprise Linux WS/ES/AS for Intel® 2
- ▶ SuSE Linux 7.3 for Intel, SLES 7 2.4
- ▶ UnitedLinux 1.0
- ▶ OS/400® 5.1, 5.2
- ▶ Red Hat Linux 7.2 for s/390
- ▶ SuSE SLES 7 for zSeries®
- ▶ HP-UX 11iv1
- ▶ Solaris 8, 9

1.1.5 Programming languages

Language support in Java 2 Enterprise Edition is limited to Java.

Java applications run within the Java Runtime Environment (JRE) and obtain services from that runtime, such as language independence and garbage collection. Java code is managed by the JRE. In order for a Java program to call anything that is not managed by the JRE, for instance a C-style function within a Dynamic Link Library (DLL) on Windows, it is necessary to leave the managed code environment during the call into the DLL.

The Java Native Interface (JNI), provided as part of the Java 2 Standard Edition, enables interaction with applications written in native code, but this is achieved at the expense of code portability.

The Java Native Interface allows you to call unmanaged, native code from within the Java Runtime Environment and vice-versa.

1.1.6 Deployment units

The Java 2 Enterprise Edition defines a standard model for packaging and deploying enterprise applications, as shown in Figure 1-3 on page 10.

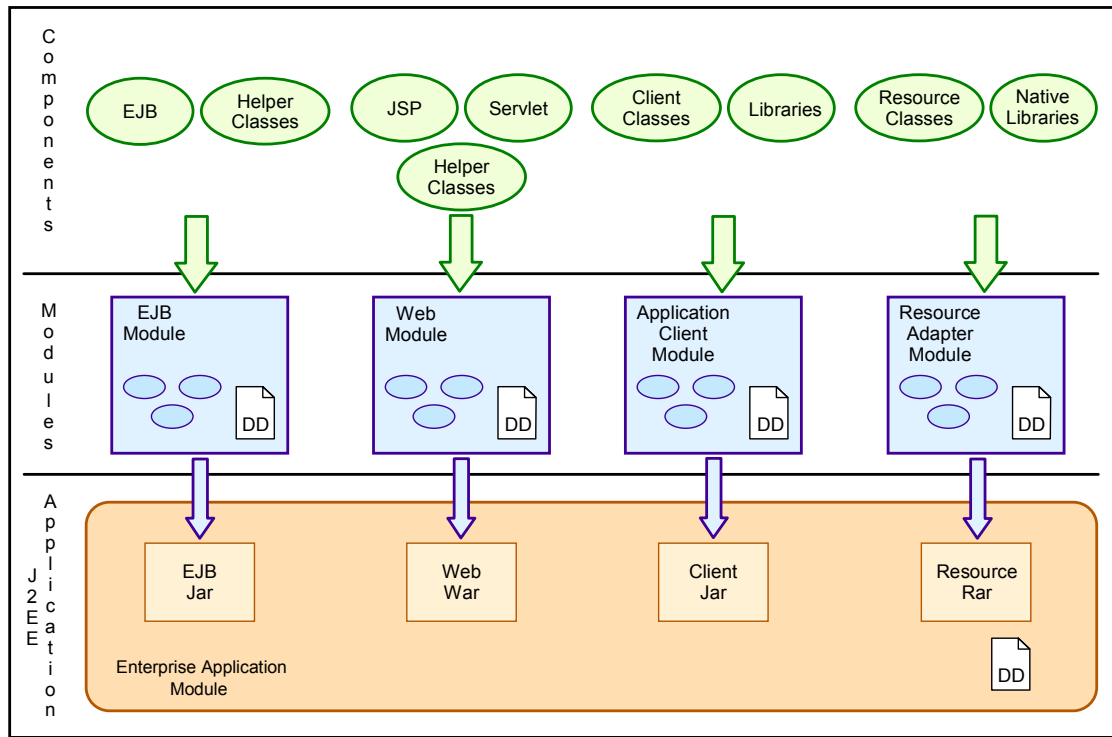


Figure 1-3 J2EE Packaging model

Application components are assembled into modules, which may then be deployed into a J2EE application server. If so desired, the modules may also be packaged into an enterprise application module, allowing the entire application to be deployed as a single unit. Each module also includes a deployment descriptor; the purpose of this XML file is to describe to the runtime container how the module should be deployed.

The contents of each module are as follows:

- Enterprise Java Bean Module

This module contains the deployed Enterprise Java Beans and the client side stubs required for remote method invocation. It may also contain additional Java helper classes which perform actions on behalf of the EJBs. Also included is an EJB deployment descriptor. The EJB module is deployed as a Java archive (.jar) file.

- ▶ **Web Module**

The Web module contains the Web application; typically, this will include Java Server Pages, Java Servlets, utility classes and any external libraries required by the Web application. This module may also include static Web content such as HTML pages and images. The module also includes the Web deployment descriptor, and is assembled as a Web archive (.war) file.

- ▶ **Application Client Module**

Client side applications are fat clients that interact with J2EE server applications. They are packaged as application client modules using the Java archive (.jar) file format. An application client actually executes in a client container, so in addition to the classes required to implement the client, the module also includes the libraries required by the client container runtime and the application client deployment descriptor.

- ▶ **Resource Adapter Module**

A resource adapter is a system-level software driver that enables a J2EE application to connect to Enterprise Information Systems (EIS). The resource adapter module contains the Java classes and native libraries required to implement a Java Connector Architecture (JCA) resource adapter to an EIS. The module is assembled as a Resource adapter archive (.rar) file. As with all other module types, the resource adapter module also incorporates a deployment descriptor.

Configuration

The WebSphere Application Server V5 configuration is stored in XML formatted configuration files. These files are stored in a directory structure that is mapped to the server deployment structure (cell/node/server).

For both single server and servers in a cell, the configuration files are stored in files. In a cell environment, the Deployment Manager takes care of the synchronization of the files between the nodes.

It is not recommended that you alter the files to make configuration changes for WebSphere. Use one of the administration tools which WebSphere provides, for example: Administrative Console or wsadmin.

1.1.7 Runtime execution environment

The Java runtime environment is a virtual machine, known as the Java Virtual Machine (JVM). This execution engine acts as an abstraction layer between the host operating system and the Java application, enabling platform independence. Like any physical machine, the virtual machine has an instruction

set and provides services to the Java application such as memory management and input and output operations.

Java programs are compiled into machine independent byte code. The Java Virtual Machine interprets this byte code at runtime. Optimizations are also provided, in the latest JVMs, by the HotSpot compiler. This works by identifying frequently executed code segments and then repeatedly optimizing them for execution. Operating system specific optimizations also exist, allowing the Java Virtual Machine to scale.

Containers

The Java 2 Enterprise Edition application server facilitates development of distributed applications by providing a number of containers that provide services to the applications deployed within them. The type of services provided to the application by the container include security, transactions, object pooling and naming services. The availability of such services enables the application developer to focus on implementing business and presentation logic.

The following containers are defined by the J2EE specification:

► Web container

In addition to managing the Web applications and components that are deployed within it, the Web container is responsible for providing the following services to Web applications:

- On startup of the application server, the container establishes a transport mechanism between the Web server and the Web container. This communication channel is used by the container to receive requests and send responses on behalf of the Web applications running within it.
- The container manages a configurable pool of worker threads for the processing of servlet and Java Server Page requests.
- Session management is also provided by the Web container. It is responsible for creating and invalidating sessions, providing a session tracking mechanism and writing and reading session data on behalf of the Web application.
- The Web container also provides a virtual hosting mechanism, allowing a single application server to masquerade as multiple Web sites.

► EJB container

The EJB container is responsible for managing the life cycle of the Enterprise Java Beans deployed within it and providing threading and transaction support to them. The container also provides a caching mechanism to optimize access to EJBs.

- Client container

The client container is installed separately on the client and manages the runtime environment for fat client applications. J2EE application client containers provide standard services to the client application, including remote object discovery and remote method invocation, thus allowing the application client to access business logic on server side components such as EJBs, and also providing access to other resources.

- Java Connector Architecture (JCA) container

The JCA container provides a pluggable and configurable bridging mechanism for J2EE applications to access Enterprise Information Systems. IBM WebSphere Application Server V5 implements this bridging mechanism as a separate container within the application server.

1.1.8 Life cycle management

In Java 2 Enterprise Edition, there are two levels of object life cycle management. The Java Virtual Machine provides life cycle management of Java objects, while life cycle management of J2EE application components is a function of the container; see “Containers” on page 12.

In Java, it is the responsibility of the Java Virtual Machine to dynamically load, link and initialize classes and interfaces. The task of loading a class is delegated either to the bootstrap classloader of the JVM or a user defined classloader. To load a class or interface, the classloader will first attempt to locate a binary representation of the class or interface by traversing the classpath and then create an object instance from the binary representation. Linking is the process of taking the class or interface and combining it into the runtime state of the Java virtual machine so that it can be executed. Upon successful instantiation, the class or interface is finally initialized by executing its `init()` method.

During the life cycle of an object, the JVM maintains a table of references to that object.

Object destruction is also the responsibility of the Java Virtual Machine. This is achieved by means of a garbage collection process that runs periodically to clean up any dereferenced objects. The algorithm used in garbage collection is dependent upon the implementation of the JVM. Generally, though, garbage collection is implemented as a mark and sweep algorithm. In this algorithm, the garbage collection process starts by locating all dereferenced objects in the memory heap and marks them for deletion. In the second phase, the garbage collector removes, or sweeps, the marked objects from the memory heap. JVM technology has improved significantly, resulting in improved performance of the garbage collection process. These enhancements include optimizations such as

memory heap defragmentation, short and long lived object heaps and multi-threaded garbage collection.

The life cycle management of J2EE components such as servlets and EJBs is the responsibility of the container in which they reside.

Object pooling

Object pooling is a mechanism employed in many object-oriented languages that enables the reuse of objects. The advantages of using a pooling mechanism is that it avoids the overhead of continuous object creation and destruction, and improves memory usage: fewer objects have to be created since they are shared amongst clients.

Java 2 Enterprise Edition application servers incorporate object pooling mechanisms to improve application performance. In WebSphere Application Server, object pools such as JDBC connection pools are created by the application server and made available as resources to J2EE applications. It is the responsibility of the application server to manage the object pool.

1.1.9 Remote object discovery

Object location in Java 2 Enterprise Edition is achieved via a directory structure known as the CosNaming namespace, in which the application may store references to objects. Objects are referenced in the namespace by being bound to a unique name. To locate a particular object, the application performs a lookup on the binding name.

This mechanism provides transparent discovery of objects, which if running in a clustered environment may not be located within the same virtual machine or even on the same physical machine. In WebSphere Application Server V5, each managed server process maintains its own unique view of the namespace; this is known as the local namespace. Typically, for an application server, the namespace will contain references for the applications and resources configured in that server. The local namespaces are linked to form a global namespace for the entire cell. Thus, any object within the global namespace can be located from any managed server process.

An application programmer interface called the Java Naming and Directory Interface (JNDI) is provided to allow programmatic access to the namespace.

1.1.10 Remote Method Invocation

The Java 2 Standard Edition provides a mechanism for calling methods on remote objects, called Remote Method Invocation. This technology is leveraged

in Java 2 Enterprise Edition to provide a distributed computing business tier through the Enterprise Java Beans (EJB) application programming interface.

Remote Method Invocation uses a common mechanism for invoking methods on a remote object, stubs and skeletons. The stub is located on the client or local system and acts as a proxy to the remote object. The client makes a method call on the stub which then handles the complexities of the remote method call. This mechanism makes the underlying communications transparent to the client. As far as the client is concerned, the method call appears local.

When a method on the stub is invoked, the following actions occur.

1. The stub establishes a connection to the Java Virtual Machine where the remote object resides.
2. Method parameters are written and transmitted to the remote Java Virtual Machine. This process is more frequently known as marshalling.
3. The stub then waits for the return value from the method invocation.
4. When received the return value is read or unmarshalled.

The underlying communication protocol for Remote Method Invocation is the Internet Inter-ORB Protocol (IIOP). The use of the IIOP protocol in J2EE facilitates legacy application and platform integration by allowing objects written in other CORBA-enabled languages to communicate with Java-based applications.

1.1.11 Web Services

Web Services standards are still evolving within J2EE and are currently being integrated into the specification. However, the importance of Web Services as an integration technology has not been overlooked by J2EE vendors and many, including IBM, provide varying levels of support for the standards.

IBM WebSphere Application Server V5 and its associated development tool, IBM WebSphere Studio Application Developer, provide support for the following Web Services standards and concepts:

- ▶ Simple Object Access Protocol (SOAP).
- ▶ Web Services Description Language (WSDL).
- ▶ Universal Description. Discovery and Integration (UDDI).
- ▶ Web Services Invocation Framework (WSIF).
- ▶ Web Services Inspection Language.
- ▶ Web Services security.
- ▶ Workflows and business processes.

- ▶ Web Services gateway.
- ▶ Java API for XML-based Remote Procedure Calls (JAX-RPC)
- ▶ JSR-109, which standardizes how Web Services are deployed into a J2EE container.

For a full discussion of Web Services and IBM tooling support, refer to the IBM Redbook *WebSphere Version 5 Web Services Handbook*, SG24-6891.

1.1.12 Transaction management

The J2EE specification states that product vendors must transparently support transactions that involve multiple components and transactional resources within a single J2EE product. This requirement must be met irrespective of whether the product is implemented as a single process, multiple processes on a single network node or multiple processes on multiple network nodes.

The following J2EE components are considered transactional resources and must, therefore, support transactions:

- ▶ Java Database Connectivity (JDBC) database connections.
- ▶ Java Messaging Service (JMS) sessions.
- ▶ Connectors for resource adapters specifying XA compliant transactions.

The specification, however, does not specify any particular protocol for supporting transaction interoperability across multiple J2EE products.

Transaction support must be provided for applications comprising combinations of Web components accessing multiple Enterprise Java Beans within a single transaction. Furthermore, support must be provided for each component accessing one or more connections to access one or more transactional resources.

Transactions may not, however, span a request from a client. This implies that a transaction initiated in a Web component must complete before the response is returned to the client.

The key point about the specification requirements is that transactions must be managed by the container, and therefore J2EE application developers need only concern themselves with whether their business process requires transactional support or not.

1.1.13 Security

Almost every enterprise has security requirements and specific mechanisms and infrastructure to meet them. While the implementation and levels of service provided by enterprise security systems may vary, they all address the following considerations to some extent.

- ▶ Authentication

This mechanism addresses how entities that are attempting to communicate prove to one another that they are who they say they are.

- ▶ Access control for resources

This is the process of ensuring that access to protected application resources is restricted to only those users or groups of users who have authority to access them.

- ▶ Data integrity

Data integrity considerations address how to validate that data passed between two entities has not been modified in some way by a third party while in transit.

- ▶ Confidentiality or data privacy

Mechanisms relating to confidentiality or data privacy deal with how to ensure that data can only be read by those users who are authorized to do so.

- ▶ Non-repudiation

Non-repudiation is a way of providing absolute proof that a particular user performed some action.

- ▶ Auditing

Auditing is the process of creating a tamper proof trail of security-related events in order to evaluate security policies and mechanisms, and when used in conjunction with non-repudiation, to provide evidence of malicious user actions.

The J2EE specification defines a number of goals for security within J2EE applications:

- ▶ Portability

The J2EE security model must support the concept of portability. In other words, it must be implemented in such a manner that J2EE applications are decoupled from the underlying security implementation.

- ▶ Transparency

J2EE application developers wishing to implement security in their components should not need to understand security in order to do so.

However, in practice, a developer should at least have an understanding of the security considerations addressed above.

- ▶ Isolation

This is related to the portability requirement. What the specification says here is that authentication and access control should be performed in accordance with instructions provided in the deployment descriptors, and managed by the systems administrator. This ensures that the application is decoupled from the underlying security implementation.

- ▶ Extensibility

The J2EE specification provides security application programmer interfaces. Provided that the application restricts itself to using these APIs for implementing security in its components, it will retain independence from the underlying platform, and thus retain portability.

- ▶ Flexibility

The application should not impose a specific security policy, rather it should facilitate the implementation of security policies.

- ▶ Abstraction

The mapping of security roles and access requirements to environment specific security roles, users and policies should be specified in the applications deployment descriptors. The deployment descriptors should also document which security properties can be modified by the deployer.

- ▶ Independence

Required security behaviors and deployment contracts should be implementable using a variety of popular security technologies.

- ▶ Compatibility testing

The J2EE security model should be described in such a way that an implementation can readily be certified as compatible or not.

- ▶ Secure interoperability

Application components executing in a J2EE product must be able to invoke services provided in a different J2EE product, irrespective of whether the same security policy is used.

Figure 1-4 on page 19 illustrates how WebSphere Application Server leverages the security provided by the operating system and other Java and J2EE components.

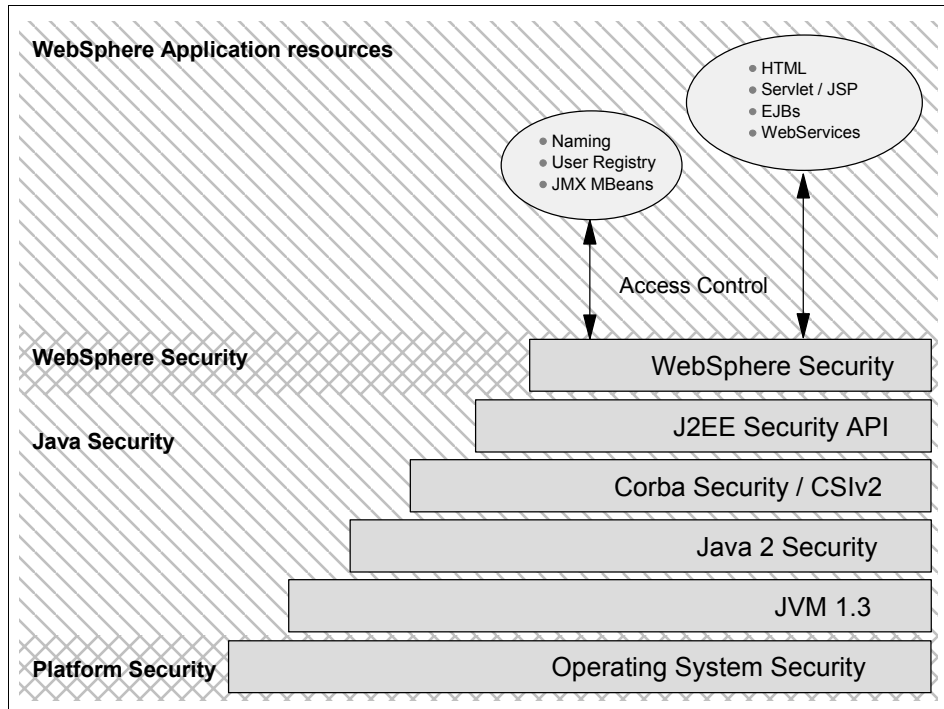


Figure 1-4 WebSphere environment security layers

Due to the layered nature of the WebSphere Application Server security model, the security considerations described earlier in this section are automatically addressed by services provided in the operating system and the J2EE security model.

When developing a J2EE application for WebSphere Application Server, it is important that decisions relating to security be made early in the application development life cycle, preferably during the design phase.

In J2EE applications, security is configured through the deployment descriptors, and it is the responsibility of the container to enforce the security policy.

WebSphere Application Server provides implementations of user authentication and authorization mechanisms providing support for various user registries:

- ▶ Local operating system User registry
- ▶ LDAP User registry
- ▶ Custom User registry

Authentication mechanisms supported by WebSphere are:

- ▶ Simple WebSphere Authentication Mechanism (SWAM)
- ▶ Lightweight Third Party Authentication (LTPA)

For a more detailed discussion of security in IBM WebSphere Application Server V5, refer to the IBM Redbook *IBM WebSphere V5.0 Security*, SG24-6573.

1.1.14 Load balancing and failover

The specification does not make any comment with regards to the partitioning of services or functions between machines, servers or processes. The only concern is that the implementation meet the specification.

In order to provide a resilient, scalable platform, J2EE vendors provide products which can be distributed and load balanced across several machines. IBM WebSphere Application Server V5 provides the following features:

- ▶ Scalability
- ▶ Load balancing
- ▶ Availability
- ▶ Maintainability

This is implemented by allowing the replication and clustering of homogenous application servers so that applications can be horizontally and vertically scaled and the workload balanced across them.

For a more detailed discussion on load balancing and failover for IBM WebSphere Application Server, refer to the IBM Redbook *IBM WebSphere V5.0 Performance, Scalability and High Availability*, SG24-6198.

1.1.15 Application logging

The J2EE specification makes no reference to how logging should be implemented by product providers. Consequently, vendors implement logging independently of one another.

IBM WebSphere Application Server provides several general purpose logs:

- ▶ Java Virtual Machine logs

These logs are created by redirecting the output streams of the JVM, stdout and stderr.

- ▶ Native logs

These logs are created by redirecting the output streams of the JVM process and any native modules that it references.

- ▶ Trace log

This log can be enabled to trace internal components of the application server. It is useful in problem determination when information recorded in the other logs is insufficient.

- ▶ Service log

This log records all WebSphere system events.

How logging is implemented in a J2EE application is left to the discretion of the application developers. However, there are logging APIs available for purchase from a number of vendors. One of the most widely used APIs for logging is the *log4j* API available from the Apache Software Foundation at the following URL:

<http://jakarta.apache.org/log4j>

1.2 Development

It is possible to develop a Java application using a text editor, but developing Web applications requires more than just writing Java code. That is where tooling comes into play. IBM WebSphere Studio Application Developer is an integrated development environment for building, testing, and deploying J2EE and Web Service applications.

A programming model should describe the entire model for developing, deploying and maintaining applications in an information system. J2EE specification breaks the life cycle of an application into six different roles or responsibilities making the development process easier.

J2EE product provider

The product provider is the company that designs and makes available for purchasing the J2EE platform, APIs and other features defined in the J2EE specification. Any vendor that implements the J2EE platform according to the Java 2 Platform, Enterprise Edition Specification is considered in this role. For example, IBM is a J2EE product provider because it implements J2EE in WebSphere products.

Tool provider

This is the person or company which implements tools to enable other tasks on the J2EE platform to be development, assembly, packaging, deploying and monitoring tools. IBM is a tool provider with products like WebSphere Studio Application Developer, Tivoli® Tools, and so on.

Application component provider

The application component provider is responsible for developing components of the application and the deployment descriptors. The application component provider develops the Web components, enterprise beans, applets, or application clients for use in J2EE applications.

Application assembler

The application assembler is responsible for combining components from application component providers into a J2EE application EAR file. The assembler or deployer can edit the deployment descriptor directly or use tools that correctly add XML tags according to interactive selections. The following tasks are performed to deliver an EAR file that contains the J2EE application:

- ▶ Assembles EJB JAR and Web components (WAR) files created in the previous phases into a J2EE application EAR file.
- ▶ Specifies the deployment descriptor for the J2EE application.
- ▶ Verifies that the contents of the EAR file are well formed and comply with the J2EE specification.

Application deployer

This is the company or person that installs and configures the J2EE application into the Runtime Environment (into the J2EE server). During the configuration, the deployer follows instructions supplied by the application component provider to resolve external dependencies, specify security settings, and assign transactions attributes.

System administrator

The system administrator is responsible for administering the computing and networking infrastructure where J2EE applications run and oversees the runtime environment.

As you can see, the output for one role is the input to the next one.

1.2.1 Writing a Java application using a text editor

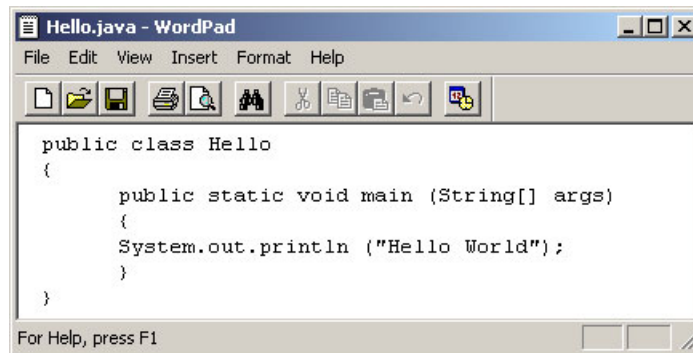
In this section, we are going to demonstrate how to develop a Hello World application in Java using a text editor, then how to compile and execute it. In this example, we are going to use Notepad, but you can use any editor program, like the DOS **edit** command, or in UNIX, **vi** or **emacs**.

Note: You will need the JDK (Java Developer's Kit) installed in your computer before you start. JDK is a set of tools including the compiler (javac), the interpreter (java) and other tools used by Java developers.

Java programs usually have different phases to be executed:

- ▶ Edit
- ▶ Compile
- ▶ Execute

1. In the text editor, write the code below and save the file as Hello.java.

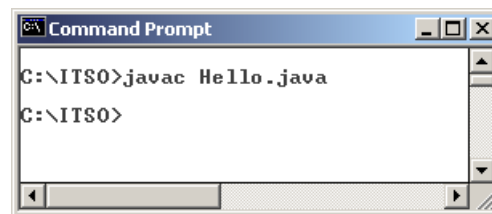


```
public class Hello
{
    public static void main (String[] args)
    {
        System.out.println ("Hello World");
    }
}
```

Figure 1-5 Hello World java code

2. To compile the java program, open a DOS command prompt window and type: `javac Hello.java` at the location of the source.

The **javac** command compiles the program and translates the Java program into bytecodes which is the language understood by the Java interpreter.



```
C:\ITS0>javac Hello.java
C:\ITS0>
```

Figure 1-6 Compiling the code

Unlike in .NET, the compiler generates a bytecode class which then runs in the JVM and is portable between various platforms. The Java compiler does not generate .exe files on the Windows platform.

To see other options for the Java compiler, type `javac`.

Figure 1-7 Java compiler options

3. If the program compiles correctly, a file called `Hello.class` will be created. This is the file that contains the bytecodes.

To execute `Hello.java`, type `java Hello` in the command prompt window.

Figure 1-8 Execution of the program

1.2.2 WebSphere Studio Application Developer (IDE)

IBM's WebSphere Studio Application Developer is a powerful, yet easy-to-use Integrated Development Environment (IDE) for J2EE application development, analogous to Microsoft's Visual Studio .NET, the IDE for developing Microsoft .NET based solutions.

WebSphere Studio Application Developer is written to J2EE specifications (supporting both J2EE 1.2 and 1.3 application development), and provides the tools, editors and wizards for rapid development of all J2EE artifacts, including

Java classes, servlets, HTML files, JSP pages, Enterprise Java Beans (EJBs), Web Services, and XML deployment descriptors, all accessible from a single user interface. It is built on top of the Eclipse framework, open source project (designed by IBM), allowing it to benefit from the platform's extensibility through new wizards and plugins.

There are several products in the WebSphere Studio family:

- ▶ **WebSphere Studio Site Developer**
A robust, easy-to-use development environment for building, testing and maintaining dynamic Web sites and Web Services. Intended for professional developers of dynamic Web applications and sites.
- ▶ **WebSphere Studio Application Developer**
Extends the functionalities of Site Developer to include support for programmers working on business logic, including advanced Web Services and EJBs.
- ▶ **WebSphere Studio Application Developer - Integration Edition**
Contains all functionality of WebSphere Studio Application Developer but extends this for accelerated development and integration of complex applications.
- ▶ **WebSphere Studio Enterprise Developer**
Extends the Integration Edition to bring the power of J2EE and rapid application development to diverse enterprise environments.

We will be focusing on the use of WebSphere Studio Application Developer in this book; Figure 1-9 on page 26 shows the view of the application after the first start.

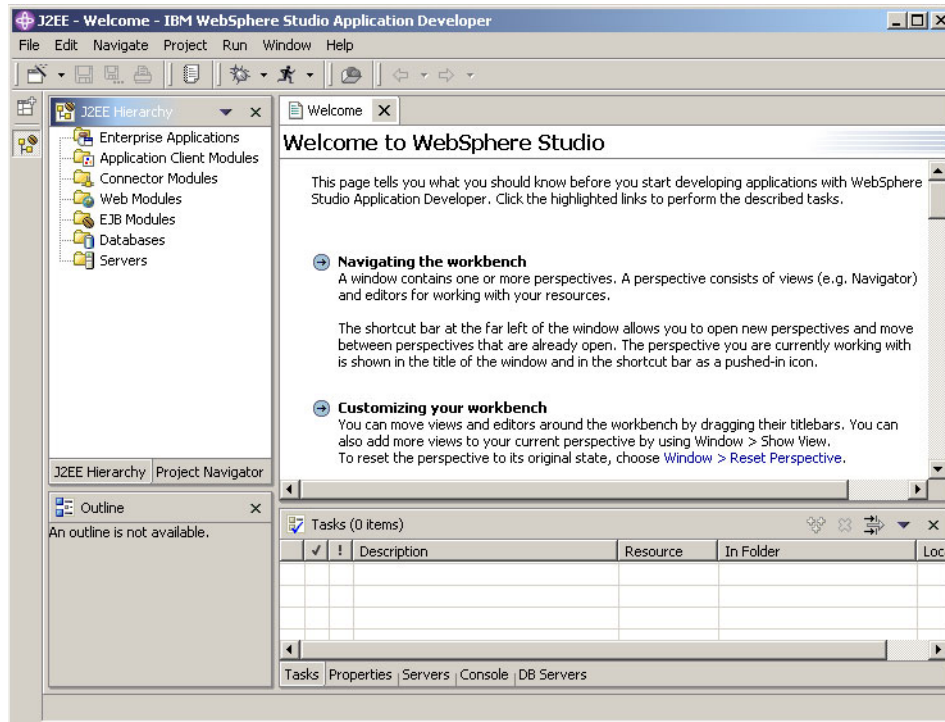


Figure 1-9 WebSphere Studio Application Developer

The development environment of WebSphere Studio Application Developer is similar in look and feel to that of Visual Studio .NET, so users familiar with the Visual Studio .NET environment should find it relatively straightforward to come to grips with the WebSphere Studio IDE.

WebSphere Studio Application Developer includes the following tools for development:

- ▶ Web development
- ▶ Relational database development
- ▶ XML development
- ▶ Java development
- ▶ Web Services development
- ▶ Team collaboration
- ▶ Enterprise Java Bean (EJB) development tools
- ▶ Plug-in development

In WebSphere Studio Application Developer, *perspectives* are a role-based collection of *views* and *editors*, and present just the tools needed for the task at hand. For example, the Java perspective is designed for the role of the Java

developer, and presents only the tools required for tasks associated with Java development. Or, there is the Server perspective, which has the tools for administering and creating server instances and configurations for the test environment. A perspective is made up of several *views* which are separate windows that provide specific ways of viewing and working with resources associated with the roles in that perspective. The tooling also has *editors* which allow you to create and modify the resources (for example: the Java editor, the XML editor, and so on).

WebSphere Studio Application Developer has the following predefined perspectives for development roles:

- ▶ Java perspective
- ▶ Java Browsing perspective
- ▶ Java Type Hierarchy perspective
- ▶ Web perspective
- ▶ J2EE perspective
- ▶ Server perspective
- ▶ XML perspective
- ▶ Plug-in Development perspective
- ▶ Resource perspective
- ▶ Data perspective

Perspectives are highly customizable, allowing the user to add, remove or arrange views and editors as they choose. There is even the option of creating a completely new perspective if none of the pre-defined ones match the needs of the developer.

The workbench loads up into the J2EE perspective by default, which contains views specific to J2EE development. For example, the J2EE Hierarchy provides a hierarchical view of your J2EE resources, and provides an easy way to view the deployment descriptor tree of your J2EE applications and modules, and to quickly navigate to editors for component objects.

When developing a J2EE application with WebSphere Studio Application Developer, the user creates an Enterprise Archive Project (EAR Project). Within the EAR project, multiple J2EE artifacts (as listed above) are organized into Projects. An EAR Project in WebSphere Studio Application Developer is analogous to a Solution in Visual Studio .NET.

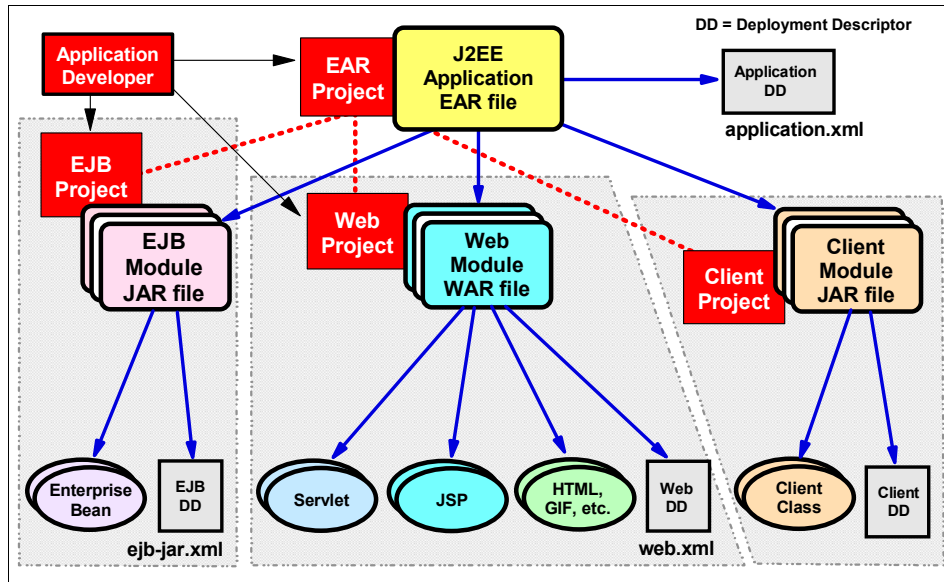


Figure 1-10 Similarities of the WebSphere Studio project hierarchy and the J2EE application structure

The structure of the file system in WebSphere Studio mirrors the structure of a J2EE application as defined in the J2EE specification. At the top level, there is the EAR project (analogous to the J2EE Application EAR file) with its deployment descriptor (application.xml). If the EAR project is expanded in the J2EE Hierarchy view, the J2EE modules contained within that EAR are visible, including the Java Client JAR files, EJB JAR files and Web application WAR files, each with its own deployment descriptors.

In WebSphere Studio, a single folder is used to represent each project, each containing the files and metadata required for deployment of the application to a server. These folders include the directories as defined in the J2EE specifications, such as the META-INF directory for EJBs and WEB-INF for WARs. Within the EJB projects, one can view the EJBs they contain, and likewise for the Web application, WAR projects where one can see the servlets and JSPs contained within the WAR file.

For more details on WebSphere Studio Application Developer and the WebSphere Studio family, refer to *WebSphere Studio Application Developer Version 5 Programming Guide*, SG24-6957.

Source code management

WebSphere Studio has built-in plugins (Eclipse plugins) to support source code management and versioning. The following clients are supported in WebSphere Studio:

- ▶ Rational ClearCase®
- ▶ CVS

Eclipse itself has several other plugins developed to support source code management; some of them are free from open source projects, others are third party applications.

1.3 Testing

This section provides information about testing.

1.3.1 WebSphere Studio Application Developer

The WebSphere Studio Application Developer IDE provides extensive features for debugging and testing J2EE applications, including:

- ▶ Integrated Debugger
- ▶ Server tools for testing and deployment
- ▶ Performance profiling

Integrated debugger

WebSphere Studio Application Developer has an integrated debugging utility allowing the user to detect and diagnose errors in their programs, which can be running either on the local machine or remotely.

The debugger allows the user to control the execution of the program by setting breakpoints in the code, suspending launches, stepping through code, and examining the values of variables.

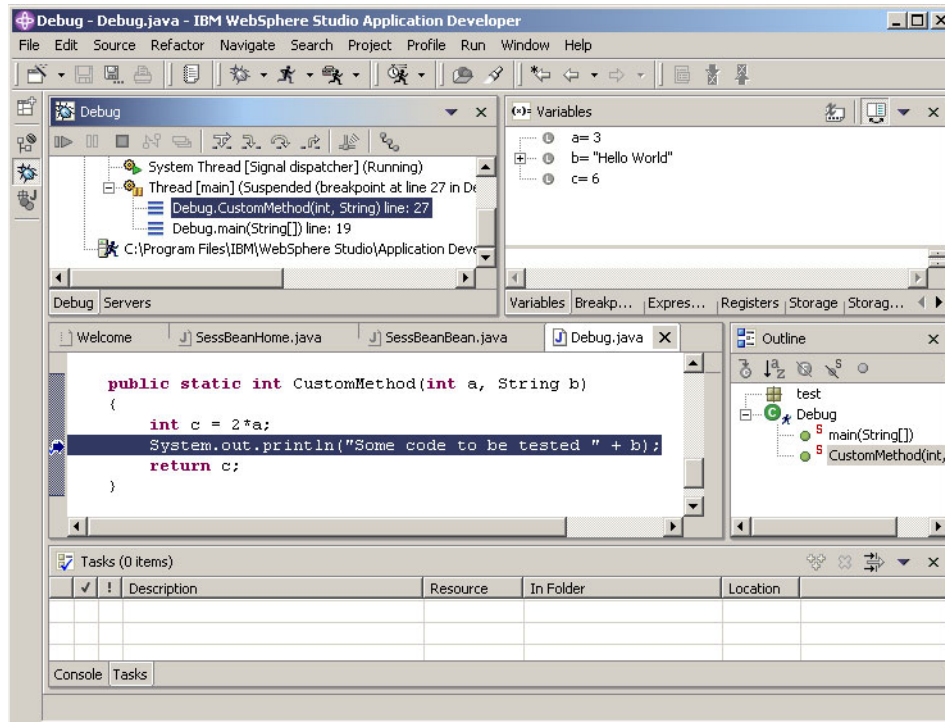


Figure 1-11 The debug perspective

There is a predefined perspective, as shown in Figure 1-11, which is dedicated to the role of debugging. This perspective contains a view to show the breakpoints in the code, a view for inspecting the values of variables, a process view to show a list of running and terminated processes, and a debug view to show threads of execution and stack frames.

Server tools for testing and deployment

The server tools include the WebSphere Unit Test Environment, which is a lightweight test environment identical in functionality to a production server and supports testing of applications on local and remote servers. It offers a runtime identical to that of the full WebSphere Application Server, but without the added complexity of deployment one would encounter in a full production server. The test server has the ability to load projects and class files directly from the WebSphere Studio workspace directories, meaning there is no need to package and publish the projects to the server.

The user can therefore test solutions in WebSphere Studio Application Developer without having to go through the trouble of deploying to a full server,

and because of the identical runtime functionality, he/she can be confident that if the application works in the Test Environment then it will also work on the real server. WebSphere Studio Application Developer also has the ability to deploy to and test applications on a remote WebSphere server.

Within the Test Environment, the tester has the ability to define multiple server configurations, enabling testing of an application on a variety of different server types and in different test environments. This also means that the testing can be extended to previous versions of the server runtime. For example, in WebSphere Studio Application Developer V5.1, there is the ability to test in WebSphere V5.0 and in WebSphere V4.0, ensuring backward compatibility of your applications.

Testing of Web and EJB projects in WebSphere Studio Application Developer is very straightforward. The user simply has to select the project they would like to test and select **Run on Server** from the context menu. If the project is not yet associated with a server instance, WebSphere Studio will automatically create one, start it, and run the project on the server. Testing Web projects will involve an HTML page being displayed in the embedded Web browser, from which the user can test the functionality of the Web project code. Testing an EJB project will start the Universal Test Client.

Universal Test Client

The Universal Test Client is a Web-based application included in WebSphere Studio Application Developer, allowing the user to test functionality of EJBs from a Web browser.

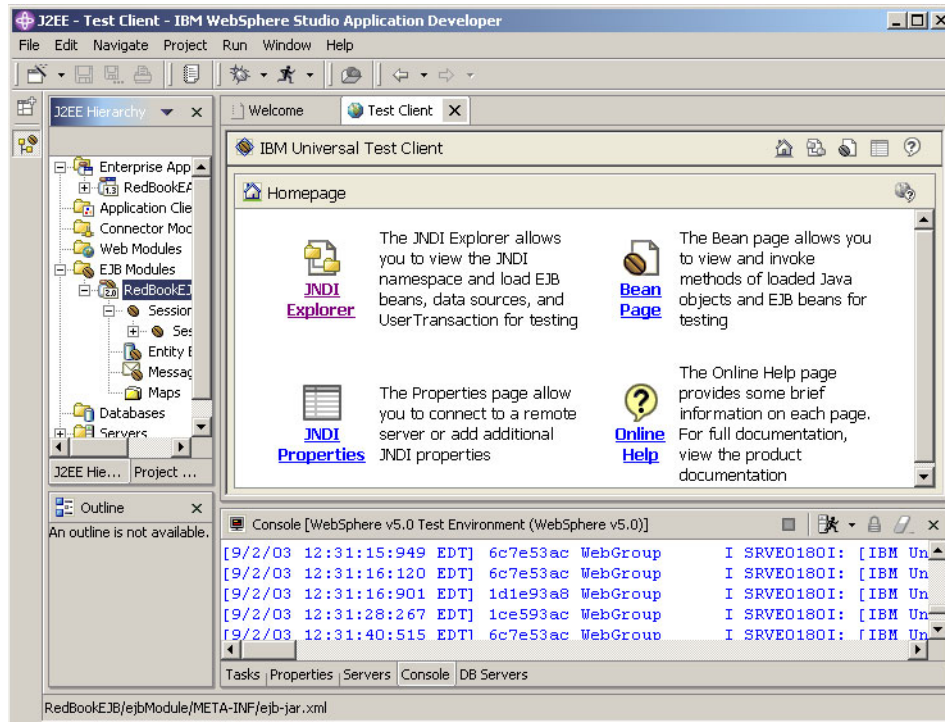


Figure 1-12 The Universal Test Client

The tester can use the Universal Test Client to create() or find() an instance on a bean, which they can then invoke methods on, passing relevant parameters as required, and viewing the results in the browser.

Performance profiling

WebSphere Studio Application Developer has a perspective for Profiling and Logging, to help users identify, isolate and fix performance bottlenecks in their application code, such as:

- ▶ Memory leaks
- ▶ Low throughput
- ▶ Deadlocks
- ▶ System resource constraints (for example: not enough memory)

The profiling tools included in the WebSphere Studio IDE provide the ability to determine problems early in the application development cycle, therefore reducing the possibility of finding serious issues in the final performance testing phase. They also give architects and designers the opportunity to make any necessary architectural or design changes early on.

The tooling collects runtime information about a program and can display the results in both graphical and non-graphical forms to help visualization of the program execution and explore different patterns in the execution.

The tools are useful for performance analysis and for gaining a deeper understanding of your Java program. You can use them to view object creation and garbage collection, execution sequences, thread interaction, and object references. They also enable you to see which operations take the most time, and help you to find and solve memory leaks. You can easily identify repetitive execution behavior and eliminate redundancy, while focusing on the highlights of an execution.

1.4 Deployment

This section describes the process of packaging and deploying a J2EE application to the WebSphere Application Server.

1.4.1 Packaging J2EE applications

Before a J2EE application can be deployed, it must first be packaged into a J2EE application EAR file. The subcomponents of a J2EE application can also be packaged individually into their respective, required J2EE containers; for example, a WAR file for a J2EE Web Project, an EJB JAR file for a J2EE EJB Project, and a Client JAR file for a J2EE Client Project.

These operations can be done in several different ways, and this chapter will focus on two options: from a command line interface and using the export tools from within WebSphere Studio Application Developer.

Command line

When performing complex and potentially frequently repeated operations, command line tools can improve the process with the use of scripting to automate the processes, thereby reducing the need for constant human interaction and improving speed and accuracy of large operations. One such commonly used tool for packaging J2EE applications for use in WebSphere Application Server is *Ant*.

Ant (which stands for “Another neat tool”) is a Java based build tool, similar to *Make*, but where *Make* uses operating system shell-based commands, Ant uses Java classes to perform its operations, making it platform-independent. There are many pre-written operations built into Ant, which are sufficient to perform the most common build operations. If extended functionality is required, then additional operations can be written in Java which can then be used by Ant.

Ant uses XML scripts to build, deploy, test, and run many other operations the user may require for Java projects. XML is used to describe the operations required in the build and the *targets* they are to be executed upon. Running the relevant scripts will automatically build and package J2EE projects with no more interaction required from the user.

See <http://ant.apache.org/> for more information about how to use Ant.

Application Assembly Tool (AAT)

The Application Assembler Tool (AAT) is a Java GUI application to help application assemblers to build deployable applications for the WebSphere Application Server.

The deployable packages, enterprise archives (EAR), can be built from scratch or the existing ones can be modified. AAT does not provide support for application development.

For more information on the Application Assembly Tool, refer to the WebSphere Infocenter at:

<http://www-3.ibm.com/software/webservers/appserv/infocenter.html>

WebSphere Studio Application Developer

For each of the J2EE components (EJB JAR, WAR, Client JAR and EAR projects) represented in WebSphere Studio Application Developer, there is an *Export* function which creates a J2EE package for the project.

For example, to export an Enterprise Application Project, you simply select the enterprise application project folder in the Navigator or J2EE Hierarchy View that you wish to export, right-click and select **Export** from the context menu. This will start up a simple wizard to guide the user through the packaging process, and will create a deployable EAR file. This EAR file will include all files defined in the Enterprise Application project as well as the appropriate module archive files for each J2EE module project defined in the deployment descriptor, such as Web archive (WAR) files and EJB JAR files.

The process is identical for packaging individual J2EE components, since WebSphere Studio Application Developer has export wizards for EJB modules (resulting in an EJB JAR file), Web modules (resulting in a WAR file) and Client modules (a JAR file).

When exporting EJB projects (or Enterprise Application projects which contain EJB projects), there is an option in the Export wizard to *Generate Deploy and RMIC code*. This is a necessary step before the user can run the enterprise beans on a server. If this option is selected in the wizard, then it will automatically generate and compile deployment and RMIC code for the selected EJBs.

1.4.2 Deploying the packaged applications

Once the EARs, WARs, JARs or EJB JARs have been exported, they can be deployed to a server. The server in this case is the WebSphere Application Server.

Deployment is the process of installing applications into application servers, without the need for customizing the application code for each server environment. The J2EE 1.3 specification assigns three explicit duties to the deployer:

- ▶ Install the application files on the application server.
- ▶ Configure the application for the particular operational environment.
- ▶ Start the newly deployed application.

The appeal of the J2EE specification is that once the applications have been written and packaged once, they can be deployed to many different environments without changing the code. This is possible because:

- ▶ The application module contains instructions for its deployment in the form of the deployment descriptor.
- ▶ The deployment descriptor provides application specific information, but leaves certain information unspecified, such as the specifics of the environment to which the application is to be deployed. The deployer provides such information at deployment time, along with any other information required for successful application deployment.

For example, the deployment descriptor may contain information about security roles, but it would be up to the deployer to map such roles to actual specific users in the target environment.

As with the process of packaging the applications, deployment can be performed using either command line tools or by using the *WebSphere Administrative Console*.

Command line

There is a command line tool included as part of WebSphere Application Server, called *WebSphere Studiomin*, a powerful scripting interface that supports a full range of administrative commands. WebSphere Studiomin is the non-graphical alternative to using the Administrative Console. The WebSphere Studiomin tool is intended for production environments and unattended operations.

For example, to install an application from an EAR file to a server, the following simple command would be used:

```
$AdminApp install c:/MyStuff/application1.ear {-server serv2}
```

The above command would be enough to automatically install the application1.ear enterprise application to server serv2, with no more interaction required by the user. Similar commands can also be used to modify configuration attributes.

For more information on, and how to use WebSphere Studiomin, see the WebSphere InfoCenter at:

<http://www-3.ibm.com/software/webservers/appserv/infocenter.html>

Also refer to the IBM Redbook *IBM WebSphere Application Server V5.0 System Management and Configuration*, SG24-6195.

WebSphere Administrative Console

The WebSphere Administrative Console is a J2EE Web-based graphical interface which will guide the user through deployment and systems administration tasks. It is extremely useful for helping the user get started exploring the available management options, and there are wizards to guide the user through the more complicated processes.

Deployment of an application is performed by simply selecting **Install New Application** from the console; the wizard will guide the user through the deployment process.

During application installation, the deployer provides environment-specific information required to run the application. The wizard presents the deployer with a series of questions (or GUI panels) to collect configuration information related to the application and its modules. The tools automatically choose default values for various pieces of required information, based on the current environment configuration.

For a full discussion of WebSphere Application Server administration refer to the IBM Redbook *IBM WebSphere Application Server V5.0 System Management and Configuration*, SG24-6195.

Hot deployment and dynamic reloading

WebSphere Application Server supports hot deployment and dynamic reloading, which involve making changes to an application and its contents without having to restart the application server.

Hot deployment is the process of adding new components such as WAR files, EJB JAR files, EJBs, servlets and JSP files to a running application server without having to restart the server. In most cases, restarting the application is all that will be required to pick up the changes.

Dynamic reloading of existing components is the ability to change an existing components without having to restart the application server, for example, changing the implementation of a servlet, or changing a deployment descriptor of a Web module.

Using this approach, there is no need to use command line administration tools or the WebSphere Administration Console. Changes can simply be made to the application files on the file system accessed by the server.

See “Updating applications” in the InfoCenter for more information about hot deployment and dynamic reloading.

1.5 Runtime

Access to enterprise business logic and data from heterogeneous devices and technologies poses a significant technical challenge: the implementation of logic and data needs to be client neutral. Using a multi-layer architectural approach is a natural way to deal with this issue. Enterprise applications can be partitioned in a number of layers that are logically independent, although they may physically reside on the same machine or run in the same process.

The J2EE specification defines a packaging structure which maps perfectly to the multiple layer architecture shown in Figure 1-13.

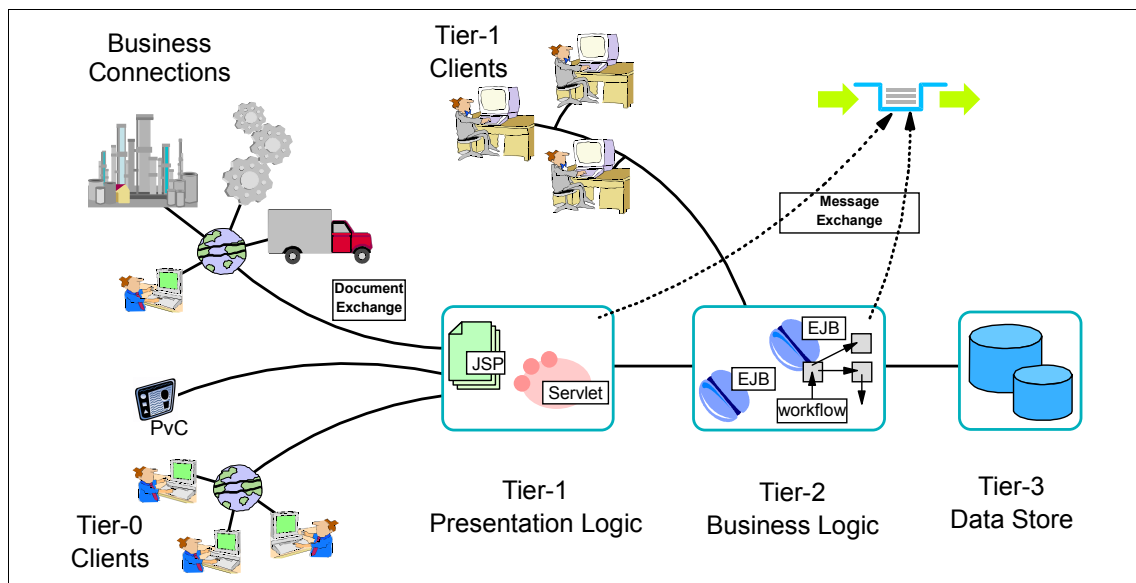


Figure 1-13 Multi-layer architecture

Application servers provide the J2EE-compliant runtime environment to enterprise applications. The application server provides a Web container, EJB container, and various services for enterprise applications. In this section, we will describe the functionality of IBM WebSphere Application Server and its runtime environment architecture.

1.5.1 WebSphere Application Server

IBM WebSphere Application Server V5 is an application server which provides the runtime environment for business applications by implementing the J2EE (Java 2 Enterprise Edition) specification, conforming to the J2EE 1.3 version. WebSphere exceeds many of the J2EE specification implementing additional services in order to provide broader functionality or to bring future extensions into the product earlier.

The base WebSphere architecture gives flexibility to the system architecture level, not addressed in the J2EE specification, by separating the managed processes and implementing an overall administration system.

IBM WebSphere Application Server has some different offerings, each having its own specific capabilities and functions, as follows:

- ▶ WebSphere Application Server Express
- ▶ WebSphere Application Server (base)
- ▶ WebSphere Application Server Network Deployment
- ▶ WebSphere Application Server Enterprise
- ▶ WebSphere Application Server for z/OS®

More details can be found at

<http://www-3.ibm.com/software/webservers/appserv/>, or in the IBM Redpaper *WebSphere Application Server V5 Architecture*, REDP-3721.

In this section, we will focus on the runtime environment of WebSphere Application Server base package and WebSphere Application Server Network Deployment.

Before we discuss the details of the runtime architecture, it is important to understand some fundamental concepts.

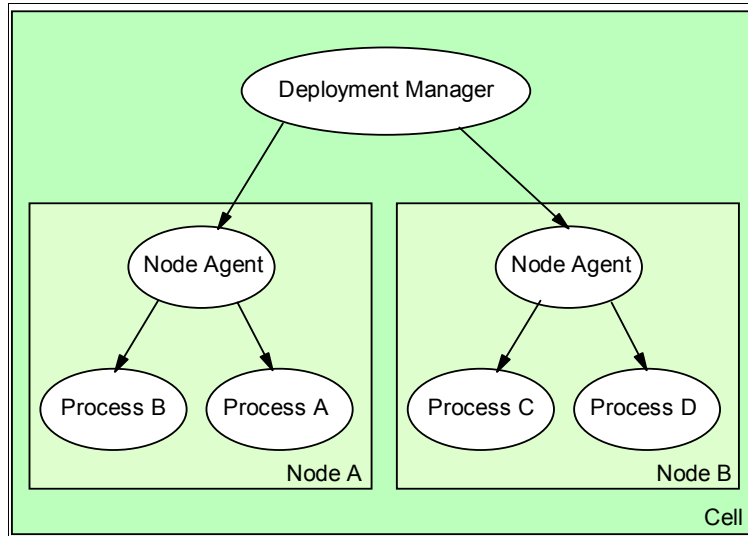


Figure 1-14 System management topology

- ▶ A cell is an aggregation of nodes in a single logical administration domain. In a cell, there is going to be a single Deployment Manager.
- ▶ A node is a set of managed servers on a physical machine in a topology composed of one or more machines. A node contains an IBM WebSphere Application Server installation.
- ▶ A managed process is any instance of a JVM that can be managed in a WebSphere V5 environment. Application Servers are managed processes, but JMS Servers (a special type of server that runs the integrated JMS infrastructure) fall in this category too. Other examples of managed processes are the Node Agent and the Deployment Manager.
- ▶ A node agent is responsible for managing all the servers on the node.
- ▶ The deployment manager manages the multiple nodes in a distributed topology. Its main purpose is to provide the ability to perform centralized administration in the cell.

WebSphere Application Server: base configuration

The base configuration of WebSphere Application Server V5 includes only the application server process. There is no node agent or Deployment Manager in this configuration. Each instance of application server is administered separately. Figure 1-15 on page 40 shows an overview of the runtime architecture in a base installation. The node agent comes with the base product but is not active until the server gets attached to a cell.

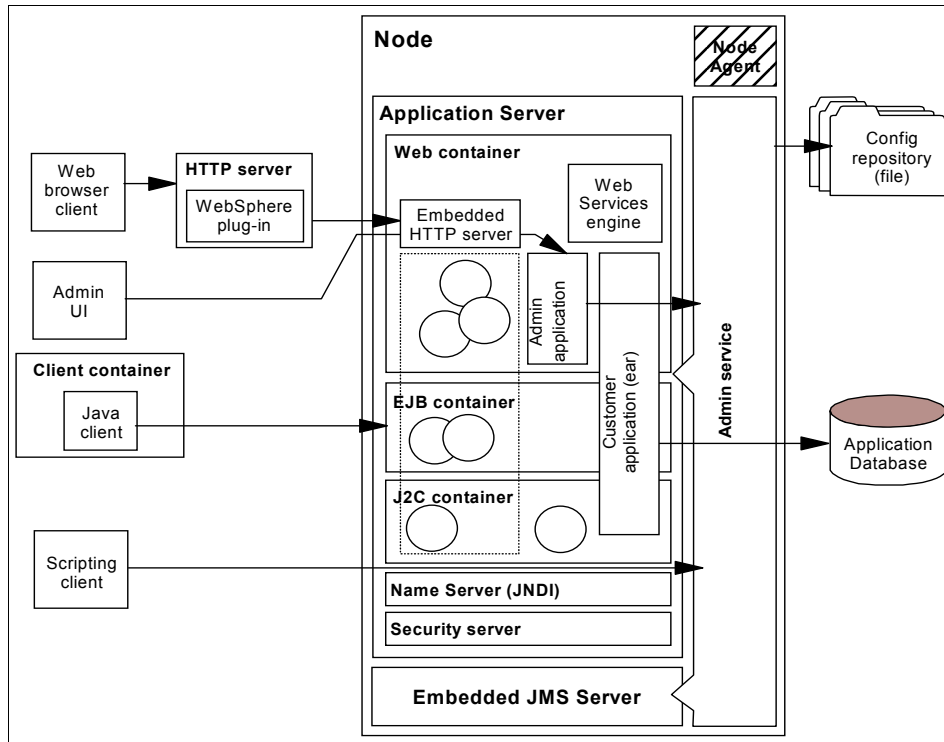


Figure 1-15 IBM WebSphere Application Server runtime architecture

WebSphere Application Server works with a Web server to handle requests for dynamic content from Web applications. The *Web server plugin* is used to communicate between the Web server and the application server and to determine if the request that comes from the client should be handled by the Web server or by the application server.

There is an *embedded Web server* in the application server for exchanging HTTP/HTTPS requests/responses with an external Web server.

The *application server* is the primary component of WebSphere. It runs in a Java Virtual Machine (JVM), providing the runtime environment for the application's code. It provides the containers that execute the Java application components and also provide services to the applications deployed within them. We have already described the containers in a previous section: "Containers" on page 12.

Apart from the containers, the application server provides other services:

- ▶ Object Request Broker (ORB)
- ▶ Security services using Java Authentication and Authorization Service

- ▶ Java Management Extension Framework
- ▶ Transaction management
- ▶ Messaging Interfaces
- ▶ Naming services (JNDI)
- ▶ E-mail interfaces
- ▶ Connectors to back-end systems
- ▶ Database connection (JDBC) and connection pooling
- ▶ Trace service
- ▶ Performance Monitoring Interface.

Web Services technology is provided as a set of APIs in cooperation with the J2EE applications. The WebSphere Application Server V5.0.2 base product supports the following implementations:

- ▶ JSR 101 (JAX-RPC) 1.0 - Java APIs to support emerging industry XML based RPC standards.
- ▶ JSR 109 1.0, which defines the Web Services support within a J2EE environment.
- ▶ SAAJ 1.1, which is the SOAP with Attachments API for Java.
- ▶ WS-I Basic Profile 1.0, which is a set of non-proprietary Web Services specifications, along with clarifications to those specifications which promote interoperability.
- ▶ WS-Security, which describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication.
- ▶ ASTK support, an application server toolkit.
- ▶ SOAP/JMS support that allows SOAP messages to flow over a messaging transport.
- ▶ WSIF (Web Services Invocation Framework).
- ▶ Web Services caching, which leverages servlet caching support and introduces caching SOAP requests.
- ▶ JSR 109 performance monitoring support.
- ▶ UDDI4J V2.0, providing full support of the UDDI V2 specification, support for multiple SOAP transports, debug logging, and configuration capabilities.
- ▶ Apache SOAP 2.3.

The *Admin service* runs within each server JVM, providing functions to manipulate configuration data for the application server and its components. The configuration is stored in a set of XML files and these are stored in the server's file system.

The *scripting client* gives flexibility over the Web-based administration application, allowing administration using a command line interface. It uses the Bean Scripting Framework (BSF) which makes possible the usage of scripting language functions for configuration and control.

There is an *embedded messaging server* which is a full JMS server running in the application server. The JMS server is used for support of message-driven beans and messaging within a WebSphere cell. In the base configuration, it runs in the same JVM as the application server. In the Network Deployment version, it is separated from the application server and runs in a separate dedicated JVM.

The *customer applications* are the business applications that follow the J2EE specification. They are all packaged into Enterprise Application Archives (EAR files).

The *application database* runs on the database server and stores data used by the customer applications.

The *security server* contains the security settings regarding authentication and authorization functionality.

The *name server* provides a service that is used to register all EJBs and J2EE resources hosted by the application server.

WebSphere Application Server Network Deployment

This version of WebSphere Application Server Network Deployment is an extension of the base one. It includes support for multiple nodes, clusters, and load balancing. It provides centralized administration of multiple nodes, allowing you to administer nodes on multiple machines. Figure 1-16 on page 43 shows the details of the architecture.

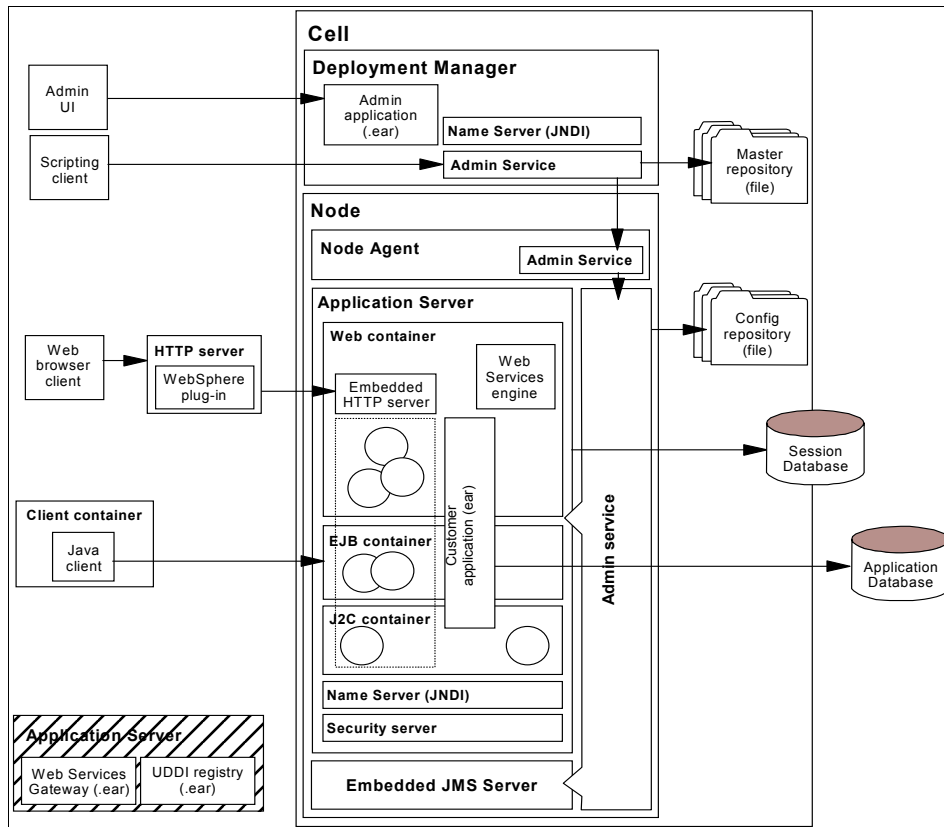


Figure 1-16 IBM WebSphere Application Server Network Deployment runtime architecture

The *Deployment Manager* is one of the components of the Network Deployment package. It provides a single point of administration for all elements in the cell because it contains the master copy of the configuration/application files. The master configuration repository contains all cells' configuration data.

The *node agent* is an administrative process which is responsible for some functions such as file transfer services, performance monitoring and configuration synchronization.

WebSphere Application Server Network Deployment V5 provides a private *UDDI* that implements V2.0 of the UDDI specification.

The *Web Services Gateway* is a runtime component that provides configurable mapping based on WSDL documents. It maps any WSDL-defined service to another service on any available transport channel. You use the IBM Web

Services Gateway to handle Web Service invocations between Internet and intranet environments. You use it to make your internal Web Services available externally, and to make external Web Services available to your internal systems.

This package also includes the Edge Components, which improve the capabilities of load balancing and enhanced caching.

For more information about supported hardware and software APIs of WebSphere Application Server, go to:

<http://www-3.ibm.com/software/webservers/appserv/doc/v50/prereqs/prereq502.html>

1.6 Administration

When we talk about administrating a J2EE application, there are two different categories that come up:

- ▶ Application life cycle management tasks, such as installing, updating, uninstalling, monitoring, new applications in the J2EE Server
- ▶ Server resource management, which includes adding, removing and configuring resource managers, security, availability, performance, and scalability

Companies have critical enterprise applications running on their systems, so managing a dynamic application server environment is not easy.

J2EE specification provides a technology called JMX which provides a management architecture, APIs and services for building Web-based, distributed, dynamic and modular solutions to manage Java enabled resources.

More details about the specification can be found at:

<http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html>

There are tools available that help to monitor the runtime environment and enterprise applications. IBM WebSphere Application Server provides a set of tools which are useful for the system administrator on a daily basis:

- ▶ WebSphere Administrative Console
- ▶ Command-line operational tools
- ▶ WebSphere scripting: IBM WebSphere Application Server includes a scripting tool called WebSphere Studiomin
- ▶ Java APIs: the Java-based JMX APIs can be accessed directly by custom Java applications

The WebSphere Administrative Console is a Web-based administration interface installed as a standard J2EE 1.3 compliant Web application. In the base configuration, it runs on the application server and can only manage that application server. In the Network Deployment configuration, it is installed under the Deployment Manager.

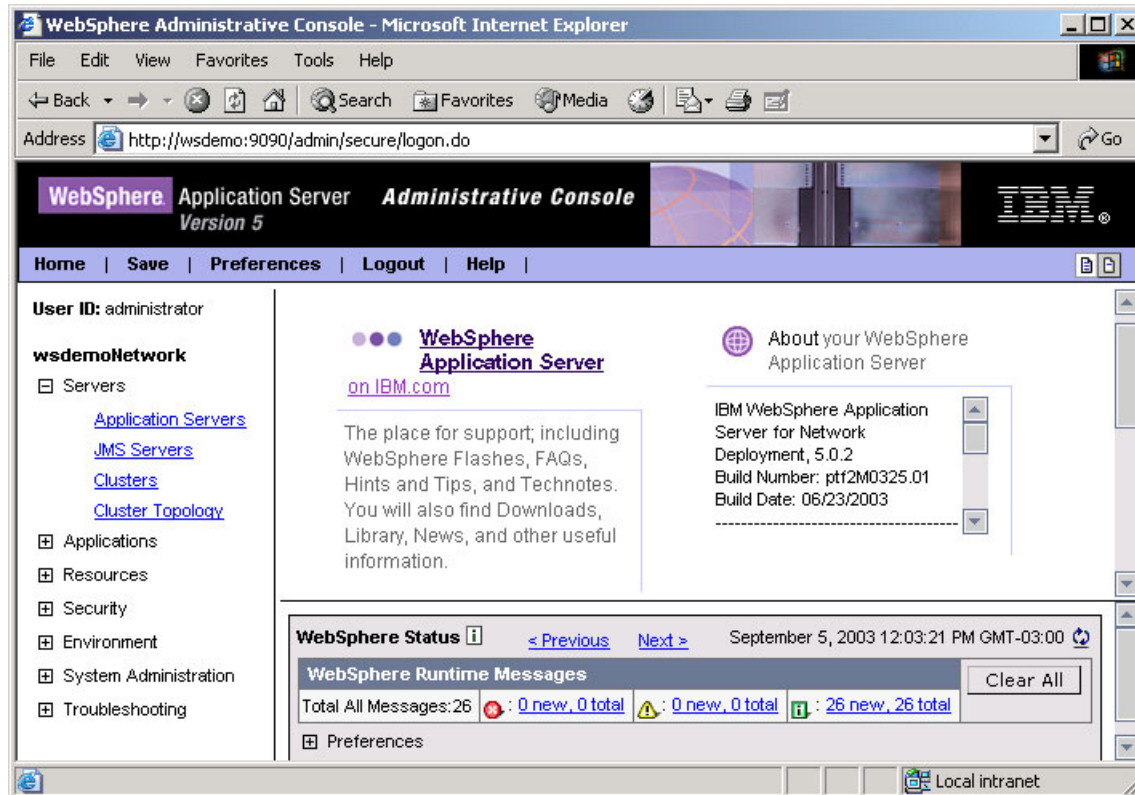


Figure 1-17 WebSphere Administrative Console

The Administrative Console provides a centralized navigation to configuration tasks and runtime information, allowing the administration of multiples nodes and of nodes on multiple machines.

All WebSphere Application Server configuration data is stored in XML files; you can interact with it through the Administrative Console or use the scripting facility called WebSphere Studiomin.

The scripting tool is based on the Bean Scripting Framework (BSF); its architecture works as an interface between Java applications and scripting languages. For example , JACL is a supported language based on the scripting language TCL. One of the major advantages of using JACL and WebSphere

Studiomin is that JACL offers robust features and prebuilt functions. The programming model for JACL is easy for Java developers to use and allows customers to utilize their existing investment in scripting skills.

The advantage of BSF is that scripts are not tied to a specific application or implementation. Any number of languages can access Java objects and interoperate with each other.

The JMX specification defines an interface called MBeanServer for communicating with MBeans; the scripting interface called AdminControl is based on this interface. WebSphere Studiomin uses the same interface as a user would using the Web Console to make configuration changes and control the WebSphere server.

MBeans are simple JavaBeans that perform operational or configuration changes on resources.

The Java Management Extensions (JMX) are a framework that provides a standard way of exposing Java resources to a systems management infrastructure. By *Java resources*, we mean a wide range of objects, such as enterprise applications, Web modules, but also application servers, clusters, and so on.

The framework allows a provider to implement functions such as listing the configuration settings and allowing users to edit them. It also includes a notification layer, which can be used by the management applications to monitor events such as the startup of an application server.

WebSphere Application Server Network Deployment allows you to manage multiple WebSphere Application Server V5 nodes from a single, central location. Distributed administration requires Network Deployment to be installed on a single machine in the environment.

The machine on which you install Network Deployment does not require WebSphere Application Server to be installed. Once you have installed and started the network deployment instance, you can use the addNode tool provided with WebSphere Application Server to add a WebSphere Application Server instance (node) to the network deployment cell. Once a node has been added to a network deployment cell, the Network Deployment Manager for the cell assumes responsibility for the configuration of any application server nodes that belong to the cell. The Network Deployment Manager creates configuration files for each WebSphere Application Server node which has been added to its cell.

WebSphere Application Server Network Deployment provides a master repository of configuration and application data for the cell. WebSphere

Application Server administrative clients are used to provide centralized functionality for:

- Modification of configuration settings in the master repository.
- Installation, update, and uninstallation of applications on application server(s) in the cell. In the process, the Enterprise Application Archive (EAR) files and deployment descriptors are also stored in the master repository.

The changes are committed to the master repository and then published to the nodes of the cell.

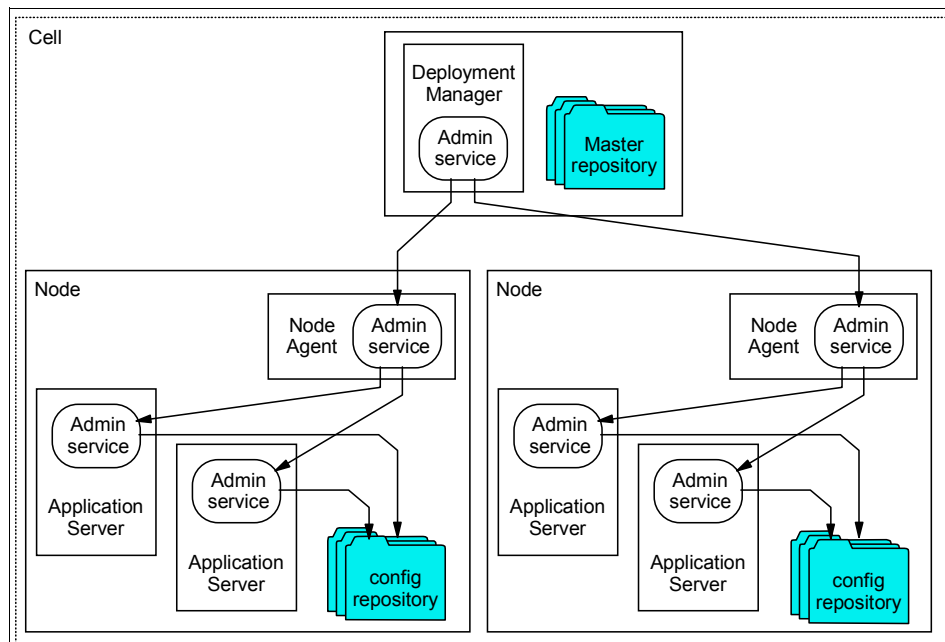


Figure 1-18 Distributed administration



.NET introduction

This chapter provides an overview of .NET, the .NET Framework, its capabilities and the technologies and specifications that make it up. This chapter is intended for those with a knowledge of J2EE and WebSphere who wish to gain a better understanding of .NET.

This chapter discusses the following items:

- ▶ What .NET is and how it is architected, including application components, packaging and deployment, the runtime environment, the standard services provided by .NET and the underlying technologies upon which it is built.
- ▶ Design considerations for .NET applications.
- ▶ Development tools for .NET applications, including the Microsoft® Visual Studio® .NET Integrated Development Environment and other development tools.
- ▶ Testing tools available for .NET applications.
- ▶ How .NET applications are deployed, and the tools available to aid in the management of this process.
- ▶ The application server applicable to .NET Web applications, examining the runtime architecture of application components at all layers within the application architecture.
- ▶ Administration requirements and tools for the .NET application.

2.1 Architecture

This section defines and describes .NET, what it is and how it is implemented. We will cover the overall architecture, application design patterns, the runtime and services provided by .NET and its underlying technologies.

2.1.1 Overall architecture

The question, “What is .NET?” is not often met with a clear and meaningful answer. Two reasons for this might be:

- ▶ Microsoft’s branding strategy uses this name extensively and in a wide variety of products.
- ▶ .NET is a collection of technologies.

Some background and history

Microsoft has been investing in its flagship operating system for over two decades now, improving Windows stability, scalability and performance as well as adding enterprise Quality of Service capabilities through various add-on offerings and Software Development Kits.

With the Component Object Model (COM), Microsoft provided an object-oriented application development model for the Windows platform to replace the Win32 C-style API. This model provided some pseudo language-independence between Visual Basic and C++. COM was later extended with ActiveX to include additional features in the object model and the beginnings of a distributed capability called Remote Automation. This networking capability was further refined in Distributed COM (DCOM).

Microsoft then began adding Quality of Service capability to COM with a transaction coordinator product called Microsoft Transaction Services (MTS) which not only provided commit and rollback capabilities, but also extended the life cycle management of server-side COM objects, allowing advanced capabilities such as multi-threaded object pooling, which manages the life cycle of instances of server-side COM objects as they are utilized by remote clients.

Microsoft then merged these Quality of Service capabilities of MTS with COM creating something called COM+.

Microsoft entered the Web application development market with the release of its Web and application server product called Internet Information Services (IIS). This server, along with Microsoft’s Internet browser (Internet Explorer) supported Visual Basic Scripting as a way to create dynamic Web applications. Combining these Web-centric technologies with the Quality of Service of COM+ begat a

comprehensive architecture for Internet Application development called *Microsoft DNA* (Distributed interNet Applications).

Microsoft DNA is a set of design patterns and an architecture for developing scalable Internet or Web-based e-commerce applications. Central to DNA was a concept of *logical* partitioning of applications into layers, as seen in Figure 2-1.

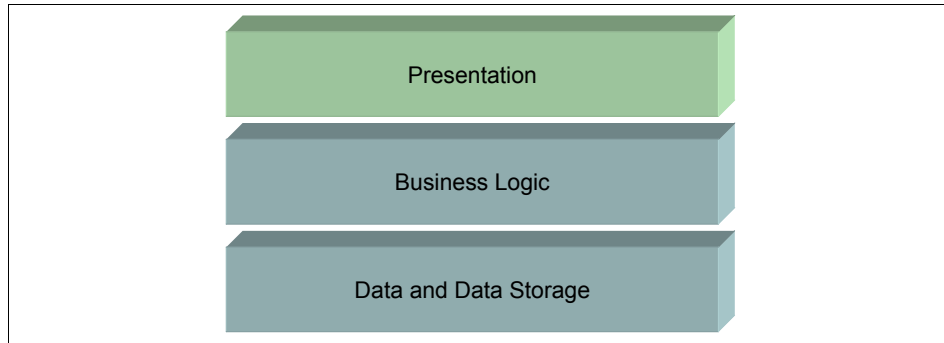


Figure 2-1 Microsoft DNA introduced the n-tier architecture to Microsoft developers.

This logical model was not intended to denote any physical implementation. The intention was for code to reside on the physical node that made the most sense, but each unit of code was to have a function in one of these layers, expose itself appropriately to layers above and use the services of the layers below.

Underlying DNA from an architectural and product standpoint was the operating system, COM+ and several additional products for specific purposes, including:

- ▶ **Internet Information Services:** A Web server and application server that supports hosting of dynamic Web applications. This is analogous to WebSphere Application Server and Apache Web Server.
- ▶ **Active Server Pages:** Written in a variant of Visual Basic, these server-side application components reside on an Internet Information Services and provided presentation to a Web user. VBScript provided a client-side scripting language similar to JavaScript for various tasks.
- ▶ **COM+ Components:** Written in either Visual Basic or Visual C++, these components provided business logic components for an application. These components could live on any physical host and provide services to Web or Windows applications.
- ▶ **Active Data Objects:** Known as “ADO,” this is a set of COM objects used for accessing relational data.
- ▶ **SQL Server:** Providing a Relational Data Base Management System.

Evolution.NET

As Microsoft was beginning to promote DNA as the Internet Application Architecture of the future, other changes were also occurring in the computing industry. Significant among these were Web Services and Java 2 Enterprise Edition. As acceptance of these technologies began to grow, Microsoft evolved DNA and the technologies underlying it to embrace them, and what they created received the name “.NET”.

Microsoft .NET is built upon the underlying operating system and products which made up DNA, including the Windows Operating System and COM+. The OS and COM+ environments have been significantly enhanced and extended to support the new features which have been added on top.

The key values .NET itself provides are as follows:

1. Encapsulates the Windows operating system and its Quality of Service mechanisms within industry standard protocols such as those involved in Web Services and Web applications.
2. Provides a runtime environment for application software with services like garbage collection, exception management and namespace support.
3. Provides enhanced programming language independence and a new programming language - C#.
4. Provides a rich Framework of pre-built classes that perform functions many applications need.

As shown in Figure 2-2 on page 53, the way .NET provides these values is through implementation of a Common Language Runtime and a Framework on top of COM+ and the Operating System. The purpose of the Common Language Runtime is to provide language independence and execution code management. The Framework provides access to the underlying Quality of Service mechanisms such as COM+ and simplifies many tasks such as building and consuming Web Services, Remoting and other features we will discuss in more detail later.

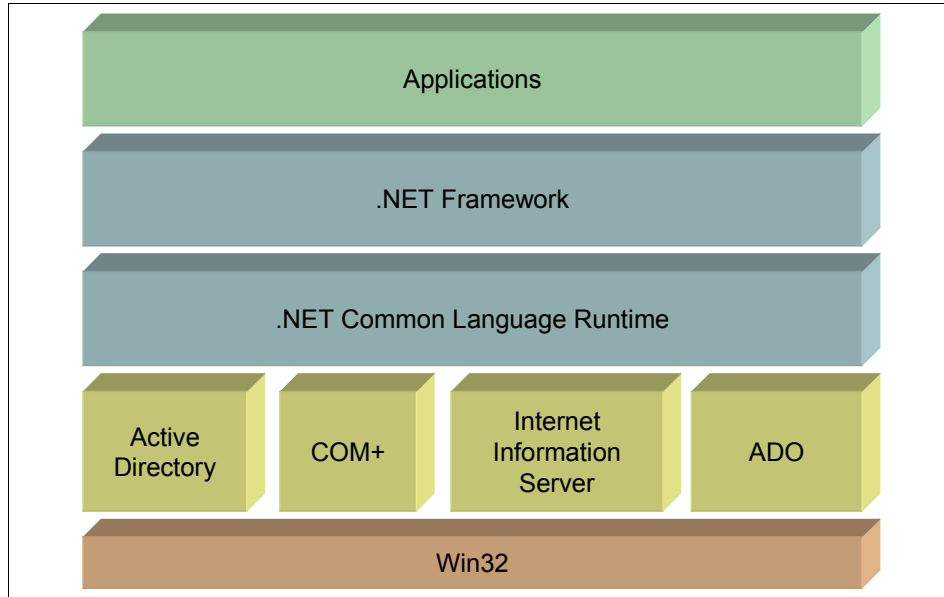


Figure 2-2 .NET provides access to, and adds value to, underlying services

In Figure 2-2, we note that various underlying products continue to play their respective roles while the framework extends them. These products have also been enhanced to provide additional features and many have been rebranded for .NET. For instance:

- ▶ A new version of the OS has been released called Windows Server 2003.
- ▶ COM+ was upgraded from v1.0 to v1.5 and has been rebranded as *Enterprise Services* under .NET.
- ▶ ADO has been enhanced and rebranded as ADO.NET.
- ▶ Internet Information Services has been enhanced and is now at V6.0.

It is important to understand that .NET is built upon (and therefore relies upon and attempts to fully exploit) these underlying products. For instance, .NET provides a new feature called Windows Forms. This feature provides a thick client GUI from an easy to use set of classes. .NET is built from the ground up to take full advantage of Windows.

Key differentiators

If you are familiar with WebSphere Application Server, you know that it is a product implementation of the Java 2 Enterprise Edition (J2EE) specification, in which many of the key services provided to applications are defined. J2EE defines the standardized external protocols used for inter-process and

inter-application communication and integration. It also provides the target platform (the Java Virtual Machine) in which all code runs. WebSphere Application Server adds value to J2EE where it is useful and possible to do so.

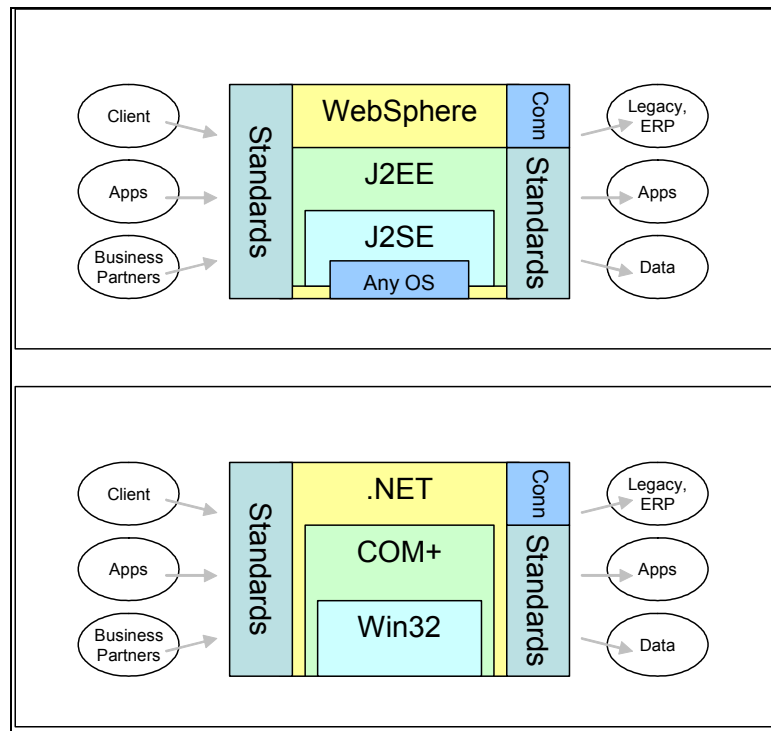


Figure 2-3 The building blocks of WebSphere and .NET

In the same spirit, .NET can be described as a product implementation built upon COM+ and other products from Microsoft. In this case, .NET provides the implementation of standard interfaces such as Web-based access via HTTP, XML and Web Services while utilizing the underlying Quality of Service mechanisms which are optimized for the platform, which is Windows. .NET adds value to COM+ where it is useful.

As products, both WebSphere Application Server and .NET aim at scalability to the enterprise for large e-business and enterprise applications and Web Services, and being as ubiquitous as possible. The services they provide to the applications are very similar in many respects and, as you will see, applications are architected for these environments in strikingly similar ways.

Important: Standing alone, the key distinction between the approaches is this: Java 2 Enterprise Edition's focus is *platform* independence while .NET's focus is *language* independence.

2.1.2 Layered services (application architecture)

Microsoft describes how to architect applications for .NET using published patterns. These patterns are described in a highly consistent manner and are available online at:

<http://www.microsoft.com/resources/practices/default.asp>

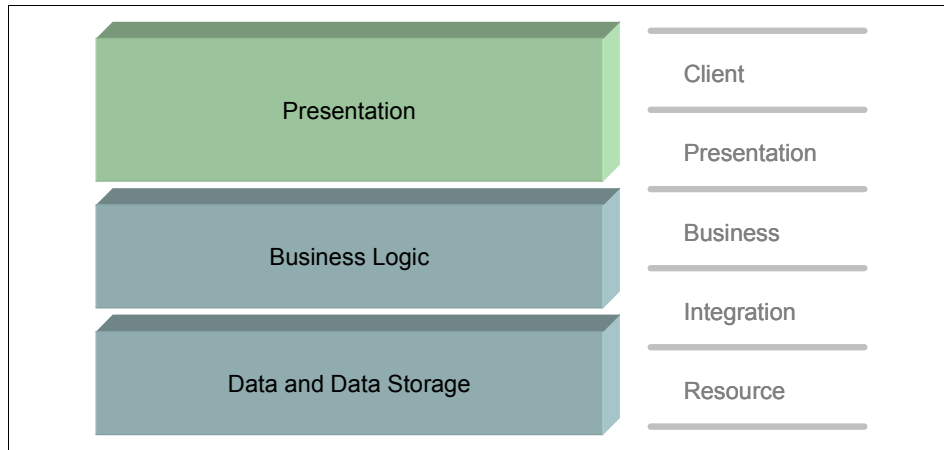


Figure 2-4 Layered Services Architecture advocated by Microsoft Patterns

The design patterns provided by Microsoft for .NET for enterprise applications describe an n-layer model that is not specific about what the layers contain, but rather extolls the benefits of layered design. They then provide simplified three-layer version of the n-layer model as a starting point for designing your own solutions. This three-layer model is similar to that advocated under Microsoft DNA. If we set that model beside the standard model we are using for reference (see Figure 2-4), we see that the Presentation layer of Microsoft's basic pattern encompasses both the client and Presentation layers of our model and that the resource and business layers overlap our integration layer. To help understand the reasons for this, we will discuss each layer of Microsoft's model in more detail and place the components that make it up into our reference model for clarity.

Presentation layer

The Presentation layer provides the user experience. For thin clients, it consists of Active Server Page.NET (ASP.NET) components executed under Internet

Information Services (IIS). Although the .NET Presentation layer may contain client-side scripting via VBScript or JScript, for thin clients the ASP.NET components do the vast majority of the presentation work.

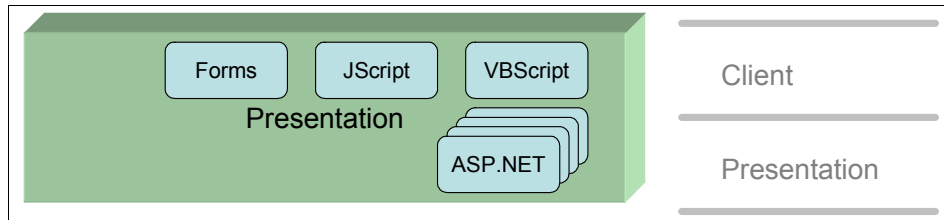


Figure 2-5 Presentation layer

ASP.NET objects can include other ASP.NET objects with these interactions often following the well-known *model-view-controller* pattern.

Table 2-1 .NET to J2EE Analogues - Client and Presentation Layers

.NET	J2EE
Windows Forms	Java Client
JScript and VBScript	Java Script
Active Server Pages.NET	Java Server Pages

Business layer

The business layer provides reusable objects that perform discrete business functions. These are accessed by the Presentation layer to perform these functions and provide business context views of resources and other applications.

The business layer contains objects running under the .NET Common Language Runtime, known as “managed code” and COM+ components which do not run under the CLR, known as “unmanaged code”.

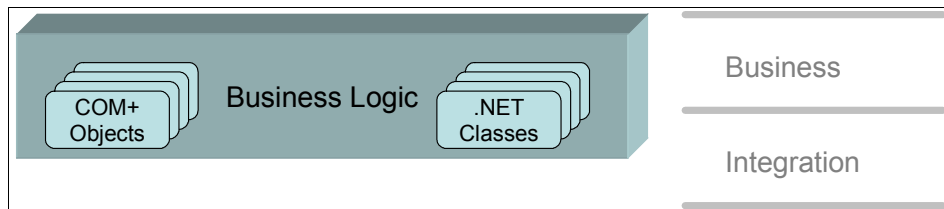


Figure 2-6 Business layer

The placement of these components in the business layer does not imply location. They may or may not run on the same machine as the Web or application server or the fat client. They may run on dedicated application machines or even on the same machines as the resources they utilize.

When objects do not live on the same machine, .NET facilitates remote invocation between these objects using something called “.NET Remoting”. Remoting is analogous to RMI under J2EE and provides for remote invocation of .NET components via SOAP or a proprietary binary over TCP protocol. You can read more about .NET Remoting on the Microsoft MSDN Web site.

Objects within the business layer will work with each other in various layers of abstraction. For instance, one object may exist to provide a service interface to one or more others via an XML Web Service interface.

Table 2-2 .NET to J2EE Analogues - Business layer

.NET	J2EE
COM+ objects	Classes, Beans, Enterprise Java Beans
.NET Classes	Classes, Beans, Enterprise Java Beans
Web Services	Web Services
Remoting	Remote Method Invocation

There are a variety of Quality of Service mechanisms with which objects in the business layer may participate, including transactions and life cycle management. These are discussed in more detail later.

Data/Resource layer

Although Microsoft design patterns refer to this layer as the data layer, our reference architecture expands this to provide access into any resource required by the application business components. This can include databases, files, and interfaces to other systems.

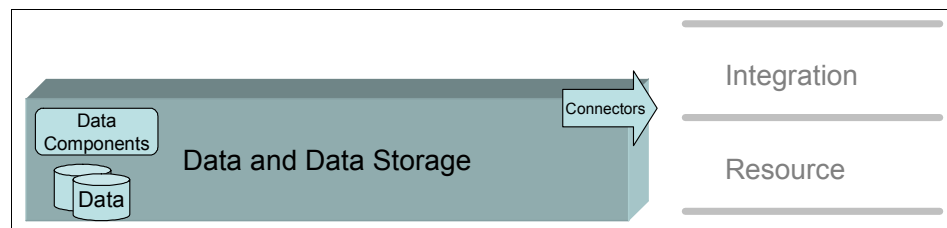


Figure 2-7 Data / Resource layer

The Microsoft patterns advocate abstraction of data and other resources with .NET Classes designed to provide access to Business layer objects. These components isolate the access code into a single location and abstract the data representation and access patterns from the business objects to minimize the impact to business objects of changes in data structure or providers.

Connectors are objects or collections of objects that access specific resource technology, as opposed to business objects which abstract business functions. So, while access to the accounting system might be abstracted at a business level by a business object, a connector may be used by that object to provide access into the technology behind which the accounting system sits, for example the middleware used to access the accounting system.

Summary

To complete the picture and relate the architecture of a .NET application with that of a common WebSphere application, Figure 2-8 on page 58 depicts application architecture under .NET using our reference architecture. This picture contains acronyms for brevity, many of which we have already covered and all of which are at least defined in this book.

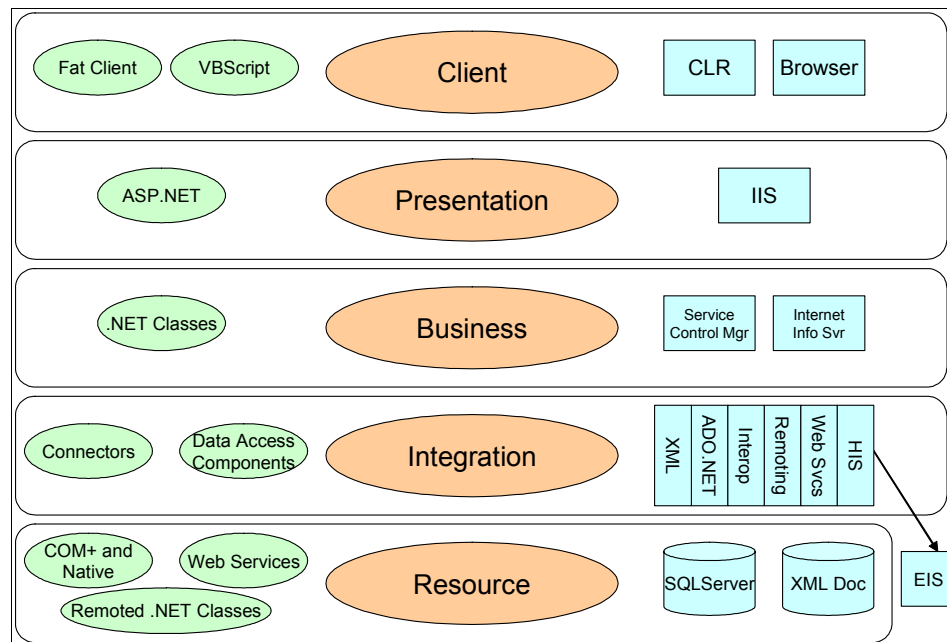


Figure 2-8 Application Architecture under .NET

If we compare this figure with Figure 2-8, we find that .NET and J2EE offer solutions to many of the same problems. In the end, building enterprise scale Web based or Web enabled applications is the primary goal of both.

Important: When it comes down to it, the J2EE developer discussing 'beans' and 'containers' and the .NET developer discussing 'components' are talking about the same thing.

2.1.3 Standard support

The .NET Framework is built on top of the Microsoft Windows operating system, it is a proprietary Microsoft technology. Although .NET itself implements many standards, the application environment is not an implementation of any standard.

Once an application is developed for the .NET environment, there are no alternatives for running or porting the application to another application server or another operating system.

2.1.4 Platform support

The Microsoft .NET Framework is designed for Windows operating system. To run any .NET application, the client or server must have a runtime called *.NET redistributable*, the same as the Java Runtime in WebSphere environment. The .NET redistributable is freely available on the Microsoft site. For configuring server side application, it is recommended that you use Windows 2000 Server or higher. The following list describes various possible Windows platforms for client and server application.

- ▶ Client
 - Microsoft Windows 98 and editions
 - Microsoft Windows Millennium Edition
 - Microsoft Windows NT® 4.0 Workstation with Service Pack 6.0a or later
 - Microsoft Windows NT 4.0 Server with Service Pack 6.0a or later
 - Microsoft Windows 2000 Professional
 - Microsoft Windows 2000 Server family
 - Microsoft Windows XP Home Edition
 - Microsoft Windows XP Professional
 - Microsoft Windows Server 2003 family
- ▶ Server
 - Microsoft Windows 2000 Professional with Service Pack 2.0

- Microsoft Windows 2000 Server family with Service Pack 2.0
- Microsoft Windows XP Professional
- Microsoft Windows Server 2003 family

2.1.5 Programming languages

While WebSphere focuses one language on many platforms, the Microsoft .NET focuses multiple language support on the Microsoft Windows platform. The .NET languages follow *Common Language Specification (CLS)* - the minimum set of features that compilers must support to target runtime. Currently, the .NET supports more than 20 languages, including vendor supported languages.

The Visual Studio .NET comes with following languages developed and supported by Microsoft:

- ▶ Visual Basic.NET
- ▶ Microsoft C#
- ▶ Microsoft J#
- ▶ Visual C++

Moreover, the Visual Studio .NET supports scripting languages like JScript.NET and VBScript. There are numerous other third party languages also available.

The .NET Framework supports so many languages because each language compiler translates the source code into Microsoft intermediate language which is a CPU-independent set of instructions which can be efficiently converted to native code.

2.1.6 Deployment units

The .NET deployment is different from the Java 2 Enterprise Edition. Assemblies are deployment units and are basic building blocks used by the .NET Framework. When an application is compiled in .NET, the output of the compilation produces an *assembly* which can be either an executable file (*.exe) or a dynamic link library file (*.dll). In general, an assembly consists of four elements:

1. Assembly metadata
2. Type metadata
3. IL code
4. Resources

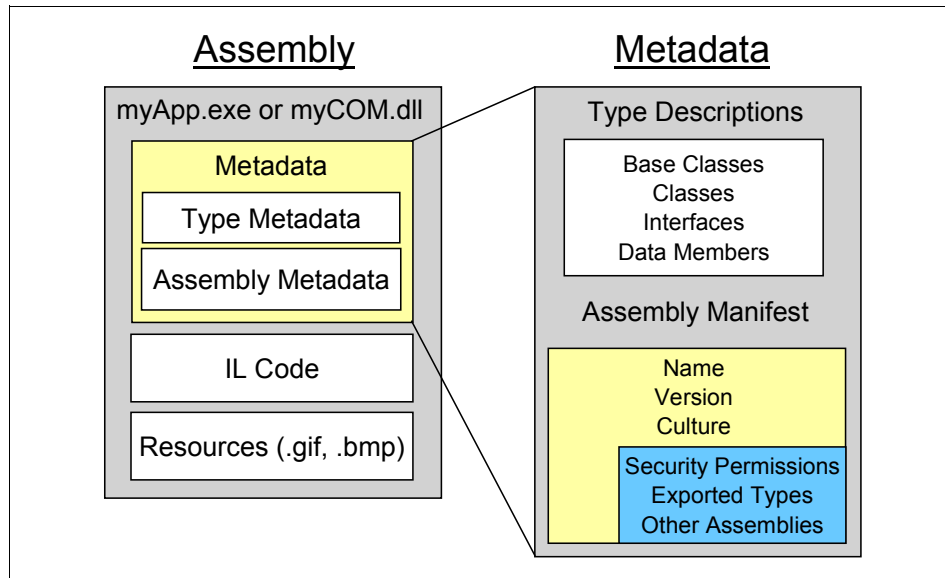


Figure 2-9 Assembly and Metadata

Assemblies do not face problems like DLL or versioning because they are self-describing through *metadata* called a *manifest*. The manifest contains the assembly's identity and version information, a file table containing all files that make up the assembly and the assembly reference list for all external dependencies. The self-describing nature of assemblies overcomes the dependency on the registry and therefore simplifying deployment. The Common Language Runtime uses the manifest at runtime to ensure the proper version of a dependency is loaded.

The resources in an assembly consist of image files like .bmp or .jpg and other files required for application.

The assembly can be *private* or *shared* depending on the visibility.

A private assembly is an assembly that is visible to only one application and deployed in within the directory structure of the application. The CLR finds these assemblies through a process called *probing*. The version information is not enforced for private assemblies because the developer has complete control over the assemblies.

A shared assembly is used by multiple applications on the machine and stored in Global Assembly Cache. Assemblies deployed in the global assembly cache supports *side-by-side execution* and must have a strong name to avoid any confliction. *Side-by-side execution* in the .NET Framework allows multiple

versions of an assembly to be installed and running on the machine simultaneously, and allows each application to request a specific version of that assembly.

The behavior of assembly or application can be changed using *configuration files* which are discussed in next section.

Configuration files in .NET

The common language runtime locates and binds to the assemblies using a configuration file. The configuration file contains information related to application, machine or security settings and stored in XML format. By updating the configuration file, the developer or administrator can change the way in which the application works. Settings in configuration files can be accessed in code by using interfaces. The .NET has three basic configuration files:

- ▶ Application configuration file

The application configuration file consist information about the application. The Common Language Runtime uses this configuration file to get the information about assembly binding policy, the location of remote objects and the ASP.NET runtime settings.

The executable applications, like Windows Forms or Console application settings, are stored in the [applicationName].exe.config file.

The Web applications like ASP.NET or Web Service settings are stored in the Web.config file.

- ▶ Machine configuration file

The machine configuration file contains information about the machine that can be applied to the ASP.NET runtime, built-in remoting channels and assembly binding. This file is located in the CONFIG directory of the runtime install path of the .NET Framework as shown:

C:\WINNT\Microsoft.NET\Framework\v1.1.4322\CONFIG

Note this file is stored in different folder for each version of .NET. In the above example, the v1.1.4322 refers to version 1.1 of the .NET Framework.

- ▶ Security configuration file

The security configuration files contain information about the code group hierarchy and permission sets associated with a policy level.

This file is stored depending on policy level. For example, for enterprise policy the file enterprisesec.config is stored in the same folder as the machine.config file. The user policy configuration file named security.config is stored in the user profile sub tree folder.

2.1.7 Runtime execution environment

The *Common Language Runtime (CLR)* is the heart of the .NET Framework and provides a runtime environment for .NET applications. The CLR is similar in function to the *Java Virtual Machine (JVM)* in the WebSphere environment and provides a fundamental set of services that all programs can use. The Java bytecode can be interpreted as well as compiled, and it is the only language that the JVM runs. The Common Language Runtime has ability to compile managed code once and run on any CPU and operating system that supports the runtime. The CLR runs Intermediate Language, which is created from any .NET programming language, such as VB.NET and C#. The figure below shows the role of the Common Language Runtime while executing the managed code.

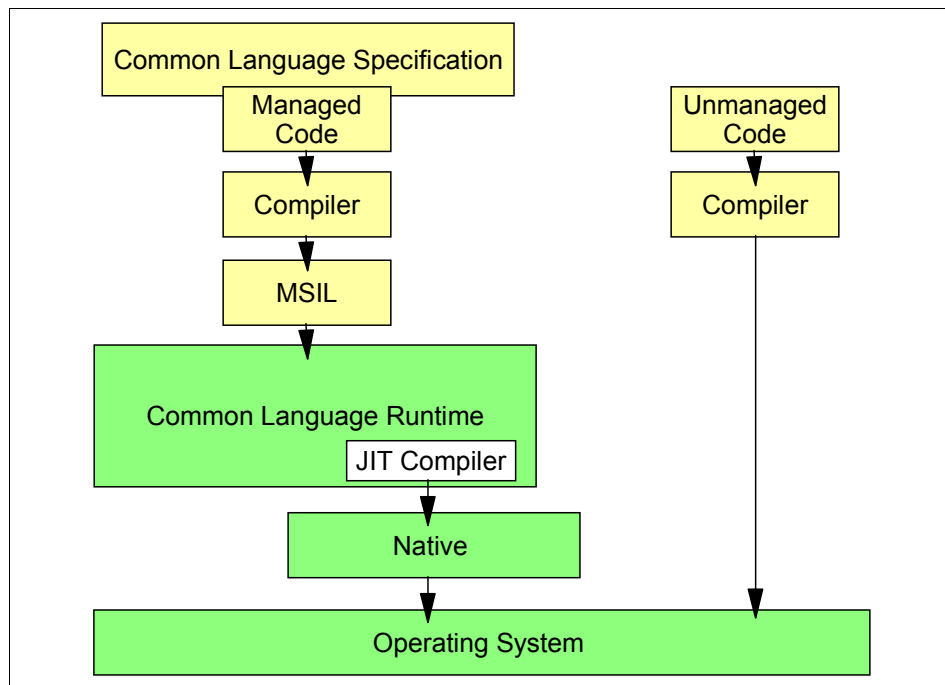


Figure 2-10 Common Language Runtime in .NET Framework

In the .NET Framework, managed code is the code which follows the *Common Language Specification (CLS)*. The managed code gets compiled into the *Microsoft Intermediate Language (MSIL)*. The Common Language Runtime provides a *just-in-time* compiler that compiles the MSIL into the machine language and then runs it, because all programs use the common services in the CLR, no matter which language they were written in. The Common Language Runtime follows the *Common Type System (CTS)*, which is a master set of data

types. Due to the Common Type System, managed code written in various languages can interoperate with programs written in another CLR language.

Some of the features of Common Language Runtime are:

- ▶ Garbage collection
- ▶ Cross language integration
- ▶ Base Class Library Support
- ▶ Thread Support
- ▶ Exception Manager
- ▶ Security
- ▶ IL to Native
- ▶ Class Loader

Web Container

The role of the Web Container is to provide a standards-based wrapper around the application interfaces so they can be accessed via industry standard protocols such as HTTP.

The Web Container under .NET is Internet Information Services (IIS). Its role is to contain Web content including HTML, DHTML and ASP.NET content, as well as manage access into other components such as Web Services and Remoted objects. IIS also hosts and manages other Internet content such as FTP sites and SMTP servers. IIS manages security around all Internet resources.

IIS may be managed via the Internet Services Manager, via the command line or via a Web based administration interface. Web application configuration and management is a combination of managing IIS and configuring the application components via XML-based configuration files that are stored in the directory from which the application or component is served by IIS. These files are always called web.config and they can be used to control a great deal about how the application operates, including session management, runtime debugging settings, security access control and much more.

Internet Information Services (IIS) performs the following functions:

- ▶ Serves HTML, DHTML to browsers.
- ▶ Serves Active Server Pages.NET (ASP.NET) pages and components for Web based applications.
- ▶ Manages access to Web Services.
- ▶ Manages access to .NET Classes that support Remoting via HTTP. We will discuss this in more detail later.

2.1.8 Life cycle management

Life cycle management discusses how code comes into and out of execution. Under .NET, much of the life cycle management that occurs is inherited from underlying technology and the operating system, but the Common Language Runtime is a major new component in this management process. Here we will overview the systems and subsystems that provide life-cycle management for the various kinds of components that make up .NET applications.

All .NET applications run under the low-level control of the Common Language Runtime. In addition, Web based applications, remotable service providers and Web Services run under the control of Internet Information Services. Services and server-side components run under the control of the Service Control Manager.

Common Language Runtime life cycle management

In .NET, the Common Language Runtime dynamically loads assemblies as they are required. The first time an assembly is loaded, it is just-in-time (JIT) compiled to compiler-specific code at the method level. The first time a method is called, it is JIT compiled by the Common Language Runtime and executed. If it is called for again within the same process, the compiled code is executed. This process of JIT compiling and executing occurs at the method level until execution is complete.

During execution, the Common Language Runtime is responsible for managing memory via the garbage collection mechanism. The garbage collector allocates and releases memory as execution takes place. When a new process is initialized, a contiguous amount of memory is allocated for the process by the Common Language Runtime. This space is known as the *Managed Heap*. As each new object is created, memory is allocated from the managed heap contiguously. Allocating memory from the managed heap is faster than allocating unmanaged memory and, because the address spaces for each new object are contiguous, accessing the objects also occurs quickly. The Common Language Runtime has several performance optimizations as well, including allocating large objects in a separate heap by dividing the heap into three areas, called *generations* based on the types and ages of objects collected.

As memory is no longer required by the application, the Common Language Runtime determines the best time to collect those unused items and performs the collection. There is a cooperative relationship between the Just-In-Time Compiler and the Common Language Runtime that allows the Common Language Runtime to determine when objects and memory are no longer needed by an application. The Common Language Runtime performs collection processes and various times throughout the life-cycle of a process to clean these up.

Internet Information Services life cycle management

In addition to the services provided by the Common Language Runtime, the life cycle of Active Server Page.NET (ASPX) application components is managed by Internet Information Services and run under its control and within its process.

The Active Server Pages.NET page framework functions in an entirely stateless and disconnected model. As such, each time a page is requested by a client, it is loaded and when execution is complete, it is unloaded. During this life-cycle, a variety of events are generated in a specific sequence. During each of these events, some information is available to the page and the controls it contains and some information is not. Certain information is persisted between these phases and some is not.

For a complete discussion of these events, please refer to the Microsoft Web site at the following address:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcontrolexecutionlifecycle.asp>

In addition to traditional Web applications, .NET classes that support Remoting via HTTP/SOAP and Web Services run under the control of the Internet Information Services. For more information on Remoting, see “Remote invocation” on page 68. For more information on Web Services, see “Web Services” on page 69.

Server components

In addition to the management provided by the Common Language Runtime, Enterprise Services under .NET provides the ability to create and run .NET applications as *server-side* components. When a .NET assembly is made available as a server-side component, its execution is isolated from the process of the consumer application. Server-side components run in their own process which is managed by the Service Control Manager (SCM).

The Service Control Manager is responsible for bringing the object into existence when it is requested by an application and handing a handle back to the client that called it. Sophisticated operations such as object pooling are made possible via the Service Control Manager, allowing multiple instances of an object to be brought up based on the number of incoming requests for that object. Objects that are managed by the Service Control Manager implement methods which the Service Control Manager calls in order to manage the life-cycle of each instance of the object.

Like library objects, server objects can be configured to function in a variety of ways, including determining the security context under which they run, whether they participate in transactions, whether they are activated under the caller’s context or under another well-known context. In addition, server-side objects can

implement special invocation patterns such as Object Pooling in order to manage multiple clients and the resources each resource is using. This is discussed in more detail in “Object pooling” on page 67.

Windows Services

Windows Services were formerly known as NT Services. They have been around a long time. Windows Services are applications that run in the background. They can be started and stopped administratively locally and remotely whether a user is logged on the system or not.

Windows Services often provide background “daemon-like” functions in the Windows operating systems and typically do not interact with the screen. .NET applications may function as NT Services.

The execution life cycle of Windows Services is managed by the Service Control Manager, which uses methods implemented by the application developer to manage the execution life cycle.

As before, Windows Services differ significantly from other types of applications written for .NET. For a more complete description of both the life-cycle management process and the differences between Windows Services and other applications, please see the Microsoft Web site at:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconintroductiontontserviceapplications.asp>

Object pooling

Object pooling can be used to improve application performance in cases where object instantiation is a relatively time consuming operation. When a poolable object is instantiated (typically by an object factory), it is placed into a pool. When an object is needed, an unused object may be retrieved from the object pool. Not only does connection pooling help to reduce instantiation costs, it also helps to reduce the cost of garbage collecting additional objects at runtime. In general, object pooling is most beneficial when objects can be and are frequently reused.

There are two primary types of object pooling provided by the .NET Framework. The most common type is pooling of ADO.NET database connection objects. This type of pooling is very similar to JDBC connection pooling. In ADO.NET, pooling occurs on a per-provider basis. For example, SQL Server and DB2 providers each provide their own connection pooling facility. ADO.NET connection pooling is typically controlled by an option on the connection string. See 10.5.3, “ADO.NET” on page 460 for more information on ADO.NET.

The second type of object pooling is provided through interfaces and classes provided by the System.EnterpriseServices namespace. This namespace

provides .NET components access to COM+ enterprise services, including object pooling. Objects pooled in this manner must inherit from the `ServicedComponent` class. By applying the `ObjectPooling` attribute to a serviced component, values such as the maximum and minimum pool size may be specified to tune performance.

2.1.9 Remote object discovery

At the time of this writing, the .NET Framework does not provide a standard naming and location service for publishing and locating remote objects. Instead, an application using a remote object must know the URL of the object it wants to reference.

Do not confuse .NET remoting object discovery with Web Service discovery. The .NET Framework contains a full implementation of UDDI, the Web Service discovery protocol. Web Service discovery is discussed further in section 2.1.11, “Web Services” on page 69 and in later sections.

2.1.10 Remote invocation

Similar to Java Remote Method Invocation, .NET provides a mechanism called *Remoting*. Remoting provides an object framework allowing .NET objects to run and communicate in a distributed environment. Remoting in .NET does not use Microsoft’s DCOM or COM+ facility. Instead, it is exposed through a set of .NET classes. It offers greater flexibility than DCOM, including the ability to communicate over multiple types of channels. By default, the .NET Framework provides two types of channels, TCP and HTTP. These channels have both advantages and disadvantages that make them better or less suited for specific types of applications.

The .NET TCP channel uses a binary socket connection, similar to DCOM. The TCP channel is used in conjunction with a binary object formatter. The main advantage of using TCP is that it is typically faster since data is transmitted in binary format. There is little to no additional parsing involved in packaging and unpackaging data to be sent over the network. The main disadvantage of using a TCP channel is that it requires additional TCP ports to be accessible and is typically only suited for an intranet environment.

The .NET Framework provides an additional channel, the HTTP channel, for providing remoting capabilities. The HTTP channel uses the XML SOAP format for remote method invocation. The advantage of using the HTTP channel is that it uses the standard HTTP protocol so it can be used over the Internet without opening additional TCP/IP ports in a firewall. The disadvantage of using a HTTP channel is that it is typically slower due to the added processing required when

formatting requests and replies to and from XML and .NET objects. See 10.4.2, “.NET Remoting” on page 454 for a more in-depth discussion on .NET Remoting.

2.1.11 Web Services

The Microsoft .NET Web Services Framework is similar to IBM WebSphere in that it uses standard protocols and services to provide and locate Web Services.

In addition, the .NET Framework provides components for building Web Services with these standard protocols and services:

- ▶ Simple Object Access Protocol (SOAP)
The .NET Framework supports both RPC and Document style SOAP.
- ▶ Web Services Discovery Language (WSDL)
- ▶ Universal Description, Discovery, and Integration (UDDI)

In addition to using automated features of Visual Studio .NET to facilitate creating Web Services, Web Services can also be created in a less convenient, but more flexible manner. The .NET Software Development Kit (SDK) contains several tools for working with Web Services.

- ▶ soapsuds.exe
The soapsuds tool generates runtime assemblies capable of accessing a remoting service provider given a XSD definition.
- ▶ wsdl.exe
The wsdl tool generates Web Services and Web Service client proxies based on a WSDL definition.
- ▶ disco.exe
The disco tool searches a specified Web server for Web Services discovery documents and saves them locally. Typically, disco.exe is used to discover Web Services and then wsdl.exe is used to generate client code to access the service.

All these tools are built based on classes provided by the .NET Framework. Therefore, all the same functionality is available at runtime and may be used within an application for dynamic Web Services discovery and consumption.

2.1.12 Transaction management

Transactions are a key ingredient in building remote and distributed applications. By using transactions, remote and distributed applications can run multi-step operations and roll back any or all of these operations if there is a failure. The .NET Framework provides support for the following transactional components:

- ▶ Microsoft Message Queuing (MSMQ)

Microsoft Message Queuing support is provided directly through objects in the .NET System.Messaging namespace. This namespace contains the object MessageQueueTransaction which provides MSMQ level transactional support. This is typically called a one-phase or level one transaction since it encapsulates one level of processing.

- ▶ ADO.NET data transactions

ADO.NET data providers typically include database level transaction support. The larger ADO.NET database providers, including IBM DB2 UDB, Microsoft SQL Server, and Oracle also provide the capability to allow database connections to automatically participate in distributed transactions. Like Microsoft Message Queuing, non-distributed ADO.NET transactions are a one-phase transaction.

- ▶ Serviced components

The .NET System.EnterpriseServices namespace, as introduced in “Object pooling” on page 67 also includes the capability to provide distributed transaction support. .NET Serviced components are built on top of COM+ and must be registered as a COM+ component. A registered component may participate in distributed transactions involving other transactional components. Wrapping a Microsoft Message Queuing or ADO.NET transaction in a distributed transaction is generally referred to as multi-phase commitment.

The level of transactional support required varies greatly from application to application and must be determined on a per-application basis. Generally, transactions involve a considerable amount of overhead and should only be used where they are required. Non-distributed transactions typically have less overhead but are limited to their respective component while distributed components require COM+ services and must coordinate transactions through a third entity. On Microsoft platforms, this entity is the Distributed Transaction Coordinator (DTC). A complete discussion of the Distributed Transaction Coordinator is beyond the scope of this book.

2.1.13 Security

Security is a critical component of enterprise applications. The ability to effectively provide authentication, access control, data integrity, privacy, non-repudiation, and auditing within an enterprise application are some of the requirements of building a secure system.

The .NET Framework contains a multi-layered security approach to meet these requirements. Each layer provides various services which may be used to

secure applications at different levels. At the most general level, .NET provides the following security services.

- ▶ Operating system level security

At the lowest level, the operating system controls access to the file system and other system level resources.

- ▶ Runtime code level security

The .NET Common Language Runtime does strict runtime type verification and code validation. These two features help to eliminate code execution problems caused by type mismatches, bad function pointers, memory bounds overruns, and many other runtime problems.

- ▶ Tamper-proof assemblies

Assemblies can be strong named by signing them with a public/private key pair. When the assembly is signed, a hash is generated based on the contents of the assembly. This hash is then encrypted with the private key. If the contents of the assembly are changed and the assembly is not re-signed, the hash will no longer match, thus the assembly will be marked as corrupt. Signing assemblies can be extremely important in keeping them from being injected with malicious code.

- ▶ Role-based security

The role-based security allows for application-level security within the .NET Framework. Role-based security can be configured at the enterprise, machine, and user levels. This is accomplished by creating permission sets based on attributes such as site, URL, and publisher and then applying them to code groups. Policies can be configured directly with XML configuration files or by using the .NET configuration snap-in, available from the Windows control panel.

Note: Role-based security is configured using XML files located on the file system. Without access restrictions on these files, they can be easily modified to allow unauthorized access to resources.

- ▶ Secure Sockets Layer (SSL)

Secure Sockets Layer is the industry standard for network data encryption. It may be used transparently by various .NET components to provide secure network communications.

- ▶ Data encryption components

The System.Security.Cryptography namespace of .NET Framework includes classes for cryptography, hashing, and authentication. Many standard

symmetric and asymmetric cryptography providers are supported, including RC2, RSA, and TripleDES.

► Authentication services

Several methods of authentication are available for use by .NET applications and services. Authentication comes into play when connections are initiated with a remote service. The most common users of authentication services are:

– ASP.NET

Supported authentication methods include Integrated Windows, Forms, and Microsoft Passport. However, authentication can also occur at the IIS Web server level. IIS allows anonymous, basic, digest, certificate, and integrated authentication.

– ADO.NET

Authentication methods vary from provider to provider. For example, the Microsoft SQL Server provider supports both integrated and SQL Server authentication.

– Remoting

When hosted by IIS, remote objects can use IIS and ASP.NET Windows authentication. If a TCP channel is used, there is no built-in authentication service available.

– Messaging

Built-in authentication, authorization and encryption services are available when using messaging services.

– Web Services

IIS and ASP.NET services are also available to Web Services. Other custom approaches, such as passing them as part of the SOAP header, are also available.

2.1.14 Load balancing and failover

Load balancing and failover capabilities under .NET come from the underlying and supporting technologies upon which it is built.

Server clustering

Server clustering has been around since the days of NT and has been enhanced to provide additional features. With clustering, a multi-server Web application can provide service despite hardware failures on individual services.

Although clustering is very powerful, it is a non-trivial implementation which must be properly planned in advance. Shared disk storage is ideally used between the clustered servers to hold common configuration and state information. The application is written to react to lost connectivity and services by re-establishing connections to database and other resources when they suddenly become unavailable.

For more information on Microsoft Server Clustering, see the Microsoft Web site at:

<http://www.microsoft.com/windows2000/technologies/clustering/default.asp>

Network Load Balancing

The Windows Server provides a feature known as Network Load Balancing, which is designed to evenly distribute Web-based traffic between servers and, should a server become unavailable, reroute traffic to another server.

In practice, Network Load Balancing is also a function of the network itself. Any considerations of this should involve careful selection of the technology to be used based on the application requirements (HTTP, FTP, SMTP, ports, other).

For more information on Network Load Balancing from Microsoft, see the Microsoft Web site at:

<http://www.microsoft.com/technet/treeview/default.asp?url=/TechNet/prodtechnol/windows2000serv/deploy/confeat/nlbvw.asp>

2.1.15 Application logging

Application logging includes the ability to track and monitor performance, provide auditing capabilities, and debug problems. The .NET Framework provides several technologies that can be used to provide standard logging features. These technologies are:

- ▶ Performance counters

The .NET Framework provides components for creating, updating, monitoring, and grouping performance counters.

- ▶ Trace class

The .NET Framework includes a standard class named Trace for tracing program execution. Trace may be enabled using online precompiler directives, compiler options, or at runtime using the configurable TraceSwitch class.

- ▶ Windows Management Interface (WMI)

A .NET application can monitor and record system status using the Windows Management Interface. This resource may be accessed through classes in the System.Management namespace.

- ▶ Windows event log

Complete access to the Windows event log is available through the EventLog component in the System.Diagnostics namespace. Using this component, applications can create new events, read existing events, and respond to events requiring input. See Figure 2-11 for an example of entries created in the Windows event log using .NET.

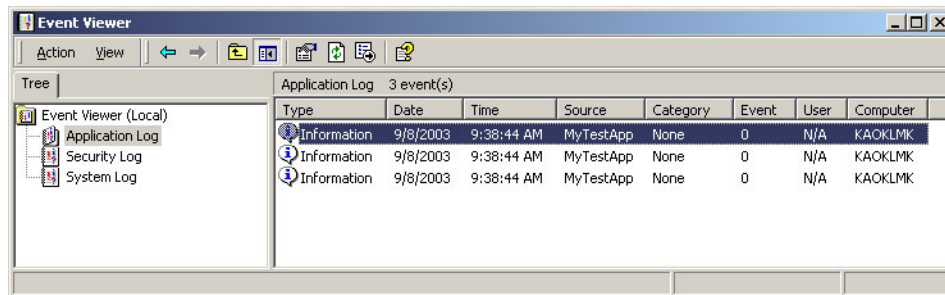


Figure 2-11 Creating entries in the Windows event log using .NET

In addition to these facilities, additional logging for Web applications is provided by Internet Information Services (IIS) and ASP.NET. Internet Information Services and ASP.NET provide the following additional features:

- ▶ ASP.NET component trace

This type of logging can be enabled on a per-application or per-page basis. ASP.NET trace allows trace data to be displayed on the current page or in an external document.

- ▶ Internet Information Services Web site and application-level configurability

The majority of logging configuration capability resides as site level configuration options. At the site level, options such as a client IP Address, user name, and host port may be chosen. At the application level, the option whether to log visits may be changed.

2.1.16 Versioning

The .NET Framework provides a robust scheme for versioning assemblies. By versioning assemblies, multiple versions of a same named object, in the same namespace, may be loaded into memory at the same time. This capability provides backward compatibility in cases where an application was built with one

version of an assembly and does not work with follow-on versions of that assembly. This also virtually eliminates the possibility of back-leveling an assembly, a common problem with traditional Windows applications.

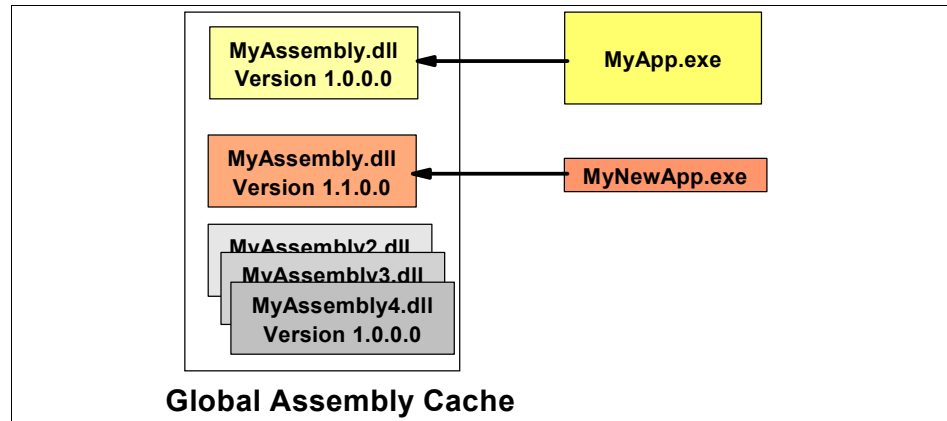


Figure 2-12 Object versioning within the .NET Global Assembly Cache

Assembly versioning is most transparent and effective when an assembly is strong named and installed into the Global Assembly Cache (GAC). The .NET Global Assembly Cache is a repository for commonly used assemblies. Unless specified by configuration and policy files, the .NET assembly loader loads generically specified assemblies from the Global Assembly Cache before searching and loaded same named assemblies from other locations. The Global Assembly Cache requires assemblies to be strong named and may contain multiple versions of the same assembly or same versioned assemblies with different culture information. See Figure 2-12 for an example of assembly versioning.

2.2 Development

In the Microsoft .NET environment, the development can be done using either a text editor like Notepad or the *Visual Studio .NET Integrated Development Environment (IDE)*. This section covers how to write, compile and run an application in your development environment.

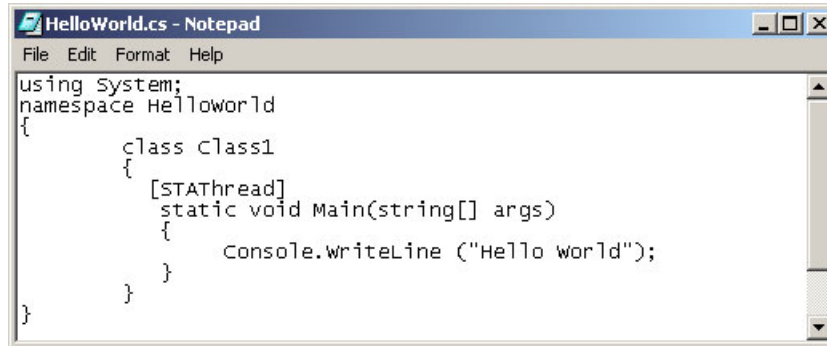
2.2.1 Writing a C# application using text editor

This is same as writing Java code in a text editor and then compiling using the command prompt. Let's take an example of writing a C# application in Notepad.

The steps are:

1. Write the code in the text editor of your preference.
 - a. Save the code with a .cs extension.
 - b. Compile and run the application.

The following screen capture is a very simple C# code written in Notepad.



```
using System;
namespace HelloWorld
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            Console.WriteLine ("Hello world");
        }
    }
}
```

Figure 2-13 C# code in Notepad

2. Compiling the code using a command prompt:

The C# compiler has different options to compile the source file, some of them listed in following table:

Table 2-3 C# Compiler options

Compiler Option	Result
csc HelloWorld.cs	Creates HelloWorld.exe file
csc /target:library HelloWorld.cs	Cerates a HelloWorld.dll file
csc /out:OtherName.exe HelloWorld.cs	Creates an OtherName.exe

To learn more about compiler options, run the **csc /?** command.

```

C:\WINNT\Microsoft.NET\Framework\v1.1.4322>csc /?
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

        Visual C# .NET Compiler Options

- OUTPUT FILES -
/out:<file>          Output file name (default: base name of file with main
                    class or first file)
/target:exe         Build a console executable (default) (Short form:
                    /t:exe)
/target:winexe      Build a Windows executable (Short form: /t:winexe)
/target:library     Build a library (Short form: /t:library)
/target:module      Build a module that can be added to another assembly
                    (Short form: /t:module)
/define:<symbol list> Define conditional compilation symbol(s) (Short form:
                    /d)
/doc:<file>          XML Documentation file to generate

- INPUT FILES -
/recurse:<wildcard> Include all files in the current directory and
                    subdirectories according to the wildcard specifications
/reference:<file list> Reference metadata from the specified assembly files
                    (Short form: /r)
/addmodule:<file list> Link the specified modules into this assembly

- RESOURCES -
/win32res:<file>     Specifies Win32 resource file (.res)

```

Figure 2-14 Available compiler options

3. Running the application from the command prompt:

To run an executable in Java environment, you have to pass the file name with a Java command. In the .NET environment, run an executable type name of the .exe file and press **Enter**. In the following snippet, HelloWorld.cs is first compiled using the **csc** command and then run by just typing the executable file name.

```

C:\ITS0>csc HelloWorld.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

C:\ITS0>HelloWorld
Hello World

C:\ITS0>

```

Figure 2-15 Compiling and running C# program

2.2.2 Microsoft Visual Studio .NET (IDE)

Developing applications in Integrated Development Environment is quite simple and fast and allows *Rapid Action Development (RAD)*. The Environment is integrated with version control software such as *Visual Source Safe (VSS)* for easy management of code. The editor in Integrated Development Environment has some common features what the WebSphere Studio has like IntelliSense (type-ahead) and Drag-Drop design.

The Visual Studio .NET has different editions like Professional, Enterprise Developer, Enterprise Architect. Each edition has its own features. In addition to these editions, special language specific editions are available such as Visual Basic.NET Standard Edition and Visual C# Standard Edition primarily for hobbyist, students and beginners.

When you open the Microsoft Visual Studio, you get the Start Page with three panes; the Projects pane has details about the history of the projects, the Online Resources pane to get the online samples and the My Profile pane to set the personalized settings.

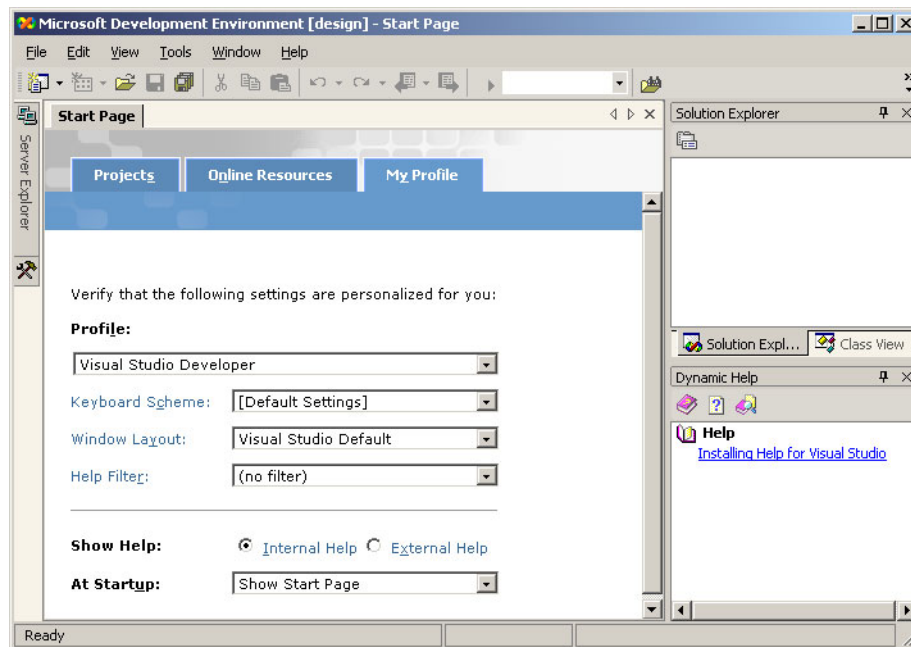


Figure 2-16 Start Page in Visual Studio .NET

The Integrated Development Environment has different windows such as the Tool window, Code window, Properties window and Solution Explorer window.

Except for the Code window, the windows can be hide using the *Auto Hide* feature. For example, in the following diagram, the Server Explorer and the Toolbar windows are hid using the *Auto Hide* feature.

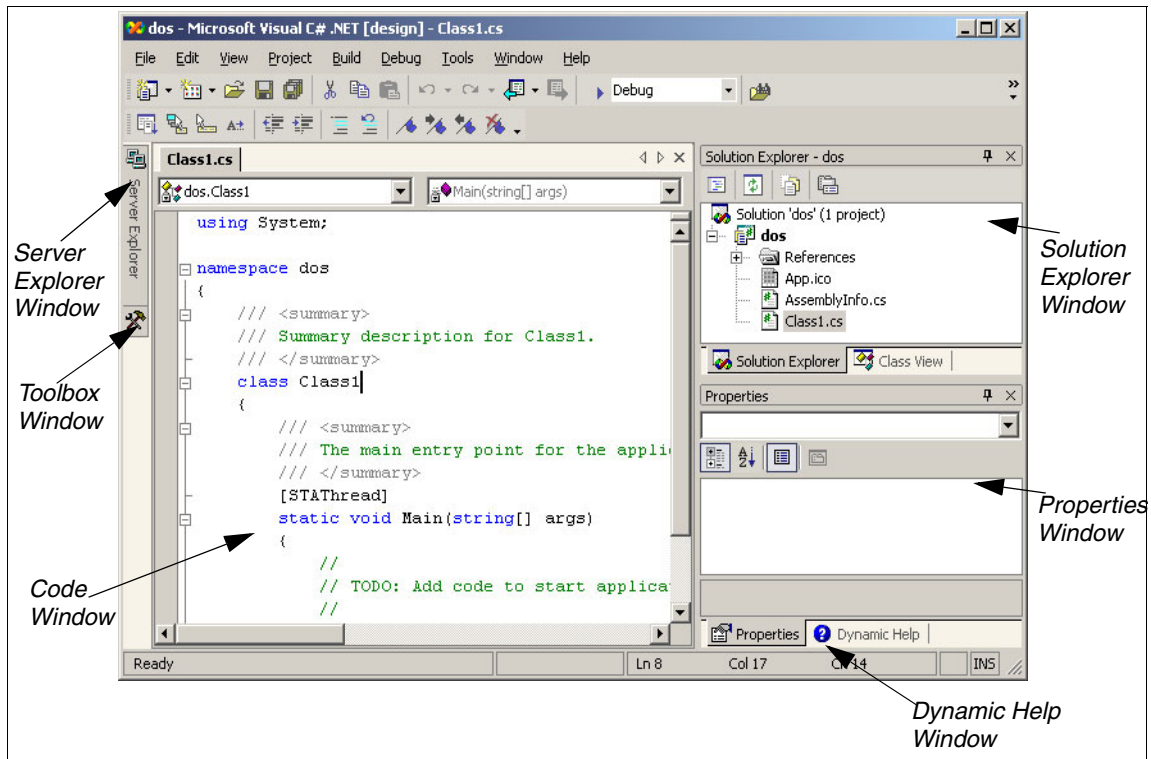


Figure 2-17 The Visual Studio Integrated Development Environment

To start a new project in Visual Studio, select **File -> New -> Project** from the main menu. The New Project window appears as shown next.

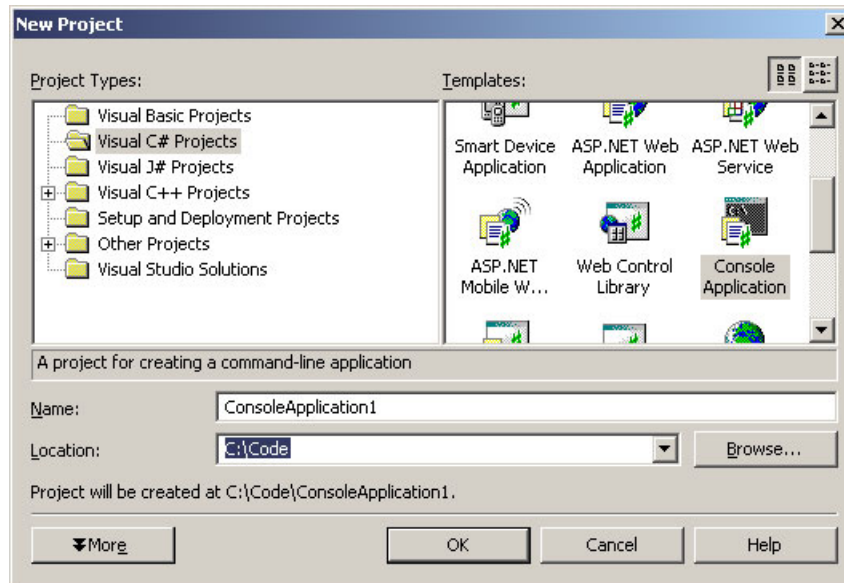


Figure 2-18 The New Project Window in Visual Studio .NET

In the Integrated Environment, various types of projects can be created by selecting the appropriate template. The user can add more than one project for development. This feature helps while debugging Class Libraries with another application.

Table 2-4 describes various templates available for projects.

Table 2-4 Project templates

Template	Description
Windows Application	Creates new Windows or Desktop project refers to traditional rich client applications.
Class Library	Creates new DLL Project or to crate components that typically encapsulate some business logic.
Windows Control Library	Creates new control project which can be used on windows application - as traditional ActiveX control.
Smart Device Application	To create new smart device project for example PDAs.

Template	Description
ASP.NET Web Application	To create new ASP.NET Web application project - dynamic and data driven browser based applications.
ASP.NET Web Service	To create new Web Service project.
ASP.NET Mobile Web Application	To create Mobile application project.
Web Control Library	Creates Web control project that can be used on Web form allowing code reuse and modularization.
Console Application	Creates console application project to run on DOS command prompt.
Windows Service	Creates windows service project that runs in background.
Empty Project	Creates an empty project or solution. You can avoid creation of default files and can insert your own files.
Empty Web Project	Creates an empty Web project.

To create an application, select the language for coding from Project Types and then select the application type from Templates.

To build the project, various options are available in the Build menu; to run the project, click the **Run** button on the toolbar.

2.2.3 Source code management

The code management in the .NET Framework can be implemented by using *Microsoft Visual Source Safe (VSS)*. The Microsoft Visual Source Safe is a repository to manage project files. The VSS is tightly integrated with Visual Studio and therefore has some advantages compared to other third party products.

The user can see the latest version of any file, make changes, and save a new version in the database. By doing *check-in* or *check-out*, the user can get or save the files, respectively.

2.3 Testing

Application testing and debugging involves various activities on an individual unit of deployment basis, as well as a cross-application and often inter-application basis with the goal of ensuring that each individual unit of software works, individually and collectively, as designed.

When we test whether a component, collection of components or even a collection of applications behaves as designed in terms of *what* it does, this is known as *functional testing*. It is also important to be able to thoroughly understand system performance and capacity based on predefined design parameters. This is known as *performance and load testing*.

2.3.1 Debugging and unit testing

Given that the developer's role is to produce *correct* code, it is to be hoped that the first person to test code is the one who wrote it. Functional testing that occurs during the development and initial debugging process primarily involves the IDE and often begins immediately after the compiler is through, rooting out the most obvious coding errors.

Visual Studio.NET provides the essential tools for the software developer. Applications are developed under Visual Studio.NET as Projects within a Solution file. Each Project will result in a separate assembly. Debugging options and features are, therefore, set at the project level. This allows the developer to control which debug features are enabled for each assembly involved in testing.

Debug settings

The behavior of the Visual Studio.NET debugger is customizable, allowing the user to control such things as warning levels, how breakpoints can be set, how data members and variables are displayed across the entire debugging IDE, creating a comfortable personalized environment for the developer.

Building a project for debugging: debug builds

The debugging capability is enabled by the creation of a "debug build," which creates, in addition to the executable code, the additional symbols and information required to allow the debug facility to associate execution code with source code.

Under Visual Studio.NET, the type of build created is controlled at the project level through the Configuration Manager dialog, which is accessed from the Project Properties page as depicted in Figure 2-19 on page 83.

The build configuration for each project is selectable, allowing the developer to build release versions of some components while continuing to debug others. Generic Debug and Release configurations are provided by default. Additional custom configurations can also be created for each project within a solution. These configurations allow the developer to refine the debugging environment further on an assembly by assembly basis.

Any assembly built from a .NET project built with the Debug configuration set can be fully debugged via Visual Studio.NET.

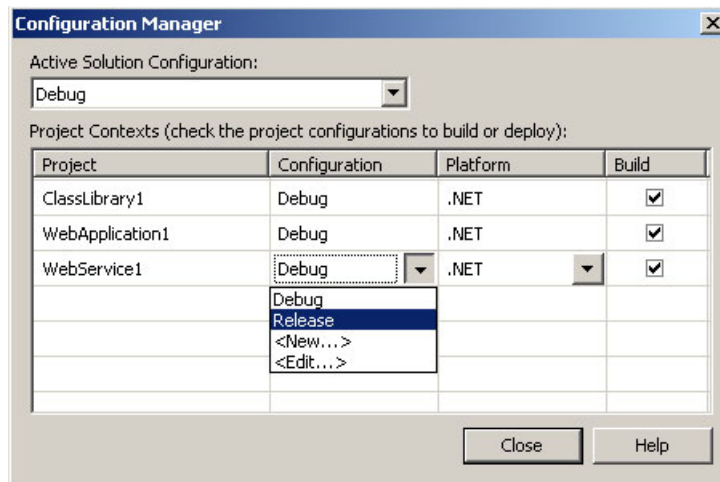


Figure 2-19 Configuration Manager

In addition to controlling these configuration build options from the IDE's visual environment, command line facilities are also provided to enable script-based control of builds for debugging.

Code animation

The Visual Studio.NET environment provides debugging animation support for a large majority of the code that can be developed within the .NET environment. This includes the following types of animation:

- ▶ Multi-language animation; ASP.NET, VB.NET, C#, C++, C.
- ▶ Support for managed (CLR) and unmanaged (binary) code and their interoperation via .NET Interop.
- ▶ Services, Server-side COM+ objects, DLLs and other unmanaged objects.
- ▶ The ability to animate multi-threaded code.
- ▶ Animation of Class libraries.

- ▶ Animation of .Net assemblies that take advantage of Remoting, both from the client and server perspectives.
- ▶ Animation of Web Services as they are invoked.
- ▶ Remote debugging of code running on other physical nodes.
- ▶ Animation of SQL Stored Procedures for SQLServer.

When you are animating code, Visual Studio.NET allows you to step through all execution regardless of threading or language concerns as long as the currently executing project has been built with a debug build. If a debug build is not available for the currently executing code, you can skip it or step through the disassembly of your release-level code, which shows the instructions created by your source.

Features available during animation include stopping execution, stepping through, over, and into code, running to cursor, and resetting the current line. In addition, you can view and modify variables, view registers and the memory space in which your process is operating. You can debug one or multiple processes simultaneously.

With symbols provided by Microsoft as part of the Platform and Framework SDKs, you can also step through framework and operating system code executed by your application.

The debugger allows you to decide how to handle exceptions during debugging and provides an edit and continue feature which allows you to edit your code in place during a debugging session and continue without rebuilding and starting over.

Just-in-time debugging

The bugs that escape unit testing are usually the ones that occur in special circumstances. These are often difficult to duplicate in a debugging environment. A unique feature of Visual Studio.NET is the ability to turn on *just-in-time debugging* for a given assembly.

With just-in-time debugging enabled, an exception within an assembly that was started and is running outside the IDE will cause a dialog box to appear, providing an opportunity to enter the debug mode with any debugging tool you specify. This allows a developer to begin animating the code and easily identify the location of the problem and the current execution state.

In Visual Studio.NET, just-in-time debugging across systems (remote debugging) was disabled due to security issues.

Tracing and debugging instrumentation

Tracing and debugging is an important cross-cutting consideration within any application.

.NET provides a set of classes that allow developers to set up their applications to make debugging and tracing information available to management tools at runtime. The process of using these classes to enable tracing and debugging functionality within applications is called *instrumenting*.

Tracing and debugging classes write their output to listeners. A listener is an object that receives trace output and persists it somewhere (such as a log, a file, a database or a screen).

Once the code is instrumented, you decide whether the tracing and debugging statements will actually be included in the assembly by setting the Trace and Debug conditional attributes in your project build settings. This prevents release versions of the application from writing to debug listeners. Further control of trace output is also available by including a Trace Switch object in your source code to set severity levels and control which trace statements will produce output.

Instrumented applications can make information available about a wide variety of execution and behavior parameters that can be used to better understand execution.

Other debugging tools

Microsoft also provides some smaller utilities which are useful for debugging various parts of a .NET application. Many of these utilities have been around for some time and have been enhanced for the new environment. A brief list follows:

- ▶ **ISAPI Web Debug Tool:** Allows debugging of ISAPI filters commonly used in Web applications.
- ▶ **Spy++:** Tracks controls, windows, threads and processes during execution to aid in debugging.
- ▶ **Visual Studio command prompt:** Most things that can be done in Visual Studio.NET can also be done from the command line. This simply opens a command line session with all of the proper environment variables set for Visual Studio.NET command line operations.

2.3.2 Performance and load testing

Any application's functionality is only as useful as its ability to perform with adequate speed for the planned number of users.

This section discusses the performance tools, techniques and practices commonly used for .NET applications.

Performance Monitor and counters

The Microsoft Operating System provides a tool for selectively measuring and/or recording performance on any application, process or thread running on a local or remote system. This tool is called Performance Monitor. Performance Monitor works by listening to *performance counters* that are installed with the Common Language Runtime and .NET Framework Software Developer's Kit.

A performance counter is an object that reports performance information. The operating system and products built upon it contain performance counters that are automatically inherited by an application. These existing performance counters cover a large array of performance information such as memory, handles, threads, locking, loading of code, networking, and many more. This means that Performance Monitor can be used to get a very detailed view of the internal performance of any .NET application regardless of where it is running.

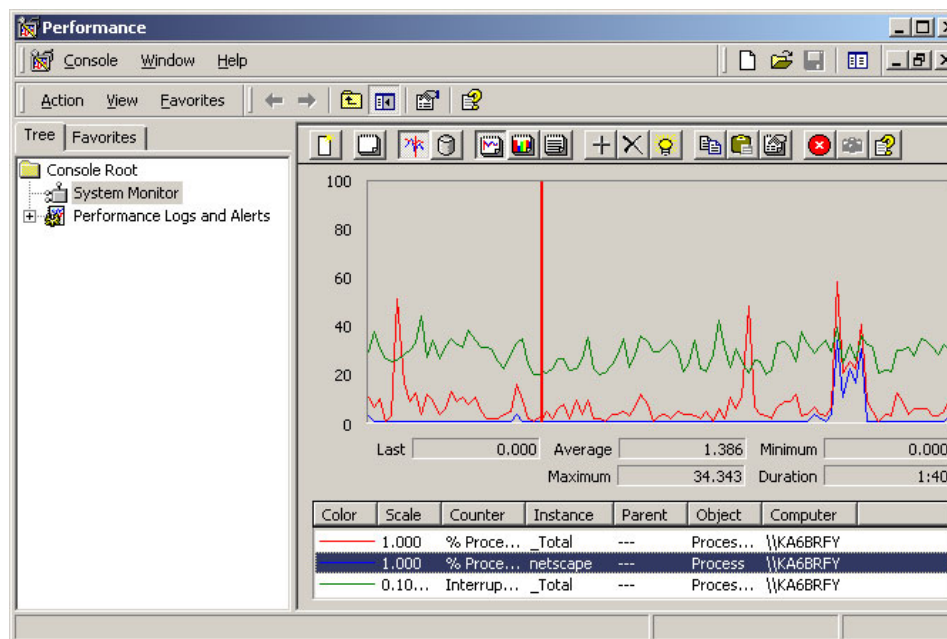


Figure 2-20 Performance Monitor

Classes provided in the System.Diagnostics namespace within .NET can be used to instrument your application with custom performance counters or to consume performance counter information programmatically in the same way Performance Monitor does.

By instrumenting your application using these classes, you can implement useful counters in specific areas of your application where performance might be an issue and the existing counters do not supply this information, for example, when counting the number of people who use a specific feature of your application on an ongoing basis.

Once implemented, these counters can then be viewed by anything that can consume these counters, such as Performance Monitor or a customer listener you create within your application.

The Microsoft Application Center Test

Included with Microsoft Visual Studio.NET for Enterprise Architects is a tool for simulating several users simultaneously accessing various parts of your Web application from browsers. This tool is called Application Center Test (ACT).

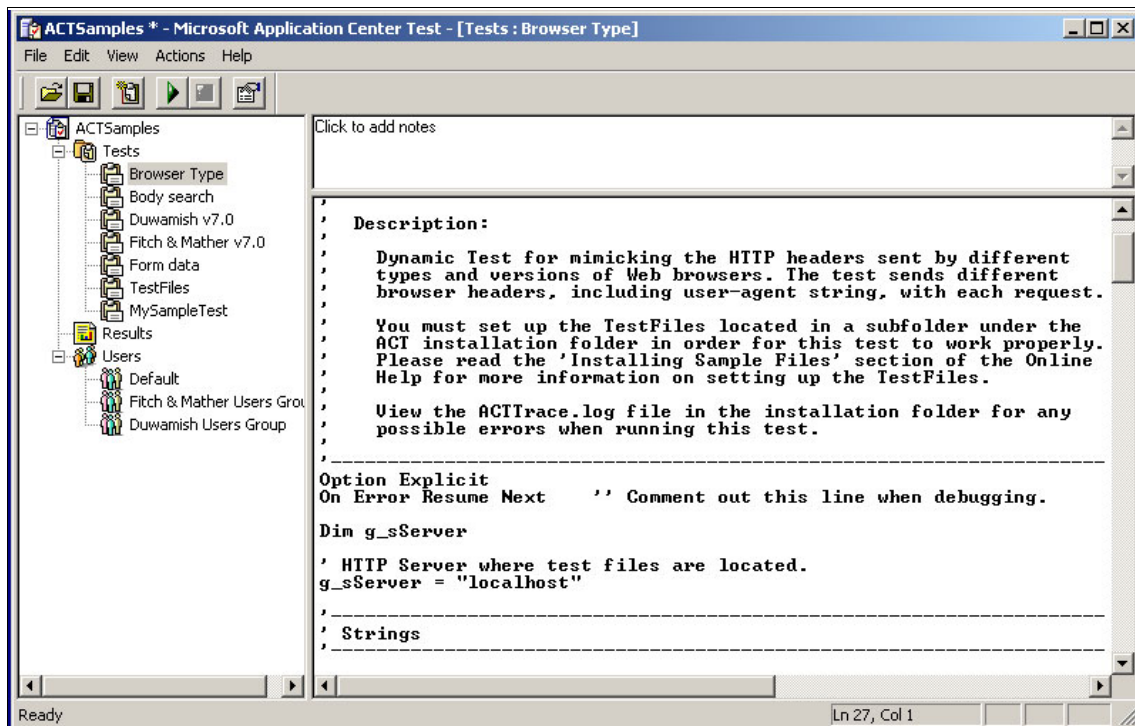


Figure 2-21 Application Center Test multi-browser-type sample

ACT allows you to observe various parts of the application while it run with a simulated load. You can create tests via several methods including recording real-time browser sessions. It can simulate multiple groups of users simultaneously accessing a Web site using different browsers and different

versions of browsers. Test sessions are scripted using either VBScript or JScript and the tool comes with several sophisticated examples.

There is also a variety of third-party tools available with rich testing facilities.

2.4 Deployment

The deployment in .NET is quite easy and different from the traditional model; it has no issues like DLL or COM registration and versioning. The deployment in .NET consists of packaging and distribution and can be done in various ways .

- By copying files

This is the simplest way of moving a .NET application from one location to another location. This can be done by using either the **Copy Project** command available on the Project menu or by using the **XCOPY** DOS command.

This method has limitations. Copying files does not register or verify the location of assemblies, and for Web projects, it does not automatically configure IIS directory settings.

- By using Setup and Deployment programs

A Setup and Deployment program is the professional way of deploying the applications in a work environment. The advantages of using Setup and Deployment projects are as follows:

- You can avoid overwriting of files that could cause other applications to break.
- Registering and deploying of COM components or assemblies can be implemented.
- The program shortcut can be added in to Windows' Startup menu or onto the Windows desktop.
- Interaction with the user, for example storing user information, licensing and so on.

The Visual Studio .NET has several ways of creating a Setup and Deployment project. Moreover, the third party Setup and Deployment programs can be used for deploying the .NET applications.

The following snippet shows various Setup and Deployment projects available in Visual Studio .NET.

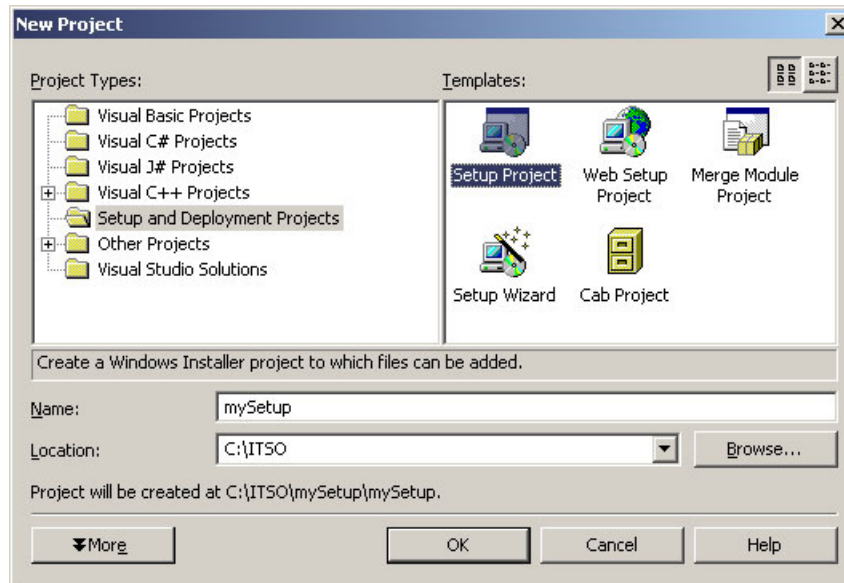


Figure 2-22 Setup and Deployment Projects in Visual Studio .NET

The following list describes various project types available in Visual Studio .NET:

Table 2-5 Project types available in Visual Studio .NET

Project Type	Purpose
Merge Module Project	The Merge Module Project allows you to share setup code between Windows Installers and avoids versioning problems. This creates a merge module (.msm file), a single package that contains all files, resources, registry entries, and setup logic necessary to install a component.
Setup Project	The Setup Project builds an installer for a Windows-based application in order to distribute an application. The resulting Windows Installer (.msi) file contains the application, any dependent files, information about the application such as registry entries, and instructions for installation. The Setup project installs the files into the file system of a target computer.

Project Type	Purpose
Web Setup Project	The Web Setup Project builds an installer for a Web application. The setup installs files into a virtual directory of a Web server.
Cab Project	The Cab project builds a cabinet file for downloading to a Web browser.

2.5 Runtime

The runtime environment for .NET applications is the Windows operating system itself.

Since .NET managed applications are not binary code suitable for direct execution on a given machine, they must run within the Common Language Runtime.

The .NET Framework provides the Common Language Runtime and base libraries that .NET applications use to access operating system features such as the Windows.Forms namespace. New versions of Windows such as XP and Windows Server 2003 come with all of the necessary products and features to run .NET applications. However, older operating systems such as Windows 2000 require additional runtime components.

At the time of this writing, the current version of the .NET Framework is V1.1. A redistributable version of this framework is downloadable from Microsoft's Web site.

The redistributable runtime can also be packaged with a Microsoft .NET Application installation created under Visual Studio.NET in order to allow it to be installed, if necessary, along with a .NET application.

For more information on the runtime for .NET, please see the following Web location:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vstchdeployingvsusingactivedirectory.asp

2.6 Administration

Distributed and Web-based applications deployed on even a medium scale can provide a significant administrative burden. The various products that provide the

features underlying .NET, such as the operating system, Enterprise Services, COM, Internet Information Services, SQL Server and other products, all have their individual management capabilities and tools. These tools are available remotely in nearly every case and are conveniently collected together within the Control Panel under Administrative Tools, as shown in Figure 2-23.

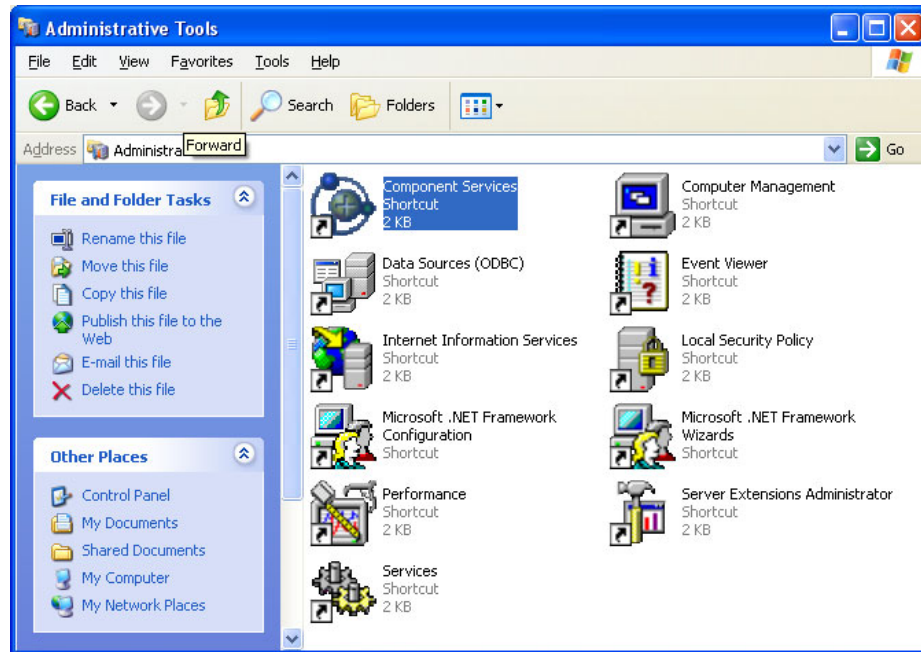


Figure 2-23 Administrative Tools

It is beyond the scope of this book to address each of these. However, we will briefly discuss a few key additional management components provided by Microsoft.

- *Microsoft Operations Manager* provides event-driven management for the Windows platform and applications that implement Windows Management Instrumentation (WMI). It allows consumption and utilization of information provided through WMI and provides a platform on which to generate alerts or take action based on specific events. For more information on Microsoft Operations Manager, see the following Web site:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mom/sdk/html/momabout_71g1.asp

- *Windows Management Instrumentation (WMI)* provides a common instrumentation method and collection model for applications and underlying technology. By implementing Windows Management Instrumentation in your applications as discussed earlier in this chapter in “Tracing and debugging

instrumentation” on page 85, and creating management agents, you can incorporate application specific performance and execution information into your management framework. In addition, the Windows Management Framework contains many agents for managing specific protocols, products and features of the operating system.

- ▶ *Systems Management Server (SMS)* provides management capabilities in the areas of operating system configuration management. SMS can be used to manage a large number of servers or workstations from the perspective of implementing operating system and software updates and managing configuration information on each device. For more information on SMS, please see the following Web location:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/momsdk/html/momabout_9fvx.asp

- ▶ The main administrative console for Internet Information Services is the *Internet Services Manager* from the Control Panel Administrative Tools. This tool can administer and manage the IIS nodes and Web sites. It works for both local and remote nodes and can manage multiple nodes and Web sites.
- ▶ The *Microsoft .NET Framework V1.1 Configuration* and the *Microsoft .NET Framework V1.1 Wizards* tools can help to configure, administer and manage the .NET Framework. The wizards include:
 - Adjust .NET Security
 - Trust an assembly
 - Fix an Application
- ▶ *Active Directory* provides a number of critical functions for the Windows operating system, including providing a common directory for distributed components, acting as a central authentication provider, managing users, groups and individual machines. Active Directory provides a Group Policy feature that enables administrators to define a host of policies for groups of users and/or nodes. A technology known as *IntelliMirror* uses Group Policy to enable software distribution and configuration management capability for groups of machines and people.

For more information on the use of Active Directory in the administration role, please refer to the following location:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vstchdeployingvsusingactivedirectory.asp



An architectural model for coexistent applications

Fundamental to a successful solution is an accurate understanding of a problem, and of that problem's domain (context). In order to evolve the understanding of a problem, and in order to convey this understanding to others, it is important to have a good model.

In this chapter, we will identify a basic architectural model which can be used to represent both WebSphere applications and .NET applications. We will then discuss how we can extend this model to deliver a composite view of these applications as coexisting applications (delivering runtime functionality composed from elements of both applications).

We will not attempt to advocate coexistence across heterogeneous implementation technologies as a good technical design principle, simply because it is not one. Our assumption is that you have already identified and justified that you have a special (extraordinary) business need to deliver such an implementation.

3.1 Coexisting heterogeneous technologies

Figure 3-1 illustrates the fundamental challenge we are addressing in this book. That is, how can we combine functionality implemented in both WebSphere technologies and .NET technologies to fulfill a single business request?

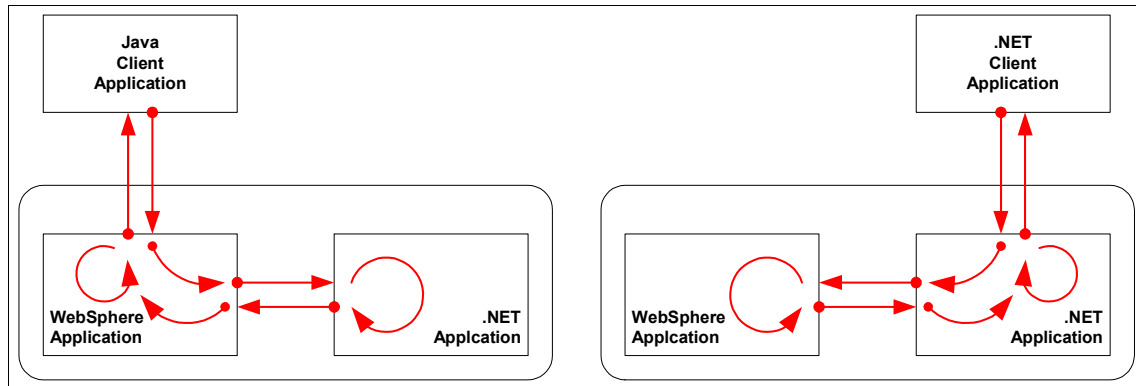


Figure 3-1 The challenge

While we will not recommend that you deliberately set out to design new applications using a combination of dissimilar technology sets, we can envisage scenarios where you may need to reuse an existing implementation across technology boundaries.

Some architectural models deliberately set out to model an enterprise as a set of cooperating services which are loosely coupled in an implementation technology neutral way. Both message-oriented architectures and service oriented architectures are examples of this approach. Sometimes, it is useful to think of the relationship between these loosely coupled implementations as a *delegation model*, where one component calls on the other to do some work on its behalf.

However, there are some situations where you just need to deliver a point in time solution, without re-architecting the whole enterprise. Also, there are some situations when you need a tighter binding between implementations that a service-oriented or a message-oriented approach cannot easily deliver. In this situation, it may be appropriate to think of these tightly coupled implementations as a *conjoined model*, where both components wrestle with the work until a result is obtained.

As with many software engineering problems, analysis and modeling are the foundation to understanding the extent of a task, and to identifying an appropriate solution. For our challenge of coexistence and integration between applications deployed in WebSphere and applications deployed in the .NET

Framework, we need to identify an architectural model that is generic enough to be suitable for illustrating any application from either technology set, and flexible enough to allow us to explore how to combine these two hypothetical applications in an appropriate way.

3.1.1 Layered application model

The classic application model presented in Figure 3-2 is probably familiar to most readers. The objective of this model is to logically partition an application into a stack of (conventionally) five layers. Each layer has separate technical concerns, and each is used as a logical container for placement and classification of an application's components.

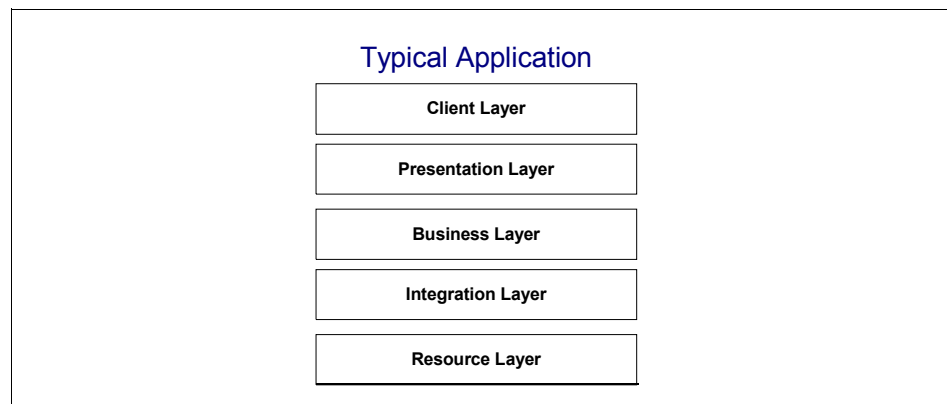


Figure 3-2 A simple layered application model

The Business layer (core)

The business layer is a logical container for application components that are the *core business functionality* of an application. In a use case centric design, the use cases are normally fulfilled by the code in this layer, or delegated to other resources via the integration layer.

The Resource layer (dependencies)

The Resource layer is a logical container for *dependencies* that business layer components rely upon in order to fulfill use cases. These resources could be persistent data stores, other systems, other services or other applications, in fact, anything that is external to the implementation scope of the modelled (subject) application.

The Client layer (invoker)

The Client layer is a logical container for the placement of consumers of use cases, or initiators of use cases, which are fulfilled by application components in the business layer. Examples of clients include fat GUI applications, Web Browsers, and other systems.

The Presentation layer (core glue)

The Presentation layer is a logical container for application components responsible for integrating components or systems in the Client layer, with components in the business layer. Presentation components are conventionally responsible for:

- ▶ Authenticating and authorizing the requester, validating the request, and routing the request to the appropriate business layer component.
- ▶ Rendering the response from the business layer component into a format that is appropriate for the requester.

The Integration layer (resource glue)

The integration layer is the logical container for application components that are responsible for integrating components in the business layer with components in the Resource layer. Integration layer components are conventionally responsible for:

- ▶ Authenticating and authorizing the requester, validating the request, and routing the request to the appropriate Resource layer component.
- ▶ Rendering the response from the Resource layer component into a format that is appropriate for the requester.

This seemingly trivial model can be used to impose some real structure on an application. It can be used as a starting point for modeling almost any new application, or for modeling almost any existing application. The simplicity and abstractness of this model is its most powerful feature. It allows us to take an application and classify every application component by the deliberate act of placing it into a layer.

One interesting feature of this layered model is that it does not necessarily mandate any implementation technology. Consider Figure 3-3 on page 97, which illustrates two layered application models in *parallel coexistence*; one model represents an application deployed in WebSphere, and the other represents an application deployed in the .NET Framework.

Figure 3-3 on page 97 models parallel coexistence, but the subject of this book is how to get these two applications to combine functionality to fulfill a use case. This indicates a need for interoperation between the two applications. If we consider that the responsibility of the integration layer is to bridge between any

abstracted resource and the business layer, we could, if we wished, remodel the target application as a resource, and place an interoperation integration layer between the calling application and the target application. This is illustrated in Figure 3-4 and Figure 3-5 on page 98 from the perspective of a WebSphere calling application and from the perspective of a .NET calling application.

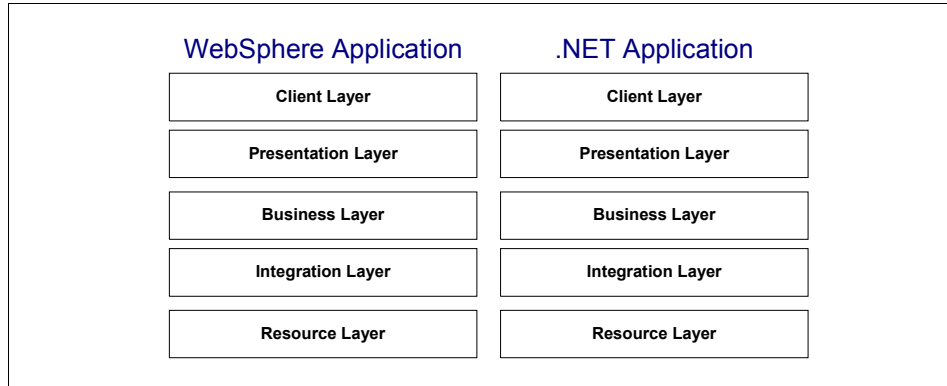


Figure 3-3 Modeling coexistent applications with layers

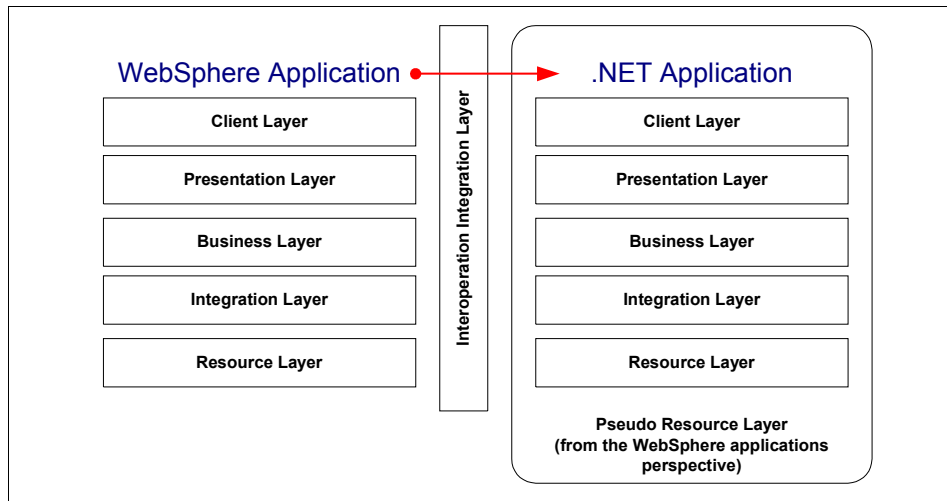


Figure 3-4 Interoperation layers (WebSphere callers perspective)

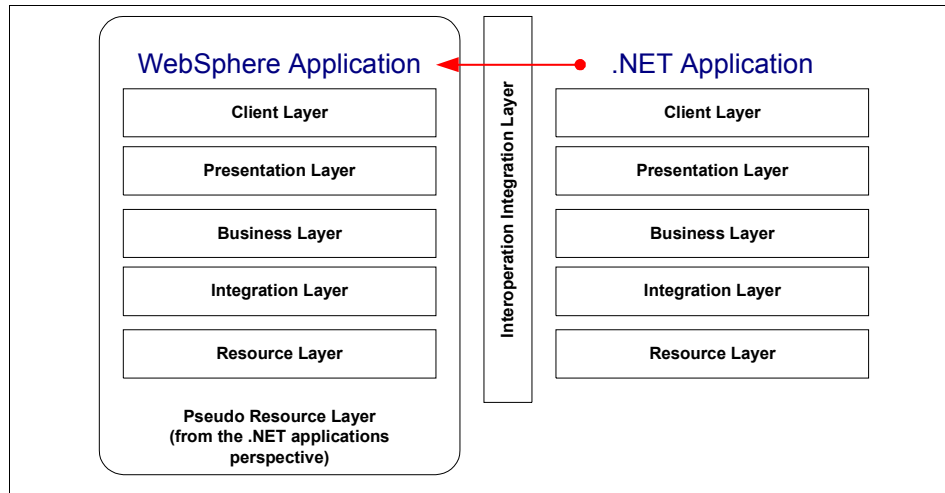


Figure 3-5 Interoperation layers (.NET callers perspective)

3.1.2 Concentric layered application model

The layered application model presented in Figure 3-2 on page 95 is a good technique for illustrating logical application structures and the relationships between two parallel applications. However, when we need to illustrate the relationship between more than two applications then this model becomes a little cumbersome.

Figure 3-6 on page 99 presents a variation of the layered application model, where layers are presented as concentric circles, and each logical application can potentially be split into multiple separate client applications, multiple separate server applications and multiple separate resources, with separately identified interoperation layers. This *concentric layered* modeling technique can be used to illustrate both problem context and a logical layered structure for *multiple coexisting applications*.

Once the relationship between multiple coexisting applications has been established and modelled, each interoperation integration layer (for instance between WebSphere Application #1 and .NET Application #1 in Figure 3-6 on page 99) can then be modeled separately in more detail.

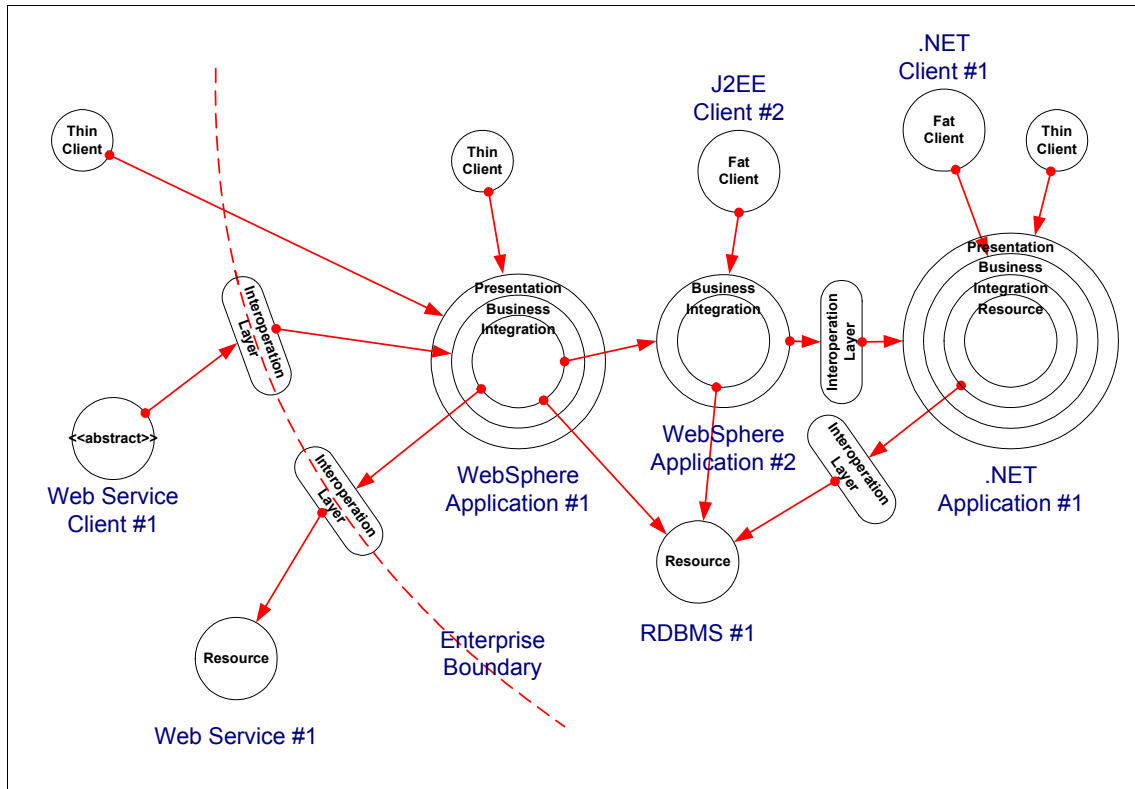


Figure 3-6 An example using a concentric layered model it illustrate more complex relationships.

Like the *flat layered* model, this *concentric layered* model is a logical model. No middleware is represented on the model. It illustrates relationships and separations of concerns. Neither concrete solution or concrete deployment schemes are implied.

3.1.3 Bridging layers and address spaces

If we extend Figure 3-4 on page 97 to include some common implementation technologies to both the WebSphere application layers and the .NET application layers, we get the situation illustrated in Figure 3-7 on page 100.

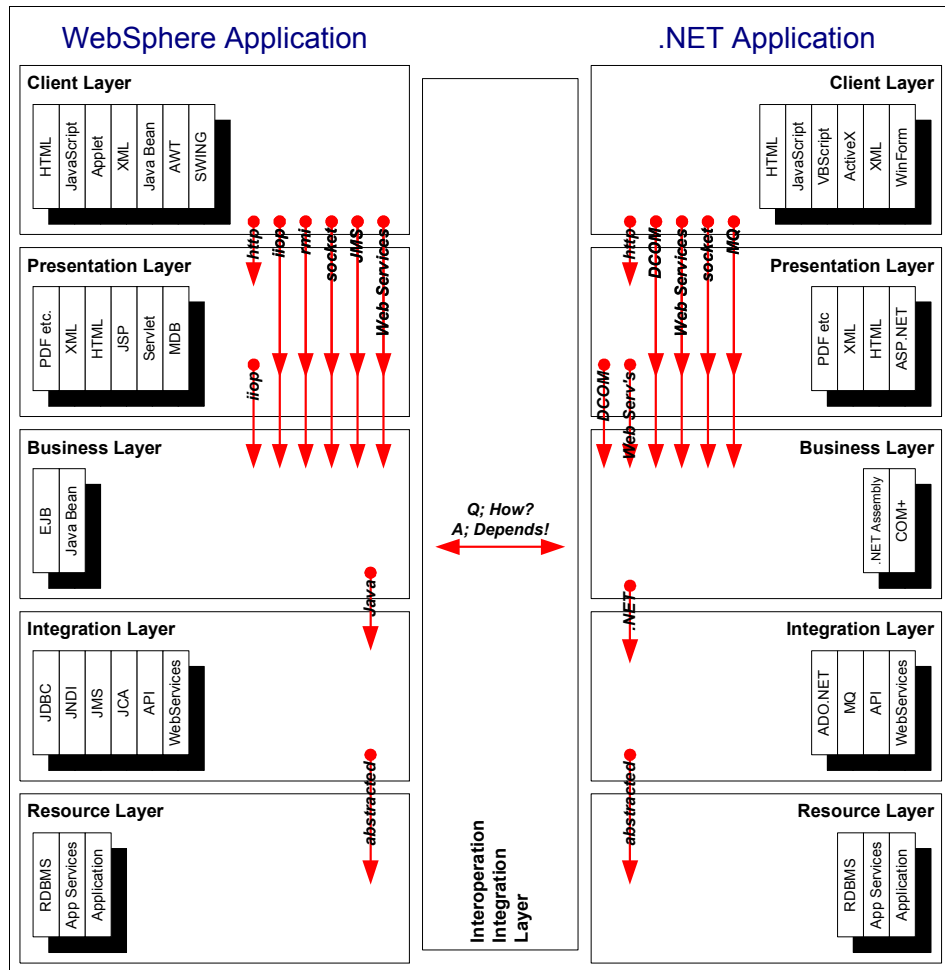


Figure 3-7 Typical layer technology mappings

Figure 3-7 also identifies some of the mechanisms that can be used for integration between application layers within a each technology set.

The objective of this diagram is not to enumerate all of the possible technologies that can be used to span the layers within each technology set, but to identify that in order to communicate across a layer, or to communicate from one component's address space to another component's address space, each side of the communication must share a common understanding of transport, protocol, state representation, and call context representation. The same applies to communication between our hypothetical WebSphere application and our hypothetical .NET application. It can be considered as just another application

layer. How we do this will depend on the nature of the required by the components in question, the dynamics of the required interaction between these components and the identification of these commonalities.

We will pursue the classification of these inter-component interactions in Chapter 4, “Technical coexistence scenarios” on page 109.

3.1.4 Interoperation layer abstraction

We can take the idea of modeling the target application as a *pseudo Resource layer*, with an *interoperation integration layer*, and apply some additional solution neutral principles to our generic coexistence architectural model.

A fundamental principle that we will advocate throughout this book is that neither the client code or the service code should be exposed to the fact that they are integrating to a dissimilar technology. This implies that they should be defended from any of the details of the technology chosen for a given technical solution. To deliver this separation of concerns between application deployment units and interoperation deployment units, we propose the solution pattern illustrated in Figure 3-8 on page 102 and Figure 3-9 on page 102.

Figure 3-8 on page 102 illustrates a Java client code deployed in WebSphere, integrating with a service implementation deployed in the .NET Framework (for the purpose of this discussion, we do not care which application layer our client code and service code implementations reside in; this subject will be addressed in Chapter 4, “Technical coexistence scenarios” on page 109). The client code binds to a Java implemented service proxy, which represents the iCalculator service to the client code (the iCalculator interface is used as an example throughout this book, and is intended to be simple to understand, rather than to be representative of a realistic interoperation scenario). The service code is bound to, and invoked by, the service stub. The details of service endpoint discovery, communication transport, communication protocol, data type mapping, remote call mechanisms and context sharing are the concern of the proxy and stub, and are not exposed in the iCalculator interface. As such, the fundamental principle of separation of concerns is adhered to.

Figure 3-9 on page 102 is similar to Figure 3-8 on page 102, but from the perspective of client code deployed in the .NET Framework invoking service code deployed in WebSphere.

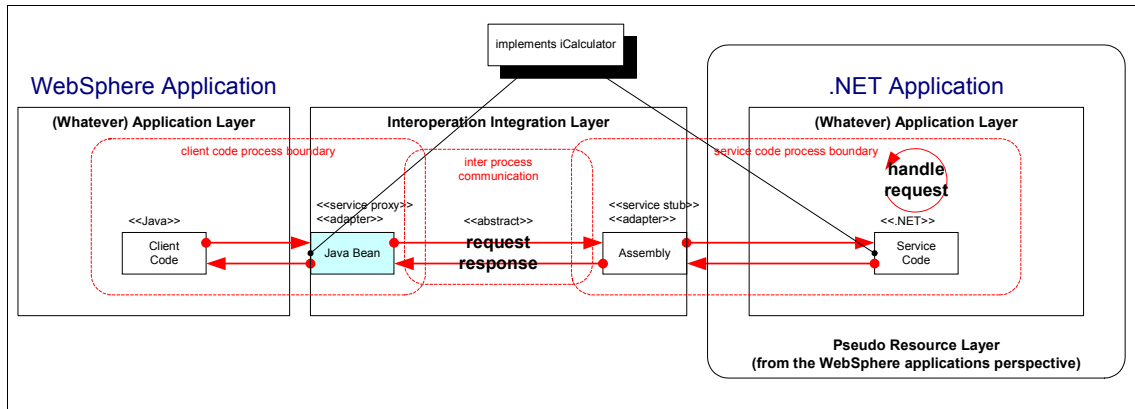


Figure 3-8 Abstraction of integration technology (WebSphere client code perspective)

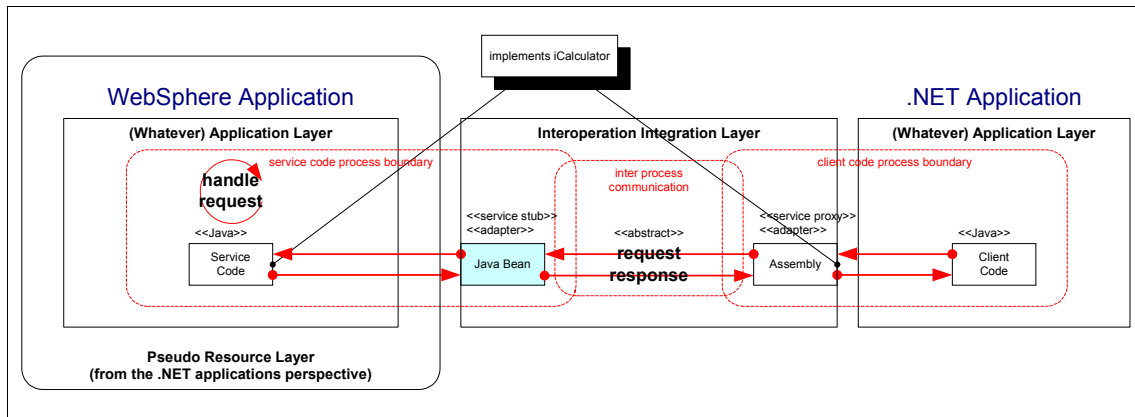


Figure 3-9 Abstraction of integration technology (.NET client code perspective)

Ideally, the language we use to define the semantics of the interface `iCalculator` should be construction language neutral. Consider Figure 3-10 on page 103; it illustrates an abstract *meet in the middle* interface from which the proxy's *top interface* and the service's *bottom interface* are derived as language specific bindings. The approach of defining a *middle interface* encourages the design of versatile construction language neutral interfaces. Figure 3-10 on page 103 suggests IDL or WSDL as a potential formal interface specification syntax.

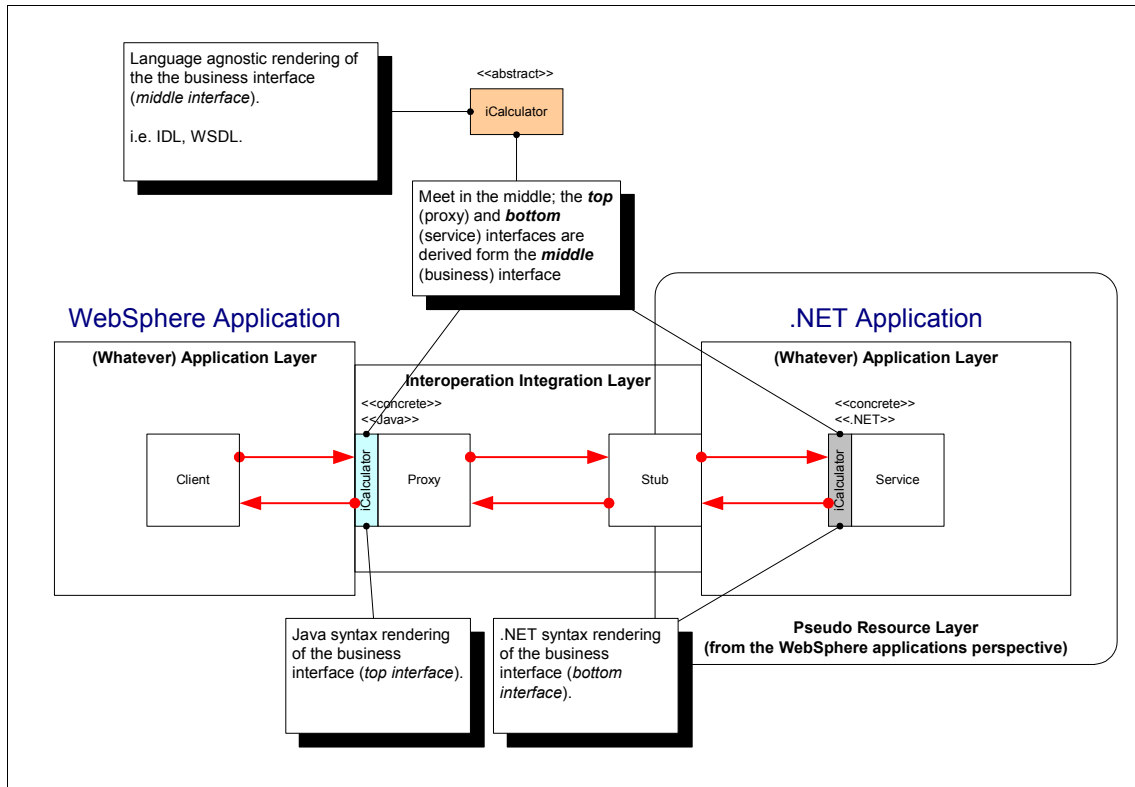


Figure 3-10 Technology neutral middle interfaces

Figure 3-10 represents an ideal situation. In reality, if you have an existing service interface, you could be forced into a *bottom up* scenario (Figure 3-11 on page 104). Or, if you have an existing client interface, you may be forced into a *top down* scenario (Figure 3-12 on page 104). Regardless of whether we are considering a *meet in the middle*, *top down*, or *bottom up* scenario, the fundamental principle here is that our primary definition of `iCalculator` is an implementation neutral business interface, not an implementation interface.

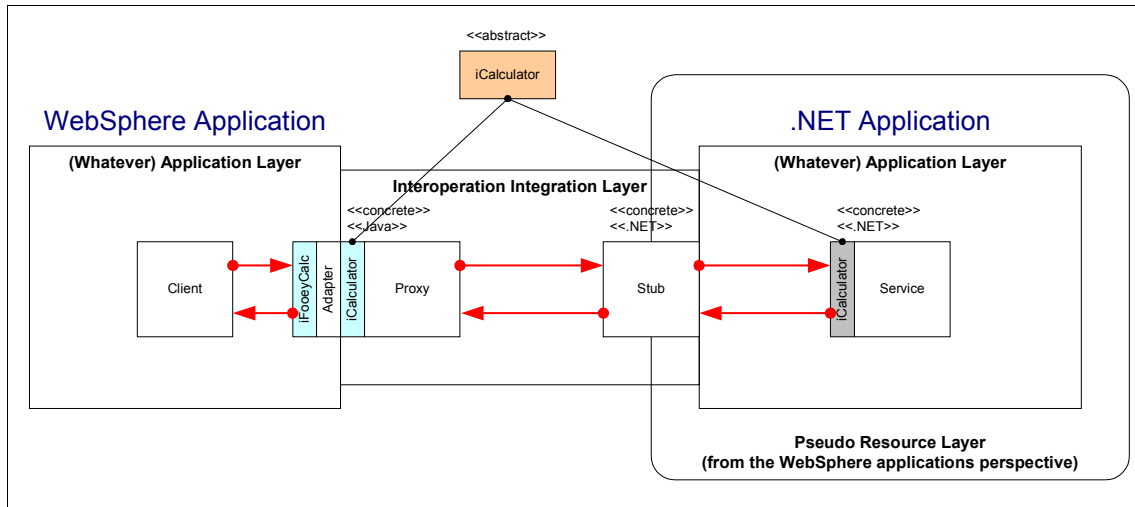


Figure 3-11 Bottom up interface scenario with interface adapter

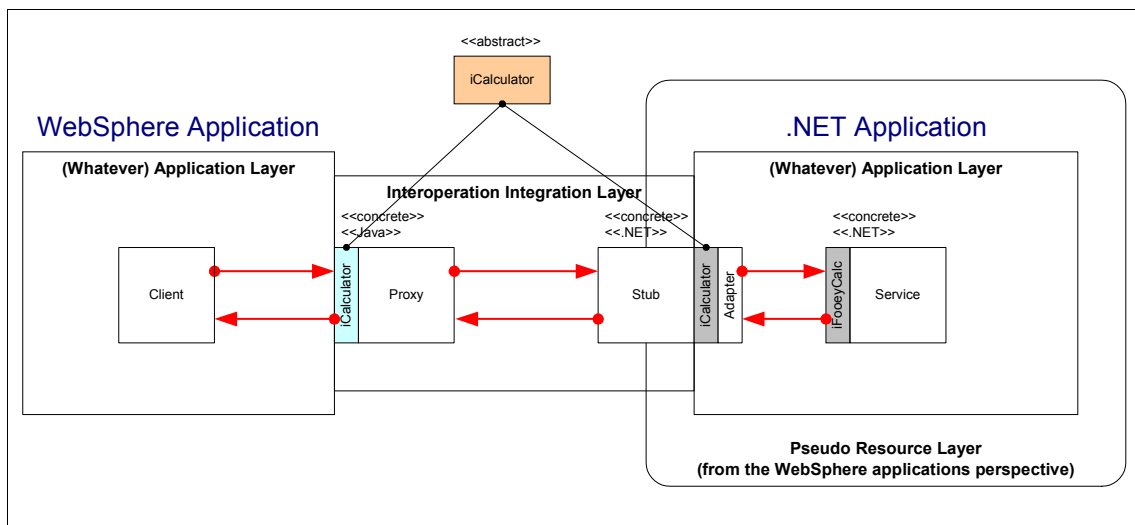


Figure 3-12 Top down interface scenario with interface adapter.

Adherence to this pattern defends the client code and the service code from possible future integration technology changes, and is an architectural enabler for managed phased migration from, for example:

- .NET Framework business code to Java business code with enhanced qualities service in WebSphere.
- Migration from a tactical integration solution today to a strategic integration solution in the future.

Figure 3-13 illustrates the service illustrated in Figure 3-10 on page 103 migrated from a .NET assembly deployed in the .NET Framework, to a local EJB deployed in WebSphere without modification to the client code (this implies that an abstract binding factory was used by the client code to resolve the concrete instance of `iCalculator` to use at runtime).

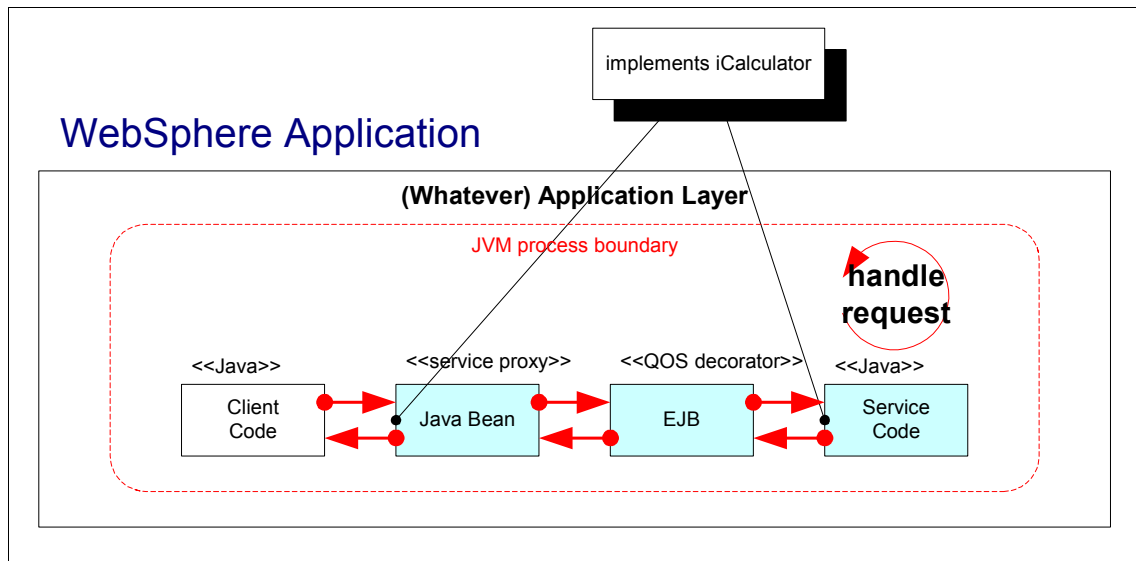


Figure 3-13 Client code defended migration from .NET to enhanced quality of service in WebSphere

3.1.5 Summary

In this chapter, we have presented some advice on the subject of modeling the problem domain for coexisting applications deployed in WebSphere and applications deployed in the .NET Framework.

We have suggested that modeling coexisting applications using a combined logical layered model, which spans multiple applications, can help us distill the problem into a simple unified cross-technology architectural model. We can apply this layered model regardless of whether or not the applications were originally modeled this way.

The value of building these models should be a good understanding of the technical problem context, and an identification of each of the interoperability points between coexisting applications.

We have indicated that we can, if we wish, model the target application as a resource to the calling application. We can place an interoperability integration layer between a calling application and the target application (target resource). We then have a clean abstraction within which to place the technical solution for the coexistence integration. Also, we have suggested that within the interoperability integration layer, technology neutral business interface definitions may be beneficial in delivering a risk managed interoperability and migration strategy.



Part 2

Scenarios



Technical coexistence scenarios

In this chapter, we expand on the architectural models introduced in Chapter 3, “An architectural model for coexistent applications” on page 93.

We will identify several potential technical scenarios for coexistence via point-to-point integration between applications deployed in the IBM WebSphere Application Server and applications deployed in the Microsoft .NET Framework.

We will also identify some candidate technical solutions for these scenarios, which we will consider in more detail in subsequent chapters.

4.1 Introduction

Ultimately, the choice of technical solution you make for your given WebSphere and .NET application coexistence challenge should be based on a good understanding of the specific problem you are addressing. This chapter is intended to give you guidance in identifying the characteristics of your given coexistence problem, identify some interaction scenarios, help you classify your coexistence problem with respect to these scenarios and interaction characteristics, and identify some candidate technical solutions for consideration.

With this in objective mind, we have divided this chapter into three sections:

- ▶ In 4.2, “Fundamental interaction classifications” on page 111, we will consider the potential interaction styles and dynamics between any hypothetical WebSphere application component and any hypothetical .NET application component. The objective of this section is to help you classify the interactions you need to resolve with your chosen technical solution(s).
- ▶ In 4.3, “Layer interaction classifications” on page 132, we will consider the potential integration scenarios between coexistent WebSphere and .NET applications. These scenarios will be limited in scope to point-to-point scenarios (between only two applications). We will also present some candidate solution models for each of these scenarios. These candidate solution models will abstract the integration technology, which should make them applicable to many technical integration solutions.
- ▶ In 4.4, “Technical solution mapping” on page 202, we will map candidate technical solutions to the scenario and the interaction classifications presented in the previous sections of this chapter (4.2, “Fundamental interaction classifications” on page 111 and 4.3, “Layer interaction classifications” on page 132). Some of the technical solutions presented in this section will provide tight coupling: some can provide loose coupling. You will need to choose which of these (if any) suit your needs. Remember that the short term tactical choice *may* be different from the long term strategic choice.

This approach may appear somewhat diagnostic. However, it is in fact intended to help you decompose the problem into tangible or cogitable pieces, identify the particular case you have, consider what an appropriate technical solution should deliver to your given problem, then normalize to one of the identified technical solutions.

This chapter will not attempt to advocate coexistence across heterogeneous implementation technologies as a good fundamental design principle, since it is not one. Our assumption is that you have already identified and justified that you have a special (extraordinary) business need to deliver such an implementation.

4.2 Fundamental interaction classifications

Before we identify candidate technical solutions to support runtime coexistence and interaction of application artifacts deployed in IBM WebSphere and application artifacts deployed in the Microsoft .NET Framework, we need to identify the potential interaction characteristics between any two deployed code units. A clear understanding of these interactions will help us classify our given integration challenge and help us make more informed decisions when we are identifying the appropriate technical solution.

Let's start by enumerating the common potential interaction dynamics between any two pieces of code. Fundamentally, these are:

- ▶ Stateful synchronous interaction
- ▶ Stateful asynchronous interaction
- ▶ Stateless synchronous interaction
- ▶ Stateless asynchronous interaction

Each of these interaction dynamics can have one of these interface styles:

- ▶ RPC interface style
- ▶ Document interface style

Each of these interface styles can have one of these interface argument paradigms:

- ▶ Argument by value paradigm
- ▶ Argument by reference paradigm

This gives us sixteen basic potential interaction classifications:

1. Stateful synchronous RPC by value
2. Stateful synchronous RPC by reference
3. Stateful synchronous document by value
4. Stateful synchronous document by reference
5. Stateful asynchronous RPC by value
6. Stateful asynchronous RPC by reference
7. Stateful asynchronous document by value
8. Stateful asynchronous document by reference
9. Stateless synchronous RPC by value
10. Stateless synchronous RPC by reference

- 11.Stateful synchronous document by value
- 12.Stateful synchronous document by reference
- 13.Stateful asynchronous RPC by value
- 14.Stateful asynchronous RPC by reference
- 15.Stateful asynchronous document by value
- 16.Stateful asynchronous document by reference

Note: It is possible to extend these classifications further by considering hybrid interfaces that combine both pass by value, pass by reference, RPC style and Document style. These hybrid classifications probably add nothing to the process of identifying new interaction characteristics. However, when you are choosing a technical solution, you need to ensure that the solution is capable of meeting these hybrid interfaces if that is a requirement for your given problem.

These fundamental interaction classifications can apply between any two pieces of code, irrespective of the implementation technology used and of whether the two pieces of code are implemented in dissimilar technologies.

For a distributed application, these are probably the most common architectures:

- ▶ Distributed object architecture
- ▶ Service-oriented architecture
- ▶ Message-oriented architecture

Let's consider these classifications further.

4.2.1 Stateful synchronous interaction

Definitions:

Stateful interaction: Called code (the service) holds information (conversational state) on behalf of the calling code (the client) across multiple method or function invocations.

Synchronous interaction: Calling code (the client) waits for (blocks) the called code (the service) to complete its operation before continuing with any further processing itself.

Stateful synchronous interaction: Interaction between the calling code (the client) and the called code (the service) is both stateful and synchronous.

Example 4-1 illustrates a pseudo interface definition which implies a stateful interaction semantic between the client code that drives the interface and the service code that implements the interface.

Example 4-1 Pseudo interface definition with a stateful semantic

```
interface iCalculator1
{
    void setArg1([in]float arg1);
    void setArg2([in]float arg2);
    [retval]float add();
    ...
}
```

Example 4-2 illustrates pseudo client code implementation driving the iCalculator1 interface from Example 4-1.

Example 4-2 Pseudo client code to invoke the interface from Example 4-1

```
...
float arg1 = 1;
float arg2 = 2;
float result = 0;
iCalculator1 objCalculator = new calculator();
objCalculator.setArg1(arg1);
objCalculator.setArg2(arg2);
result = objCalculator.add();
delete objCalculator;
assert(3 == result);
...
```

It can be seen that client code makes three method invocations of the service implementation (setArg1(...), setArg2(...) and add()). In order for the service implementation to correctly service the client's request, it must maintain state on behalf of the client across all three invocations. This implies that the service implementation must be associated with a single client (or must be able to distinguish between clients) until the client no longer requires its services (the implication is that the service implementation is stateful).

Because multiple communications are required to fulfill a single task, this style of interface can be classified as *chatty* or *verbose*. One disadvantage of this chatty interface is that when the service implementation is deployed in another process (either locally on the same physical machine, or remotely on a separate machine), each method invocation results in an inter-process communication (see Figure 4-1 on page 114).

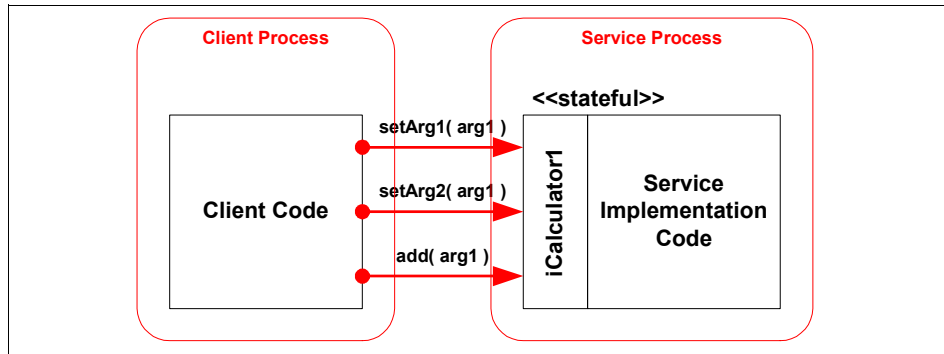


Figure 4-1 An (implied) stateful chatty interface

When we consider the technical implications associated with bridging an application deployed in WebSphere and an application deployed in the .NET Framework, this chatty style of interaction is unattractive.

Note: The technical implications associated with bridging these applications are discussed in 4.3, “Layer interaction classifications” on page 132.

Figure 4-2 and Figure 4-3 on page 115 illustrate how chatty interfaces can be given a concise façade to minimize inter-process communications. Applying these techniques to chatty interfaces between a WebSphere application and .NET application may result in a better performing solution than the chatty interface.

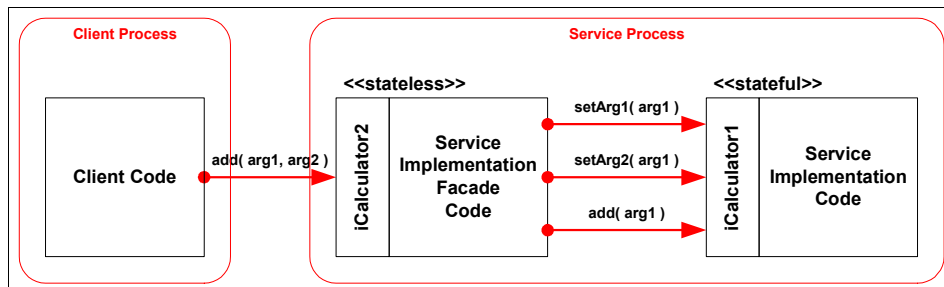


Figure 4-2 A concise façade to a chatty service

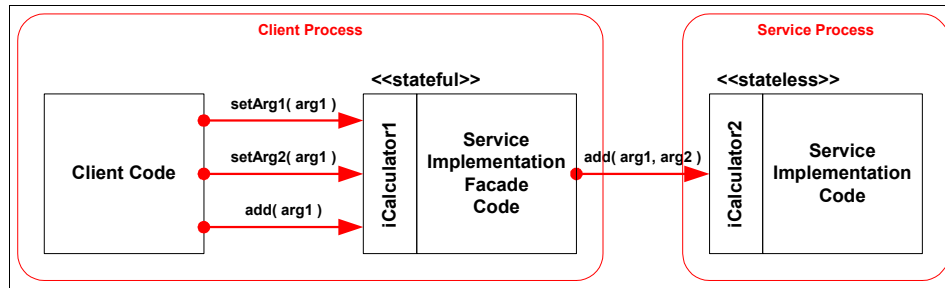


Figure 4-3 A concise façade to a chatty client

4.2.2 Stateless synchronous interaction

Definitions:

Stateless interaction: Called code (the service) does *not* hold information (conversational state) on behalf of the calling code (the client) across multiple method or function invocations.

Stateless synchronous interaction: Interaction between the calling code (the client) and the called code (the service) is both Stateless and Synchronous.

Example 4-3 illustrates a pseudo interface definition which implies a stateless interaction semantic between the client code that drives the interface and the service code that implements the interface. This interface differs from the interface in Example 4-1 on page 113, because all the arguments to the `add()` method are passed as input parameters to the `add()` method itself. As a consequence, the service implementation can service the client's request in a single method invocation (see Figure 4-4 on page 116). Consequently, the service implementation has no requirement to maintain any client specific state beyond the scope of a single service request. Example 4-4 on page 116 illustrates pseudo client code implementation driving the `iCalculator2` interface from Example 4-3.

Example 4-3 Pseudo interface definition with a stateless semantic

```

interface iCalculator2
{
    [retval]float add([in]float arg1, [in]float arg2);
    ...
}
  
```

Example 4-4 Pseudo client code to invoke the interface from Example 4-3 on page 115

```
...  
float arg1 = 1;  
float arg2 = 2;  
float result = 0;  
iCalculator2 objCalculator = new calculator();  
result = objCalculator.add(arg1, arg2);  
delete objCalculator;  
assert(3 == result);  
...
```

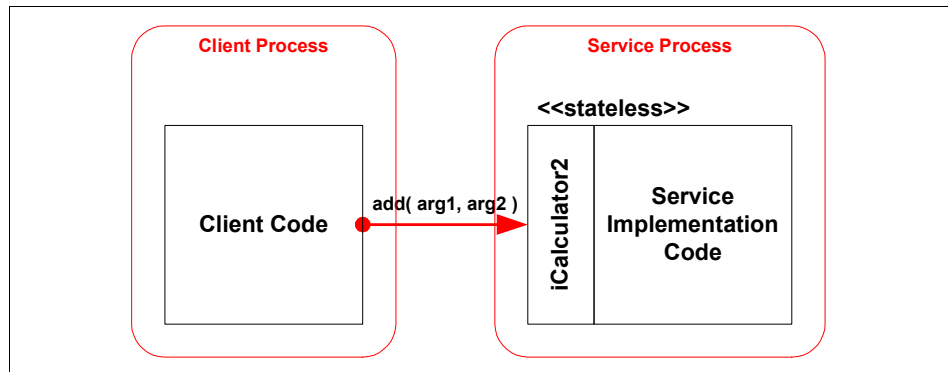


Figure 4-4 A stateless interface (implied)

Example 4-5 illustrates an alternative pseudo interface definition. This definition also implies a stateless interaction semantic between the client code that drives the interface and the service code that implements the interface. This differs from Example 4-3 on page 115 in that it passes a *state* object instance (of type `iCalculator3Args`) as the single input parameter to the `add()` method. Example 4-6 on page 117 illustrates pseudo client code implementation driving the `iCalculator3` interface from Example 4-5.

Example 4-5 An alternative pseudo interface definition with a stateless semantic

```
interface iCalculator3Args  
{  
    void setArg([in]float arg);  
}  
  
interface iCalculator3  
{  
    [retval]float add([in]iCalculator3Args args);  
    ...  
}
```

```
...
float arg1 = 1;
float arg2 = 2;
float result = 0;
iCalculator3Args objCalculatorArgs = new calculator3Args
objCalculatorArgs.setArg(arg1);
objCalculatorArgs.setArg(arg2);
iCalculator3 objCalculator = new calculator();
result = objCalculator.add(objCalculatorArgs);
delete objCalculator;
assert(3 == result);
...
```

We have assumed that the implementation of interface `iCalculator3Args` in Example 4-7 on page 118 is local to the client; do not assume this in real scenarios.

These interfaces *imply* a stateless interaction, but it should be remembered that stateless is really a classification of an implementation, not a classification of an interface. Do not assume an implementation is stateless just because its interface implies that it is.

4.2.3 Stateless asynchronous interaction

Definitions:

Asynchronous interaction: Calling code (the client) does *not* wait for (block) the called code (the service) to complete its operation before continuing with any further processing.

Stateless asynchronous interaction: Interaction between the calling code (the client) and the called code (the service) is both Stateless and Asynchronous.

Stateful interactions are generally synchronous (this is not always the case, but frequently so). That is, if client code needs to make several invocations of a service, and state needs to be maintained by the service implementation on behalf of the calling client between these invocations, and the order (sequence) of these invocations is important, then the client is generally synchronized with the service. This normally means that the client invokes the service, then waits (blocks) until the service responds and returns logical flow back to the client before the client invokes the service again.

Because stateless interaction (generally) communicates all the state for a given task in one invocation, stateless service implementations (generally) have more potential to be implemented as asynchronous (non-blocking) implementations than stateful service implementations.

Example 4-7 illustrates a pseudo interface definition which implies a stateless asynchronous interaction semantic between the client code that drives the interface and the service code that implements the interface. The implication is that the `delegateTask()` method takes some abstracted state (a message or document), acknowledges receipt of the message (returns logical flow to the client), then processes the message in its own time.

Example 4-7 Pseudo interface definition(s) with a stateless asynchronous semantic

```
interface iBusinessProcess
{
    void delegateTask([in]iMessage arg1);
    ...
}
```

Example 4-8 Pseudo client code to invoke the interface in Example 4-7

```
...
iMessage objMessage = new Trade('sell Microsoft stock, buy IBM stock');
iBusinessProcess objTrader = TradeProcessor();
objTrader.delegateTask(objMessage);
...
```

A stateless asynchronous interaction does not necessarily imply messaging middleware (such as WebSphere MQ). For example, a service proxy or a service façade could deliver an asynchronous solution using threads.

Asynchronous processing is useful when a consumer requires independent services from more than one service provider, especially when the elapsed time for processing for any particular service provider is unpredictable or simply takes too long. Since information provided back to the consumer at the time of service initiation is scant, asynchronous operations typically are not used for real-time queries, but are often used to initiate a request or action on the part of a service provider.

Because stateless interaction communicates all the state for a given task in one invocation, stateless service implementations have a greater potential to be implemented asynchronously than do stateful interactions.

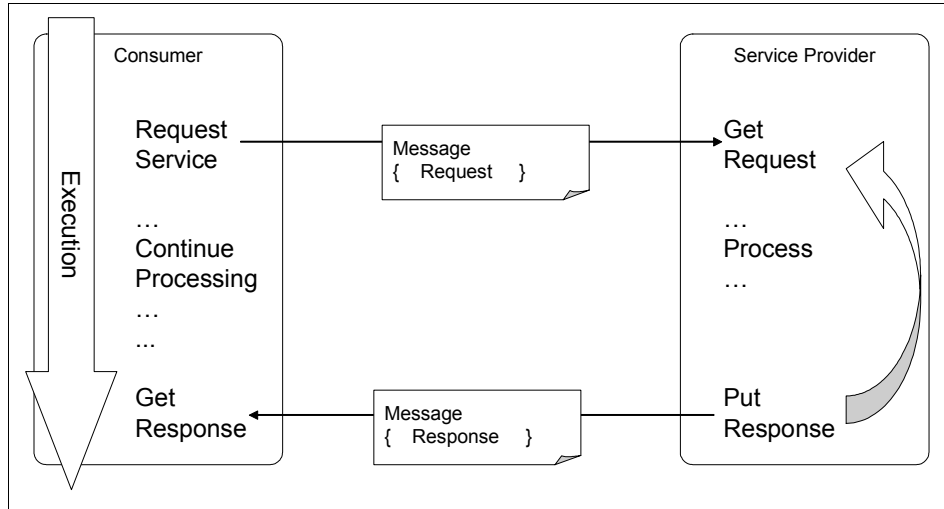


Figure 4-5 Asynchronous interaction between consumer and service provider

Figure 4-5 illustrates an asynchronous request between a consumer and service provider. As you can see, the service provider is written to consume incoming requests and provide responses while the consumer provides the requests, continues processing as the request is satisfied and then gets the response. Although we show this entire activity taking place within a single consumer execution, this is not a requirement. The only requirement to be considered asynchronous is that the consumer not block execution between the request and the response.

Although we represent the information being passed between the artifacts here as messages, a stateless asynchronous interaction does not necessarily imply messaging middleware. A service proxy or a service façade can also deliver an asynchronous solution using threads. In the same way, the use of messaging middleware does not automatically imply asynchronous invocation either. Many applications that use messaging middleware are written using a synchronous request/response paradigm.

4.2.4 Stateful asynchronous interaction

Definition:

Stateful asynchronous interaction: Interaction between the calling code (the client) and the called code (the service) is both Stateless and Asynchronous.

Important: We will not be considering stateful asynchronous interactions in this book.

Stateful interactions are usually synchronous. That is, if a consumer needs to make several invocations of the service, and the state needs to be maintained by the service implementation between these invocations, and the order (sequence) of these invocations is important, then the consumer must be synchronized with the service. This normally means that the consumer invokes the service, then waits (blocks) until the service responds before invoking the service again.

Consider, then, how the management of state is simply the storing of state information. Under the asynchronous paradigm, the state can be stored by the service provider in the same way, but this often makes little sense because asynchronous service providers are usually engineered so that they don't have to remain running. A better way to store state information for asynchronous interaction is within the messages passed between the consumer and provider.

Although there are limitations to the type and amount of state information it is possible to store, this makes for some interesting and useful new paradigms regarding state where asynchronous operations occur as represented in the following two figures.

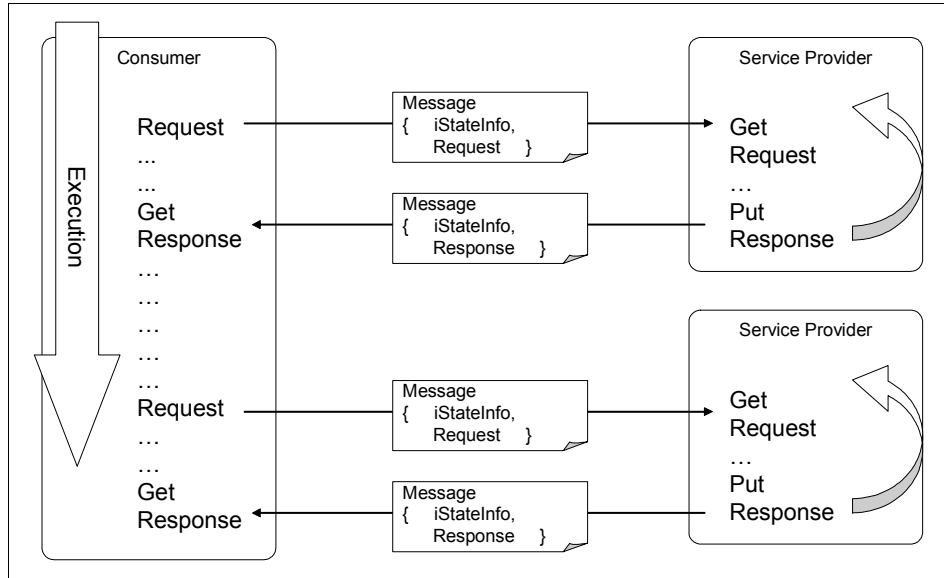


Figure 4-6 Stateful interaction using multiple service providers

In Figure 4-6, we see that the messages contain a mutually agreed-upon `iStateInfo` object containing state information. Because the message maintains state, the service providers are capable of performing state-dependent operations while the interaction remains asynchronous in nature.

In Figure 4-7 on page 122, we see that we can easily aggregate services as well using the asynchronous paradigm while maintaining state information. Properly engineered messages between the artifacts can help maintain the loosely-coupled nature of these interactions.

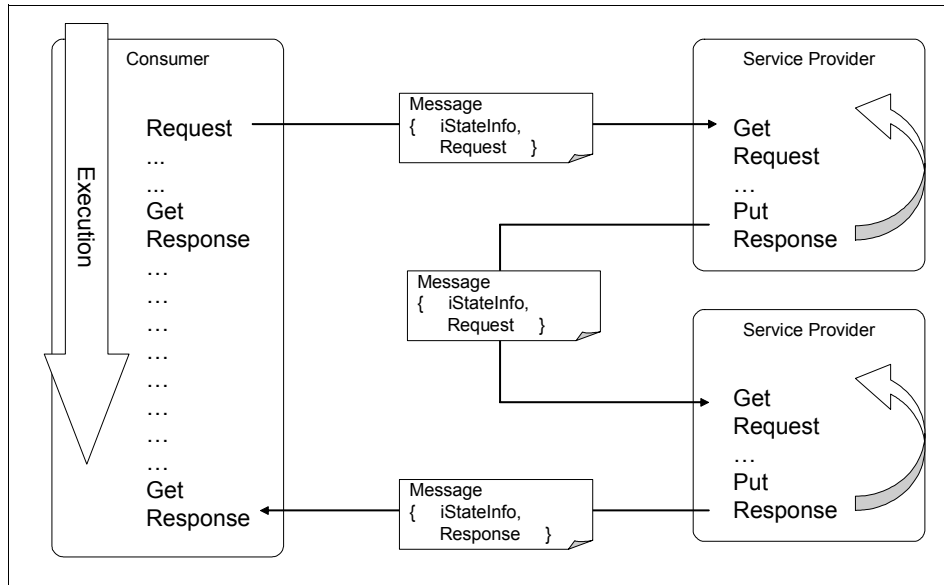


Figure 4-7 Stateful asynchronous service aggregation

Finally, asynchronous processing allows the introduction of other powerful middleware tools such as WebSphere Integrator. Integrator provides sophisticated message transformation and routing capability, allowing the creation of highly-sophisticated message flows between consumers and providers.

4.2.5 RPC interface style

Definitions:

RPC (Remote Procedure Call) call mechanism: RPC is a generic term for a protocol that allows calling code (the client) to invoke a procedure (the service) over a distributed (cross-process or cross-network) environment. An RPC protocol implementation abstracts the remote (inter-process) call mechanism from both the client code and the service code. From both the client implementations perspective and the service implementations perspective, the call seems like a regular local (in-process) function call with distinct and discrete arguments.

RPC interface style: An interface style in which each call argument represents a simple, distinct and discrete piece of information that is strongly typed and of a predetermined size. This interface style implies that each argument is marshalled from the client to the service (and vice versa) as a distinct typed value, and the marshalling infrastructure (the RPC call mechanism implementation) understands the fundamental data type of each argument being marshalled.

Assuming a distributed or multi-process implementation, the interfaces in Example 4-9 imply an RPC call paradigm.

Example 4-9 Some interfaces imply an RPC call style (assuming an inter-process call)

```
void foey1([in]float arg1, [in]float arg2, [in]myclass arg3);  
[retval]int foey2([in,out]int arg1);
```

Examples of middleware infrastructure that support the RPC call style are RMI/IIOP, the .NET TCP remoting channel, DCOM, and RPC style SOAP.

4.2.6 Document interface style

Definitions:

Document interface style: An interface style in which all (or many) call arguments are packaged by the calling code (the client) into one or more compound documents (messages), and these documents are communicated to the called code (the service).

Document call paradigm: A call mechanism that implies that all arguments are marshalled from the client to the service (and vice versa) as untyped binary arguments of determinable sizes, and that the remote call infrastructure (often an RPC call mechanism implementation or a messaging middleware) does not understand, or care about, the structure or semantic of the documents.

Obviously, both the calling code and the called code need to have a common understanding of the structure and semantic of the communicated documents.

A document call paradigm implies that the middleware infrastructure supporting the communication between a client and a service does not care about the structure or type of the state (the message) that is being communicated. It only cares about the message size and destination. However, both the client implementation and the service implementation need to have a common understanding of structure and semantics of the communicated state (message).

An example of a technology neutral message could be an XML Schema typed XML document. With this example, both the client and this service have prior knowledge of the structure of the communicated XML document from the XML Schema, and both have prior knowledge of the semantic of the message (delivered by the programmer at construction time as structure programming constructs).

Examples of middleware infrastructure that support this call paradigm are WebSphere MQ messaging, Microsoft MSMQ, and Document style SOAP.

Example 4-10 Some interfaces imply a Document call paradigm

```
void method3([in,out]byte[] arg1);  
[retval]int method4([in]myXML arg1);
```

The interfaces illustrated in Example 4-5 on page 116 (iCalculator3) and Example 4-7 on page 118 (iBusinessProcess) imply, but do not mandate, a document call paradigm implementation.

Documents call paradigm and the RPC call mechanism

In reality, at an implementation level, both documents and fixed types are serialized as bytes between processes and between network endpoints. Figure 4-8 illustrates the document call paradigm as an abstraction over an RPC call mechanism. The implication is that the document call paradigm is modeling abstraction for the benefit of the relationship between client and service. The interface to the service (or, depending on the technical solution, the interface to the supporting middleware) delivers the document call paradigm.

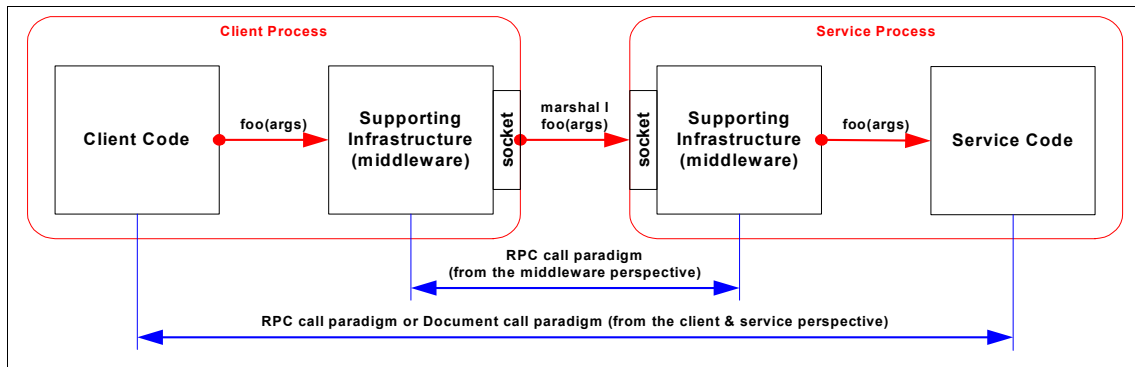


Figure 4-8 The document call paradigm as an abstraction over an RPC call mechanism

4.2.7 Argument by value paradigm

Definitions:

Pass by value: When an interface argument (parameter) is a *pass by value* argument, the called code (the service) receives a copy of the argument. Any changes made to the value of a pass by value argument by the called code *do not* affect the value of the caller's original copy, and *are not* visible to the calling code.

In-only arguments: Pass by reference arguments are often referred to as in-only arguments. For example:

```
method([in] arg1); is equal to... method([byValue] arg1);
```

Figure 4-9 on page 126 illustrates a *pass by value* paradigm using the `iCalculator3` interface from Example 4-5 on page 116 and Example 4-6 on page 117. Note that real interfaces can (and frequently do) have both *by value* and *by reference* arguments in the same interface definition.

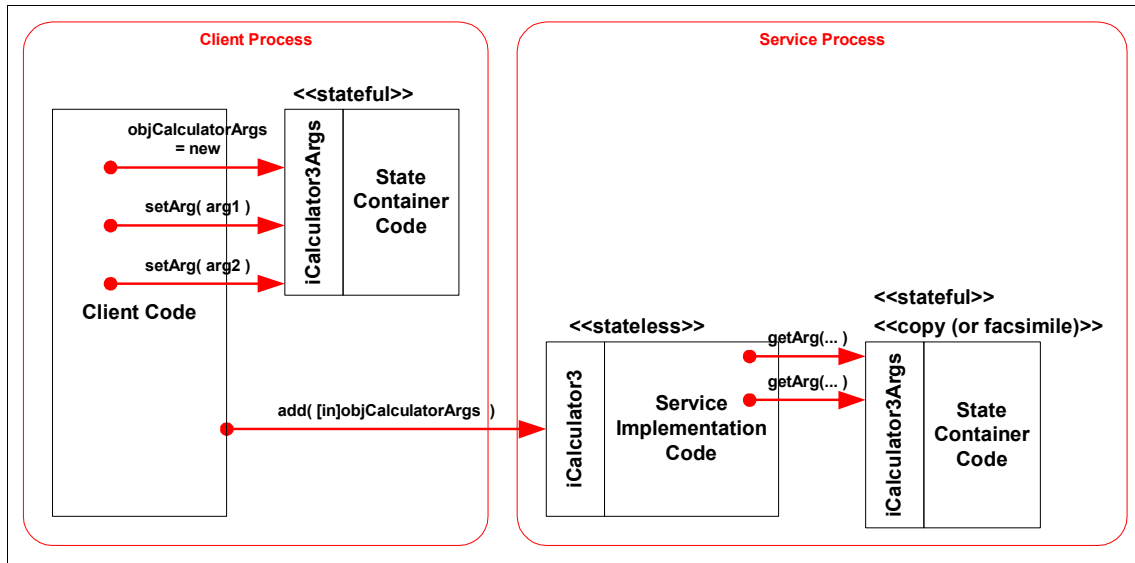


Figure 4-9 A pass by value paradigm

Java and all .NET programming languages support the *argument by value* paradigm in a distributed environment.

One of the challenges in integration between WebSphere applications and .NET applications across remote (or local) interfaces is that we need to identify a technical solution that maps argument types from Java to .NET (and vice versa).

4.2.8 Argument by reference paradigm

Definitions:

Pass by reference: When an interface argument (parameter) is a *pass by reference* argument, the called code (the service) receives a reference to the caller's instance of the argument. Any changes made to the value of a pass by reference argument by the called code *do* affect the value of shared argument instance, and are visible to the calling code (the client).

In/out arguments: Pass by reference arguments are often referred to as in/out arguments. For example:

```
foo([in,out] arg1); is equal to... foo([byReference] arg1);
```

Figure 4-10 illustrates a *pass by reference* paradigm using the `iCalculator3` interface from Example 4-5 on page 116 and Example 4-6 on page 117. Note that real interfaces can (and frequently do) have both *by reference* and *by value* arguments in the same interface definition.

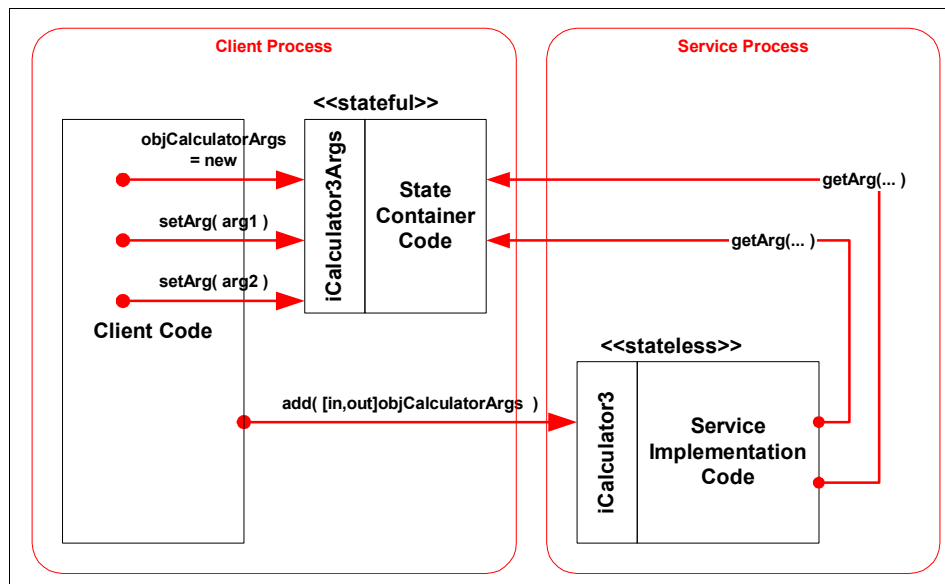


Figure 4-10 A pass by reference paradigm

Pass by reference interfaces can often result in chatty solutions. Contrast the three inter-process communications in Figure 4-10 with the single inter-process communication in Figure 4-9 on page 126.

The Java language is *pass by value* for primitive types (int, float, boolean, etc.) and *pass by reference* for complex types (arrays and objects). RMI and RMI/IIOP follow the same *pass by value* and *pass by reference* rules as basic Java.

.NET is also *pass by value* for primitive types (int, struct, enum etc.) and *pass by reference* for complex (arrays and object, including boxed primitive types). .NET remoting can provide *pass by reference* for complex types.

Definition: boxing

In .NET, all types (even primitive types) are fundamentally objects. For example, an int is actually an alias to the CTS System.Int32 type. System.Int32 derives from System.ValueType, so it has *pass by value* behavior.

Boxing is a .NET technique that allows a *pass by value* type to be treated as a *pass by reference* type. For example:

```
//--- i is by default a 'pass by value' object ---  
int i = 99;  
//--- boxed_i is a 'pass by reference' copy of i ---  
object boxed_i = i;  
//--- unboxed_i is a 'pass by value' copy of boxed_i ---  
int unboxed_i = (int)boxed_i;
```

One challenge for integration between WebSphere applications and .NET applications across remote (or local) interfaces is that we need to identify a technical solution that maps argument types from Java to .NET (and vice versa) while maintaining the pass by reference paradigm.

4.2.9 Distributed object architecture

Definition:

Distributed object architecture: An extension of the object-oriented application architecture which allows objects (that is, combined state and behavior) to live across a physically distributed environment. Clients direct the life cycle of the objects on a remote machine, maintain references to these remote objects, set properties and invoke methods on them.

Examples of technologies that support the distributed object paradigm in WebSphere are:

- ▶ RMI between Java Objects
- ▶ RMI/IIOP between Java Objects and EJBs in WebSphere
- ▶ RMI/IIOP between Java (including EJBs in WebSphere) and CORBA (encompassing many implementation languages)

An example of technologies that support the distributed object paradigm in .NET is:

- ▶ TCP remoting between .NET components

It is not possible to integrate these sets of distributed object technologies directly: you cannot, for example, create a .NET object from Java using RMI. This implies that if a conjoined distributed object paradigm is important to your WebSphere and .NET coexistence challenge, then we need to identify a technical solution (bridging technology) that can handle this paradigm.

Note: Some open source projects are building IIOP remoting channels for .NET. This has the potential to make Java and .NET integrable in a distributed object architecture.

4.2.10 Message Oriented Architecture

Definition:

Message Oriented Architecture: A model for point-to-point communication of a message from one location to another. Point-to-point can be extended to many-to-many communication (publish/subscribe pattern).

Document interface style: An interface style in which all (or many) call arguments are packaged by the calling code (the client) into one or more compound documents (messages), and these documents are communicated to the called code (the service).

Document call paradigm: A call mechanism that implies that all arguments are marshalled from the client to the service (and vice versa) as untyped binary arguments of determinable sizes, and that the messaging infrastructure does not understand, or care, about the structure or semantic of the message.

Obviously, both the calling code and the called code need to have a common understanding of the structure and semantic of the communicated messages.

An example of technologies that support the distributed object paradigm in WebSphere is:

- WebSphere MQ between Java and .NET (and various other languages and technologies).

An example of technologies that support the distributed object paradigm in .NET is:

- Microsoft MQ .NET clients and .NET services (and also between COM and .NET).

While MSMQ and WebSphere MQ (the technology formerly known as IBM MQ Series) both deliver a message-oriented paradigm, they are not directly compatible technologies. It is definitely possible to build (and parse) any given message structure in both a WebSphere application and a .NET application; the WebSphere MQ queue manager and the MSMQ queue managers are not directly compatible or interoperable.

Luckily, WebSphere MQ provides both Java and .NET client APIs. This makes WebSphere MQ a very good candidate solution technology for addressing coexistence between WebSphere applications and .NET applications.

4.2.11 Service-oriented architecture

Definition:

Service-oriented architecture; A distributed application architecture where applications (can) both expose discrete functionality (behavior or services) to calling applications (clients) and consume discrete functionality (behavior or services) exposed by other applications (services).

Service interface definitions and service binding definitions are (generally) implementation technology neutral, enabling abstraction (concealment) of the client's implementation technology for the service, and abstraction (concealment) of the service's implementation technology from the client.

Examples of technologies that support the service orient architecture are SOAP Web Services, CORBA, DCE RPC. Message-oriented technologies and distributed object technologies can also be used to implement service-oriented solutions.

4.2.12 Conclusions and recommendations

Hopefully, we have provided you with sufficient information in this chapter to help you consider and identify the characteristics for your given coexistence integration challenges. The next section, 4.3, "Layer interaction classifications" on page 132 should help you further classify your specific interaction case (or cases).

Some design issues transcend implementation technologies. With this in mind, we make the following technology neutral recommendations that we anticipate will benefit integration between an application deployed in WebSphere and an application deployed in .NET:

- ▶ Do not design chatty or stateful interfaces; they generally do not perform or scale well.
- ▶ Design inter-process interfaces with coarse *business* granularity, not with fine *implementation* granularity.
- ▶ Do not assume that an implementation is stateless just because its interface implies a stateless interaction.
- ▶ Try to minimize inter-process and cross-machine communication.
- ▶ Try to take inter-process and cross-machine communication away from the critical execution path. This implies that you should consider asynchronous communications whenever pertinent.

- ▶ If you have to inter-operate with an existing chatty service interface, consider using a façade in the service process. This is illustrated in Figure 4-2 on page 114.
- ▶ If you have to interact with an existing chatty client, consider using a façade in the client process. This is illustrated in Figure 4-3 on page 115.
- ▶ Stateless interfaces can be chatty; do not assume that they are not (see Figure 4-10 on page 127).
- ▶ Consider using asynchronous techniques to reduce call blocking so as to improve invocation responsiveness.

4.3 Layer interaction classifications

In the previous section, we identified several possible interaction classifications between an application deployed in WebSphere and an application deployed in the .NET Framework. In Chapter 3, “An architectural model for coexistent applications” on page 93, we discussed a layered architectural model which can be used to model software solutions for both J2EE applications and .NET applications. Let’s take these interaction classifications and this layered architecture, and enumerate some potential cross-technology and cross-layer interaction scenarios for further consideration.

If we consider two hypothetical layered applications, one a WebSphere application and the other a .NET application, the most likely layer-to-layer, single point to single point interactions between these applications are as follows:

- ▶ *Case a:* Client layer to Client layer
- ▶ *Case b:* Client layer to Presentation layer
- ▶ *Case c:* Client layer to Business layer
- ▶ *Case d:* Presentation layer to Presentation layer
- ▶ *Case e:* Presentation layer to Business layer
- ▶ *Case f:* Business layer to Business layer
- ▶ *Case g:* Business layer to Resource layer
- ▶ *Case h:* Resource layer to Resource layer

Note: We will be explicitly modeling *one-to-one* interactions between coexistent WebSphere application layers and .NET application layers. Depending on the scale of your given coexistence challenge, it is likely that you will need to consider potential added complexities (such as concurrency and quality of service) of *many-to-many*, *one-to-many*, or *many-to-one* runtime interaction dynamics.

<http://www.ibm.com/developerworks/patterns/>
<http://www.enterpriseintegrationpatterns.com/>

Figure 4-11 on page 134 and Figure 4-12 on page 135 illustrate these potential interactions from the perspective of both the WebSphere application and the .NET application. In each figure, we have also drawn an interoperation integration layer separating the two coexistent applications' strata.

Interactions from the perspective of an application deployed in WebSphere (Figure 4-11 on page 134) have been labeled as follows:

- ▶ *Case a1*: WebSphere Client layer to .NET Client layer
- ▶ *Case b1*: WebSphere Client layer to .NET Presentation layer
- ▶ *Case c1*: WebSphere Client layer to .NET Business layer
- ▶ *Case d1*: WebSphere Presentation layer to .NET Presentation layer
- ▶ *Case e1*: WebSphere Presentation layer to .NET Business layer
- ▶ *Case f1*: WebSphere Business layer to .NET Business layer
- ▶ *Case g1*: WebSphere Business layer to .NET Resource layer
- ▶ *Case h1*: WebSphere Resource layer to .NET Resource layer

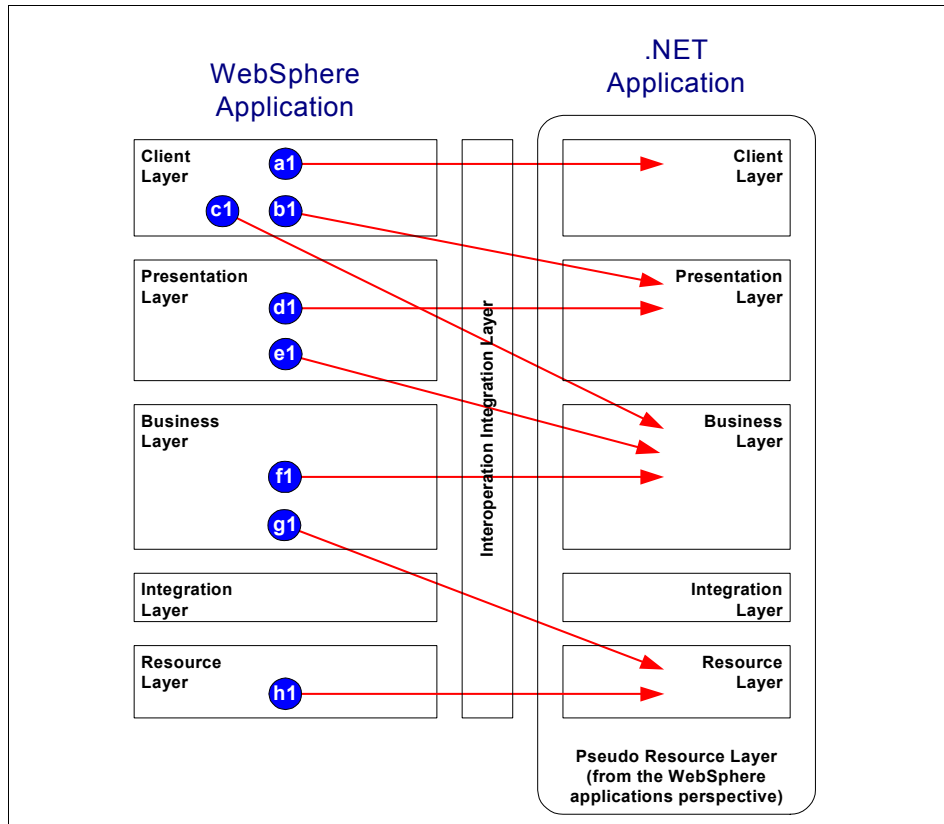


Figure 4-11 Potential cross-tier interactions from the perspective of a WebSphere application

Interactions from the perspective of an application deployed in the .NET Framework (Figure 4-12 on page 135) have been labeled as follows:

- ▶ Case a2: .NET Client layer to WebSphere Client layer
- ▶ Case b2: .NET Client layer to WebSphere Presentation layer
- ▶ Case c2: .NET Client layer to WebSphere Business layer
- ▶ Case d2: .NET Presentation layer to WebSphere Presentation layer
- ▶ Case e2: .NET Presentation layer to WebSphere Business layer
- ▶ Case f2: .NET Business layer to WebSphere Business layer
- ▶ Case g2: .NET Business layer to WebSphere Resource layer
- ▶ Case h2: .NET Resource layer to WebSphere Resource layer

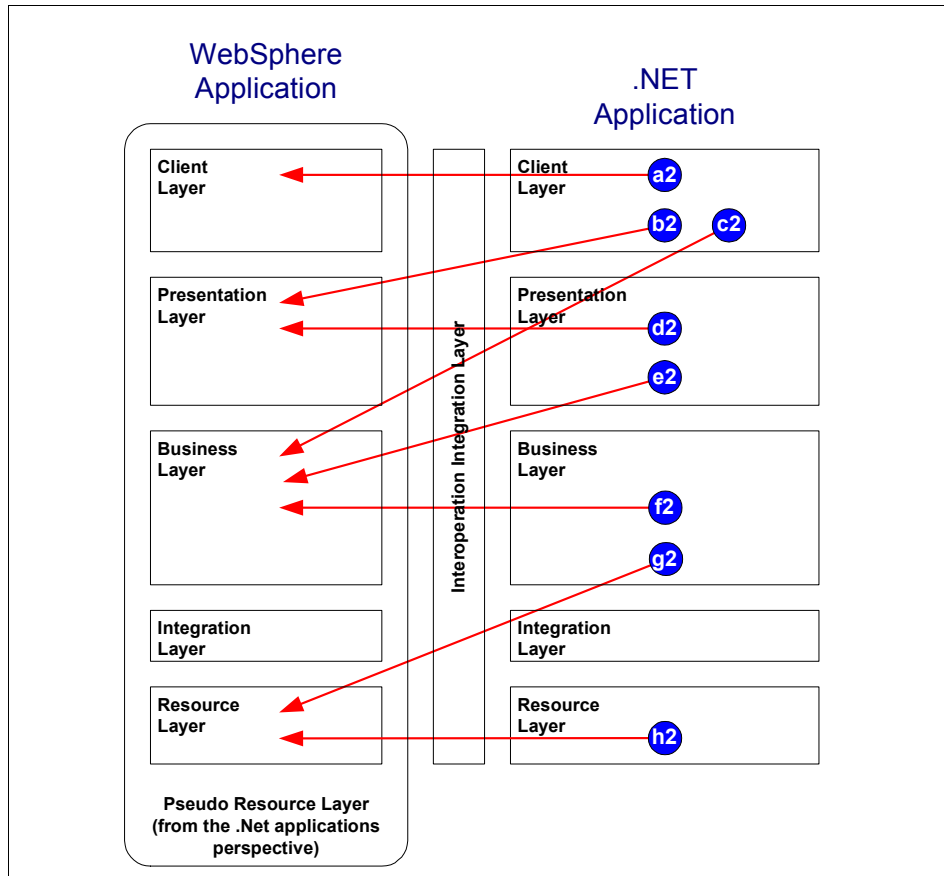


Figure 4-12 Potential cross-tier interactions from the perspective of a .NET application

Each of these interactions assumes an event flow either horizontally across identical coexistent layers (for example, case a: Client layer to Client layer), or diagonally from one layer to a lower coexistent layer (for example, case c: client to business). Each of these interaction cases has the potential to form the basis of a solution model for an integration relationship between the layers of our coexistent applications.

The fundamental event flow direction is *down* through the strata of our applications layers. This is representative of typical client (user) initiated events. In real applications, sometimes events are initiated *up* through the strata of layers as callbacks (for instance, from the Business layer to the Client layer or from the Resource layer to the Business layer). We will not be illustrating these explicitly in this book, but it is not difficult to imagine how these events could be modeled in a similar way.

In order to fulfill any one of these interaction cases (a1, a2, b1, b2 , etc.), certain criteria need to be met by the candidate technical solution for the interoperation integration layer. Specifically, we need the following features:

- ▶ *A Head Protocol*: a client technology bindable application programming interface (API).
- ▶ *A Service Discovery Mechanism*: a mechanism for the client to either:
 - Tell the interoperation layer the physical location of the service, or
 - Give the interoperation layer a unique identifier, which will enable the interoperation layer to resolve the physical location of the service.
- ▶ *A Transport Mechanism*: a common network and network protocol used between the Head and Tail implementations.
- ▶ *A Communication Protocol*: a common set of rules used between the Head and Tail implementations to represent:
 - A service request
 - A service response
 - Argument type mapping
 - Call context
 - Multiple (stateful) call associations
- ▶ *A Tail Protocol*: a service technology bindable service provider interface (SPI).
- ▶ *A Service Binding/Hosting*: a mechanism for the Tail implementation to either:
 - Bind to an existing (hosted) service implementation, or
 - Launch or host, then bind to, a new service implementation.

These interoperation criteria are illustrated in Figure 4-13 on page 137 from the perspective of an interaction initiated by a runtime artifact in the .NET application (with a .NET Head Protocol and Java Tail Protocol). A very similar model can be drawn from a WebSphere applications perspective (with a Java Head Protocol and .NET Tail Protocol).

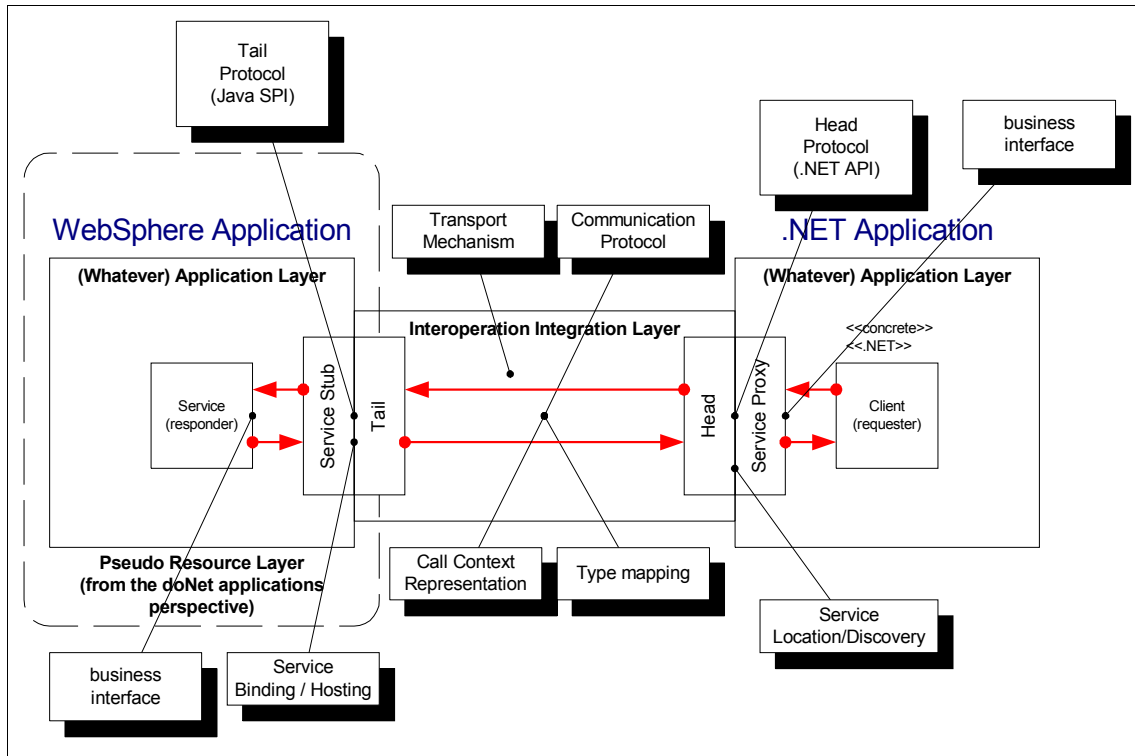


Figure 4-13 Interoperation integration layer criteria from the perspective of a .NET application

Fulfilment of these criteria will deliver service location and service implementation transparency. Any technology which meets these criteria is a candidate for consideration as a coexistence integration enabler. Any technology that provides both Head and Tail implementations in both Java and .NET is a very good candidate. Any technology that meets all of these criteria, including Java and .NET Head and Tail implementations, using open standards, is a prime candidate. Any technology that fails to meet any one of these criteria *may* not be a suitable candidate, or may have limited (perhaps tactical) capabilities.

Figure 4-14 on page 138 illustrates the *proxy-stub* pattern. The proxy exposes a business interface to the client implementation to bind to, acts as a proxy to the real service implementation, adapts the communication from the business interface to the integration solution's Head Protocol, and thus abstracts the client from the chosen integration solution. The stub adapts communication from the integration solution's Tail Protocol to the business interface, binds to the real business implementation, invokes its services on behalf of the original client code, and thus abstracts the service from the chosen integration solution. The implementations of these proxies or stubs can vary considerably, depending on

the required quality of service and on whether/how the chosen integration solution supports these qualities of service.

Use of the *proxy-stub* pattern proves separation of concerns (business interfaces, technical integration solution, adaption) and enables the entire integration mechanism between WebSphere and .NET to be encapsulated (concealed). This will allow us to choose any appropriate technical integration solution (tactical or strategic) without impacting either the client implementation code or the service implementation code. We will use the proxy-stub pattern wherever possible in the candidate solution designs we will present for our interaction cases.

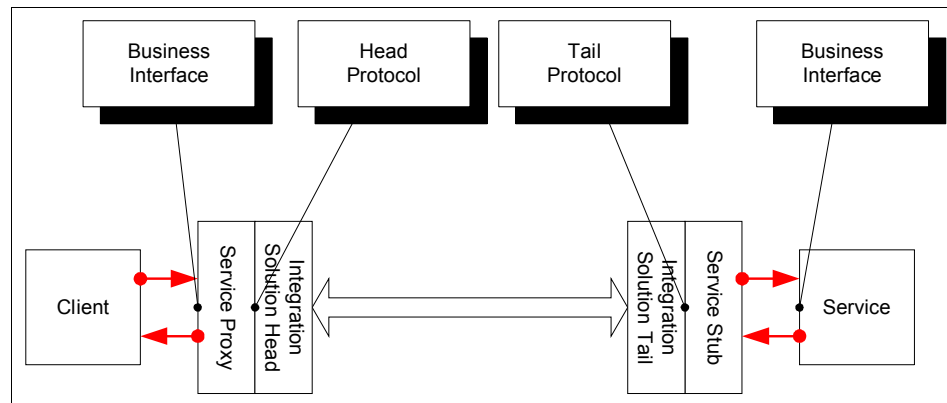


Figure 4-14 The proxy-stub pattern

Let us consider each of these point-to-point interaction cases further and identify some candidate solution models. If you already know that you are interested in a specific interaction case, then please feel free to skip straight to the appropriate subsection in this chapter.

4.3.1 Interaction case a: client logic to client logic

Let us consider the coexistence of an application deployed in WebSphere and an application deployed in the .NET Framework via Client layer logic to Client layer logic integration (case a). We can consider interaction between these two applications from the following perspectives:

► *Perspective a1: WebSphere to .NET*

A runtime artifact executing within a Fat Java Client application, such as a Swing client application or an AWT client application (possibly deployed in the WebSphere Client Container), interacting with a runtime artifact executing within a Fat .NET client application, such as a Windows Form application.

► *Perspective a2: .NET to WebSphere*

A runtime artifact executing within a Fat .NET client application, such as a Windows Form application, interacting with a runtime artifact executing within a Fat Java Client application, such as a Swing client application or an AWT client application (possibly deployed in the WebSphere Client Container).

are illustrated in Figure 4-15 on page 140.

Thin Java Clients and Thin .NET clients communicate directly with the presentation tier using HTTP. It probably makes little sense to consider thin Client layer to thin Client layer interactions. A potential hybrid scenario could be peer to peer *thin Client layer to fat client tier* interaction (via http). With this scenario, the fat client, deployed in a client tier, could be modeled as a pseudo Presentation layer to the thin client.

Explanation: The phrase '*thin client layer to fat client tier*' is intended to represent a peer to peer interaction where a Web browser client communicates via HTTP with another client application deployed on a desktop PC. In this scenario, the fat client is an HTTP server to the thin client. This is an interesting, but probably uncommon, solution architecture.

Definition: The following client application definitions apply to this book.

- Fat client - a generic description for a traditional GUI-based client tier application.
- Thin client - a generic description for a Web browser hosted client tier application.
- Fat Java Client - a fat client application implemented using Java technologies (typically using Swing, AWT or SWT)
- Fat .NET client - a fat client application implemented using .NET technologies (typically a Windows Form application)
- Thin Java Client - a thin client application that uses the downloaded client side logic implemented using Java technology (typically JavaScript, Applets or Java Beans).
- Thin .NET client - a thin client application that uses downloaded client side logic implemented using .NET technology (typically VBScript or ActiveX Components).

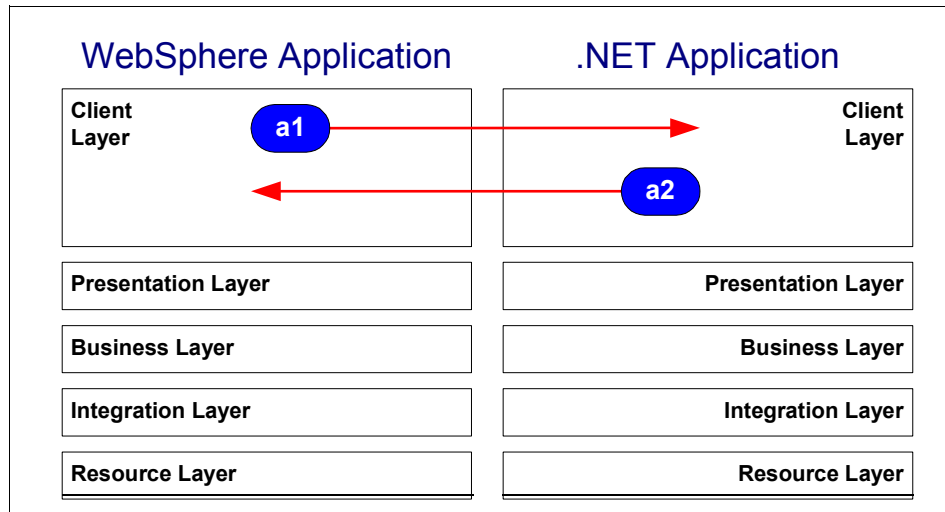


Figure 4-15 Interaction cases a1 (WebSphere application's perspective) and a2 (.NET application's perspective)

Interaction case a: structure and dynamism considerations

Consider the scenario where a .NET client application exposes some specialized calculator functionality. Suppose that a Java client application requires this functionality, and that it is considered an appropriate solution for the Java client application to make a peer to peer call to the .NET client application. Figure 4-16 on page 141 illustrates a high-level objective model for case a1 (a very similar model can be drawn for case a2).

Note: An iCalculator interface is used as an example throughout this book, and is intended to be simple to understand, rather than representative of a realistic interoperation scenario.

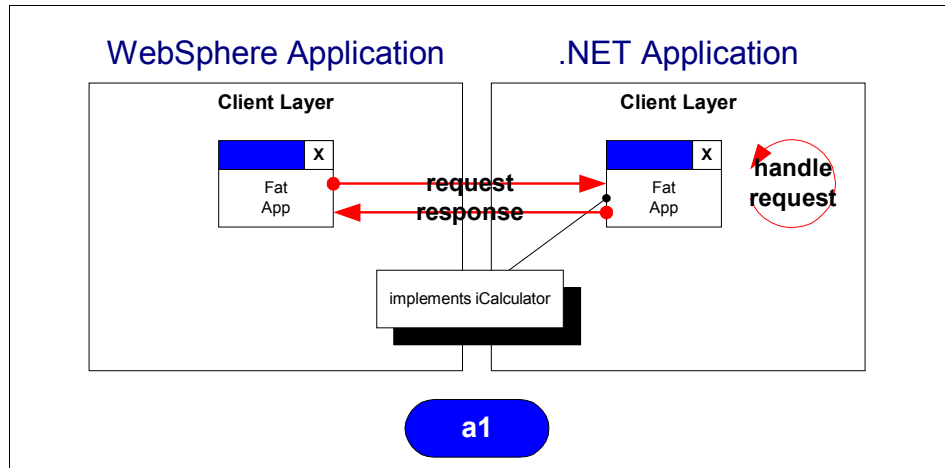


Figure 4-16 A high-level objective model for case a1

Figure 4-17 on page 142 illustrates a candidate solution model for interaction case a1. The proposed use of a Java service proxy and a .NET service stub fulfills our design principles of separation of concerns and encapsulation of the solution technology used for the interoperation integration layer. A very similar candidate solution model can be drawn for interaction case a2.

Figure 4-18 on page 143 illustrates that interaction case a1 (and by implication, a2) can be considered analogous to interaction case c1 (and c2). See 4.3.3, “Interaction case c: client logic to business logic” on page 152.

If we take a moment to consider thin clients, it *may* be possible to envisage invoking the service proxy illustrated in Figure 4-17 on page 142 using an applet, or client side scripting in a browser-based application. However, this presents the added complications of working with, or overcoming the default limitations of the browser sandbox. For thin client applications, it could be strongly argued that this service invocation should be performed from the Presentation layer as an analog of interaction case e. See 4.3.5, “Interaction case e: presentation logic to business logic” on page 167.

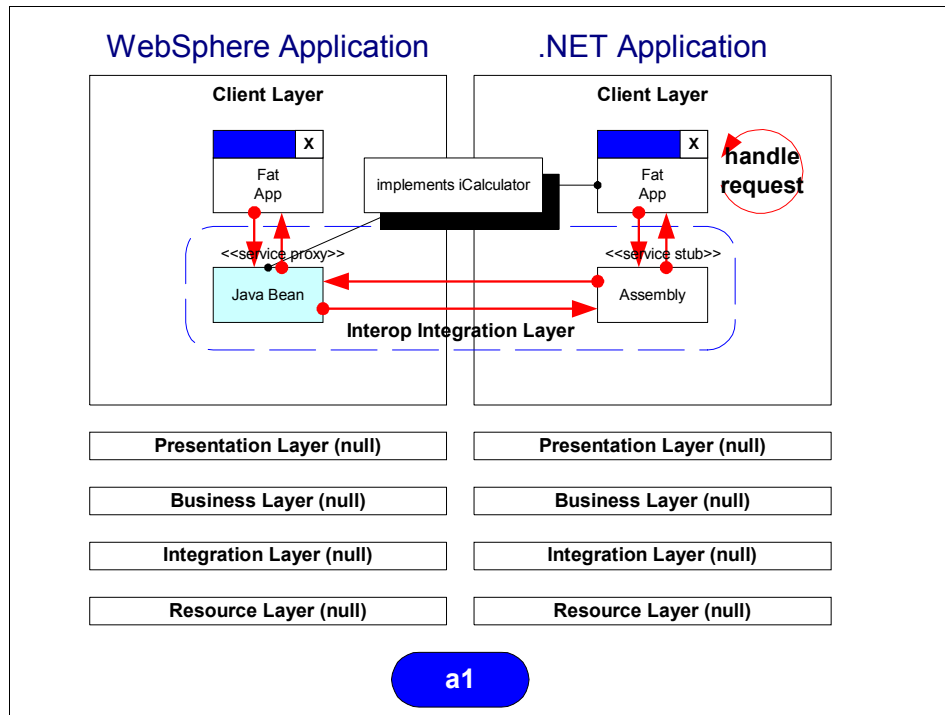


Figure 4-17 Interaction Case a1: candidate solution model (a very similar model can be drawn for interaction case a2)

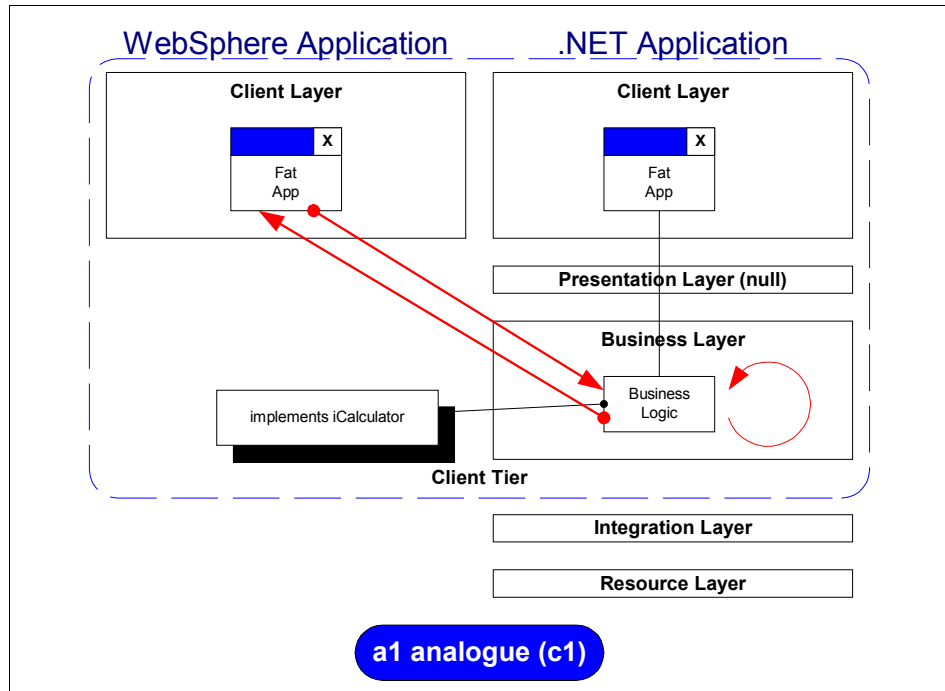


Figure 4-18 case a1 is technically analogous to case c1

Interaction case a: component interaction classifications

For each of these interaction perspectives (case a1 and case a2), the interaction between components can be considered to have one of the following integration classifications:

- ▶ *a*: Stateful Synchronous Integration.
- ▶ *b*: Stateless Synchronous Integration.
- ▶ *c*: Stateless Asynchronous Integration.
- ▶ *d*: Stateful Asynchronous Integration (not considered further in this book).

It is possible to further decompose these interactions into finer levels of detail, but these classifications will be adequate for modeling the technical challenges and technical solutions addressed in this book.

This gives us six potential interaction cases to consider:

- ▶ *Case a1.a*: Stateful Synchronous Integration from WebSphere to .NET.
- ▶ *Case a1.b*: Stateless Synchronous Integration from WebSphere to .NET.
- ▶ *Case a1.c*: Stateless Asynchronous Integration from WebSphere to .NET.

- *Case a2.a*: Stateful Synchronous Integration from .NET to WebSphere.
- *Case a2.b*: Stateless Synchronous Integration from .NET to WebSphere.
- *Case a2.c*: Stateless Asynchronous Integration from .NET to WebSphere.

Figure 4-19 and Figure 4-20 on page 145 illustrate these interaction cases from the perspective of a WebSphere application (cases a1.a, a1.b and a1.c) and a .NET application (cases a2.a, a2.b and a2.c), respectively.

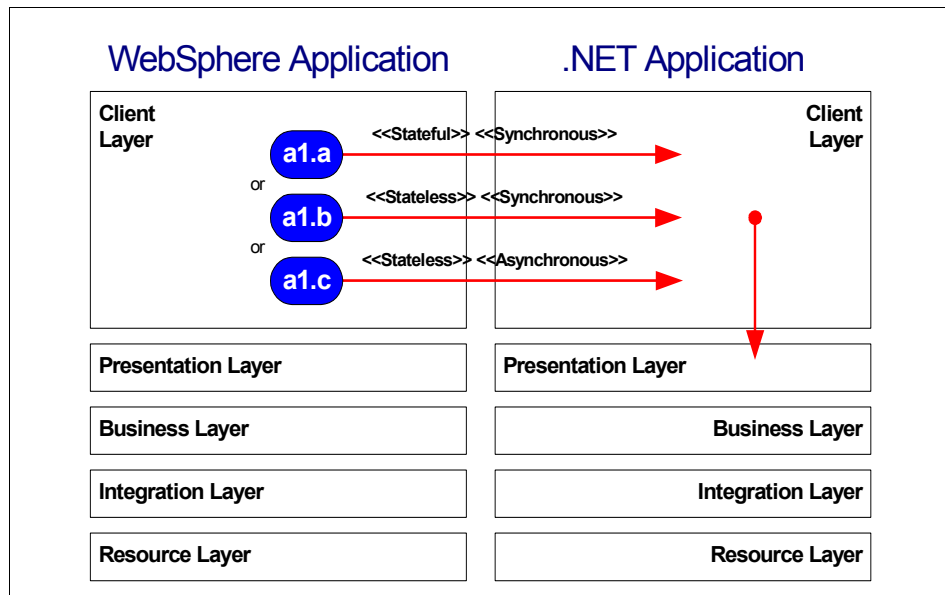


Figure 4-19 Interaction cases a1.a, a1.b and a1.c (WebSphere application's perspective)

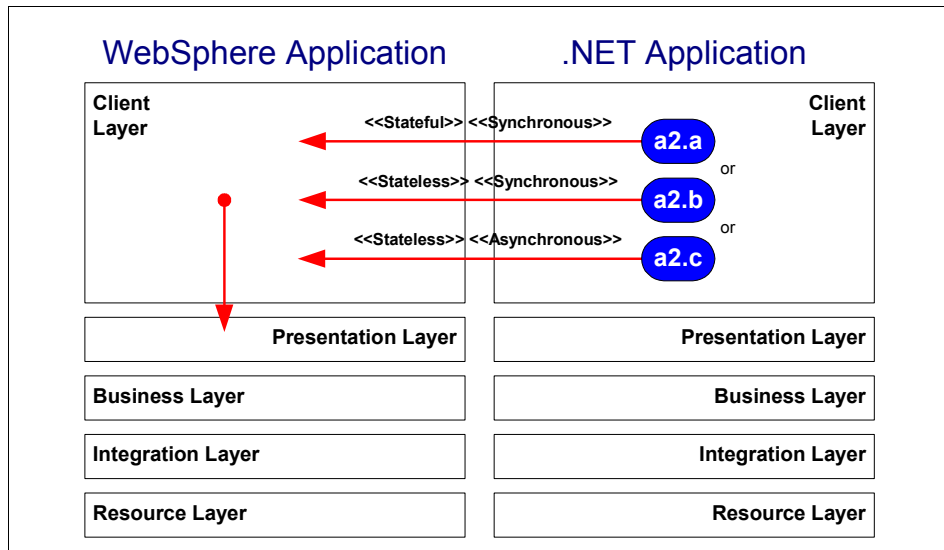


Figure 4-20 Interaction cases a2.a, a2.b and a2.c (.NET application's perspective)

4.4, "Technical solution mapping" on page 202 provides guidance on identifying potential technical solutions for each of these interaction cases (a1.a, a1.b, a1.c, a2.a, a2.b and a2.c).

4.3.2 Interaction case b: client logic to presentation logic

Let's consider the coexistence of an application deployed in WebSphere and an application deployed in the .NET Framework via Client layer logic to Presentation layer logic integration (case b). We can consider interaction between these two applications from these perspectives:

- *Perspective b1: WebSphere to .NET*
 - A runtime artifact executing within a Fat Java Client application, such as a Swing client application or an AWT client application (possibly deployed in the WebSphere Client Container), interacting with a runtime artifact executing within the Presentation layer of a .NET application (possibly an ASP.NET artifact) deployed in IIS.
 - A runtime artifact executing within a thin client application (possibly client side Java presentation logic such as a Java Applet), interacting with a runtime artifact executing within the Presentation layer of a .NET application (possibly an ASP.NET artifact) deployed in IIS.

► *Perspective b2: .NET to WebSphere*

- A runtime artifact executing within a Fat .NET client application, such as a Windows Form application, interacting with a runtime artifact executing within the Presentation layer of a WebSphere application (possibly a JSP or servlet) deployed in the WebSphere Web Container.
- A runtime artifact executing within a thin client application (possibly client side native presentation logic such as an ActiveX Control), interacting with a runtime artifact executing within the Presentation layer of a WebSphere application (possibly a JSP or servlet) deployed in the WebSphere Web Container.

are illustrated in Figure 4-21.

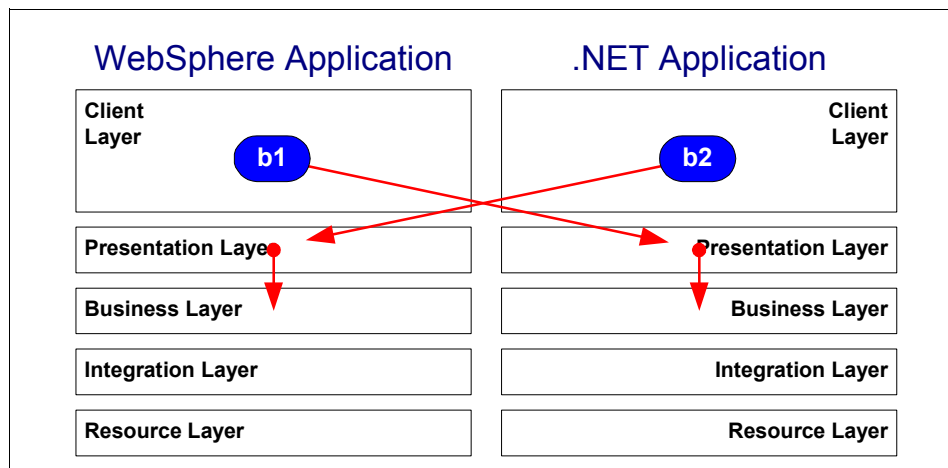


Figure 4-21 Interaction cases b1 (WebSphere application's perspective) and b2 (.NET application's perspective)

Interaction case b: structure and dynamic considerations

Consider the scenario where a WebSphere application exposes some enterprise configuration information at a fixed HTTP URL (perhaps an XML service description file such as a WSIL file). Say that a fat client .NET application requires this information at runtime to resolve service configuration. Figure 4-22 on page 147 illustrates a high-level objective model for case b2 (a very similar model can be drawn for case b1).

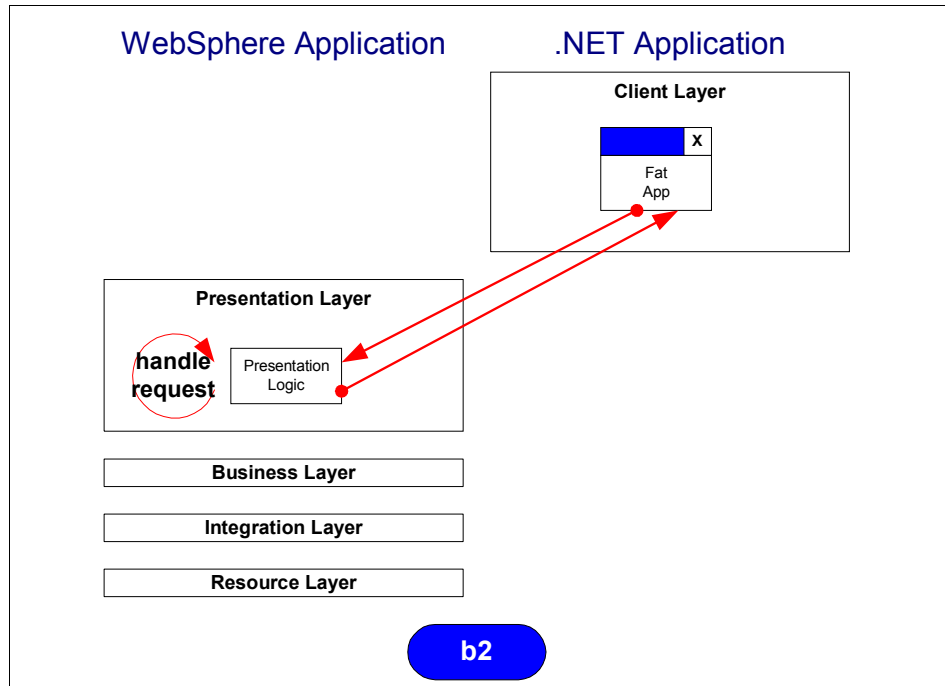


Figure 4-22 A high-level objective model for case b2

Figure 4-23 on page 148 illustrates a candidate solution model for interaction case b2. This model uses a .NET service proxy to present a business style interface (iConfig) to the client, and abstracts the integration solution from the client. In this case, the Presentation layer logic hosted by WebSphere's Web container, which exposes the Servlet as a URL over HTTP. Consequently, we have no need for a stub implementation (the Web container can be considered a generic HTTP stub capable of binding to any (many) hosted URLs).

A very similar candidate solution model can be drawn for interaction case b2.

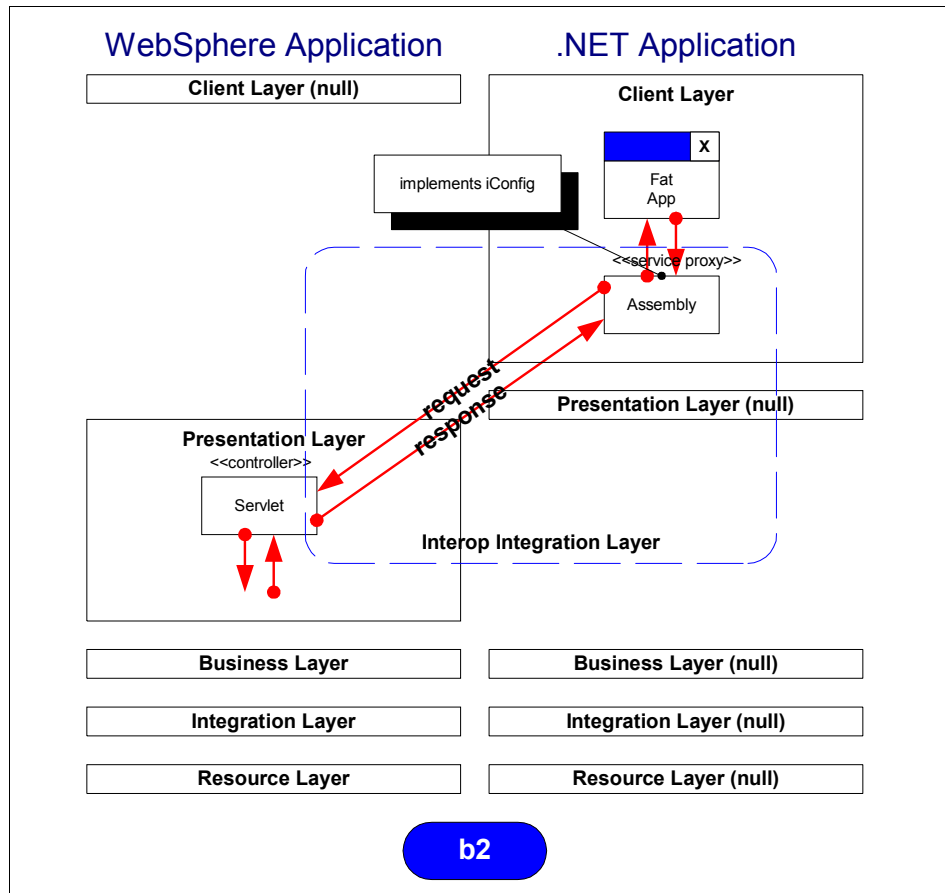


Figure 4-23 Interaction Case b1: candidate solution model (a very similar model can be drawn for interaction case b2)

For thin client applications communicating over HTTP, interaction cases b1 and b2 initially seem like “business as usual.” It is certainly true to say that a Java Presentation layer will have no problem serving ActiveX Controls or VBScript for a .NET thin client application, and that a .NET Presentation layer will have no problems serving Applets or Java Script for a Java thin client application.

Potential challenges arise when view renderings from both a Java Presentation layer and a .NET Presentation layer are needed to coexist with conjoined state dependencies on the same thin client view. This is illustrated in Figure 4-24 on page 149. The challenge for this model is how to share state, via client mediation, between the two Presentation layer subsystems. It may be possible to devise some form of URL extension to pass a limited representation of state between the two systems, but there will almost certainly be no middleware

support. It is strongly recommended that you do *not* do this. For more information about state maintenance between Presentation layers, refer to Chapter 9, “Scenario: Web interoperability” on page 367.

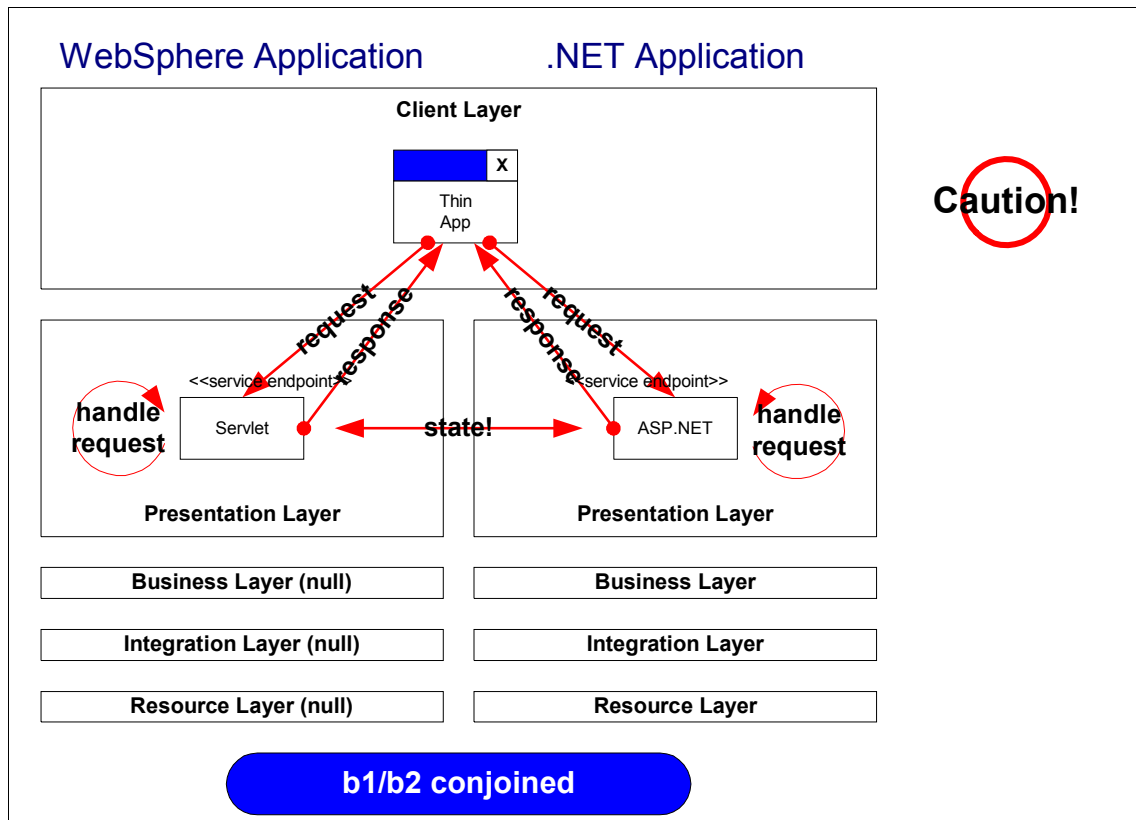


Figure 4-24 Caution: interaction cases b1 and b2 state conjoined Presentation layers

One of the few possible situation where it may be considered appropriate to share thin client view renderings from both Presentation layers is as part of a risk managed strategy for phased migration of a non trivial application from .NET to WebSphere. In this scenario, it is imperative that a thorough analysis of presentation state life cycle be undertaken. The output of this analysis will identify state dependencies, and indicate where migration slice divisions exist.

Figure 4-25 on page 150 illustrates a hypothetical .NET application that has already undergone Business layer migration to WebSphere, and is now having its Presentation layer migrated, slice by slice, to WebSphere. The down side of this approach is that during migration you need runtime support for both Presentation layers.

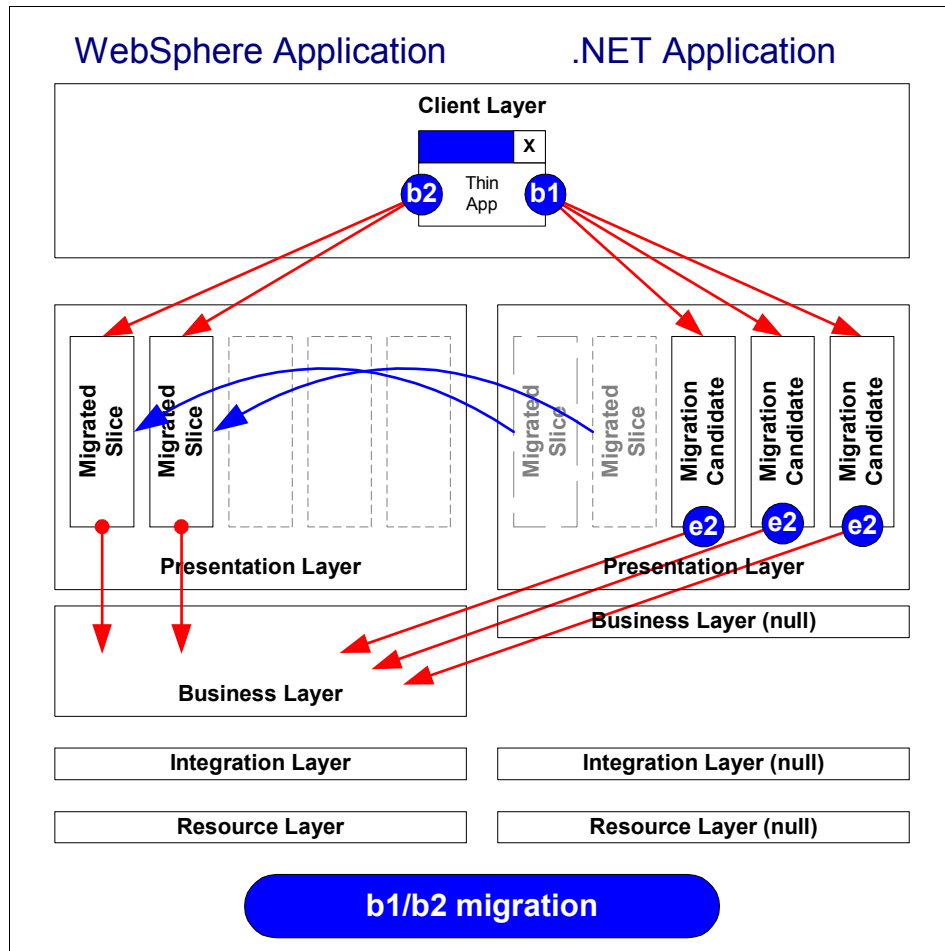


Figure 4-25 Managed phased migration of a Presentation layer from .NET to WebSphere

Interaction case b: component interaction classifications

For each of these interaction perspectives (case b1 and case b2), the interaction between components can be considered to have one of the following integration classifications:

- ▶ *a*: Stateful Synchronous Integration.
- ▶ *b*: Stateless Synchronous Integration.
- ▶ *c*: Stateless Asynchronous Integration.
- ▶ *d*: Stateful Asynchronous Integration (not considered further in this book).

It is possible to further decompose these interactions into finer levels of detail, but these classifications will be adequate for modeling the technical challenges and technical solutions addressed in this book.

This gives us six potential interaction cases to consider:

- ▶ *Case b1.a:* Stateful Synchronous Integration from WebSphere to .NET.
- ▶ *Case b1.b:* Stateless Synchronous Integration from WebSphere to .NET.
- ▶ *Case b1.c:* Stateless Asynchronous Integration from WebSphere to .NET.
- ▶ *Case b2.a:* Stateful Synchronous Integration from .NET to WebSphere.
- ▶ *Case b2.b:* Stateless Synchronous Integration from .NET to WebSphere.
- ▶ *Case b2.c:* Stateless Asynchronous Integration from .NET to WebSphere.

These cases are illustrated in Figure 4-26 and Figure 4-27 on page 152.

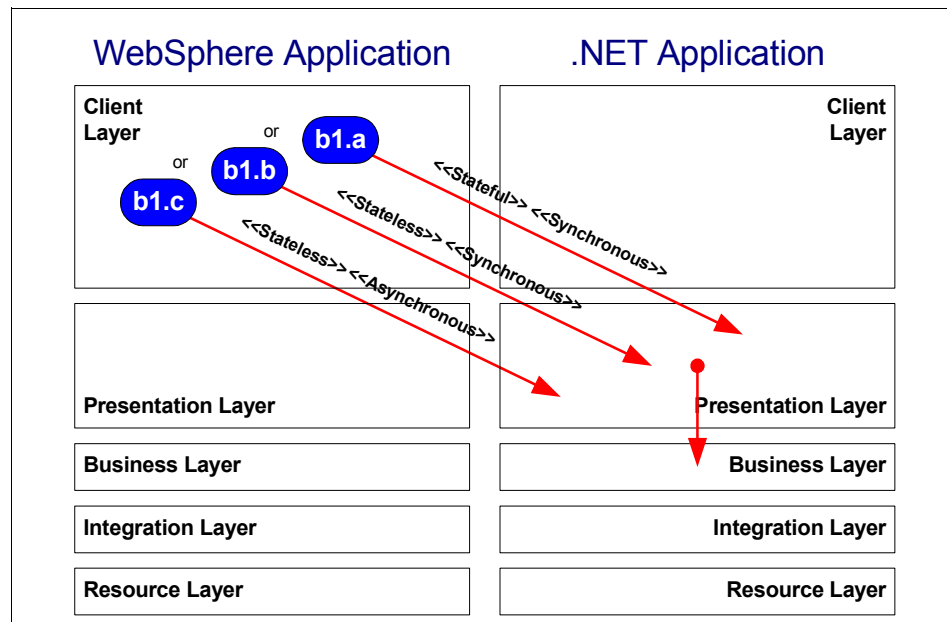


Figure 4-26 Interaction cases b1.a, b1.b and b1.c (WebSphere application's perspective)

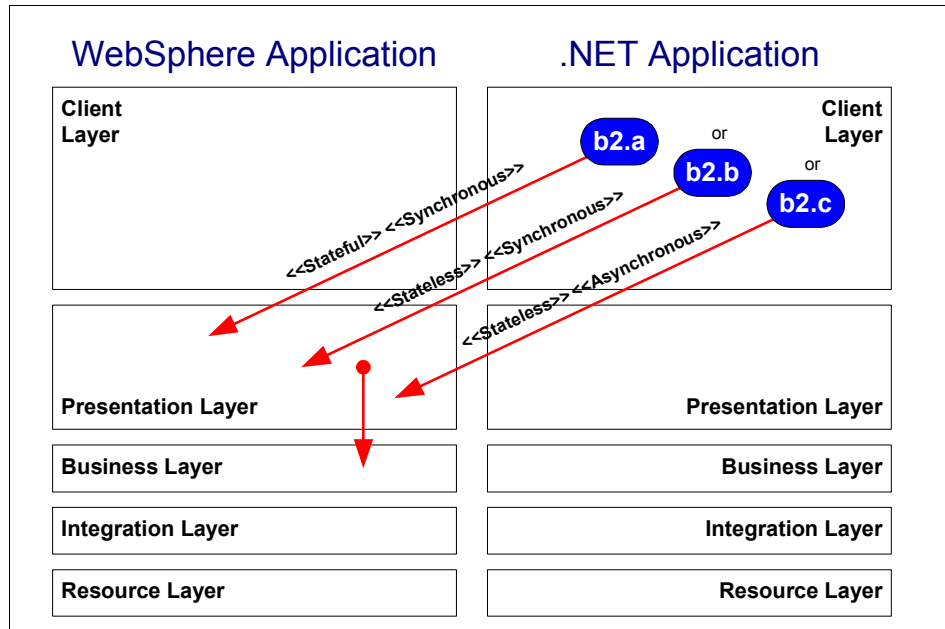


Figure 4-27 Interaction cases b2.a, b2.b and b2.c (.NET application's perspective)

4.4, “Technical solution mapping” on page 202 provides guidance on identifying potential technical solutions for each of these interaction cases (b1.a, b1.b, b1.c, b2.a, b2.b and b2.c).

4.3.3 Interaction case c: client logic to business logic

Let's consider the coexistence of an application deployed in WebSphere and an application deployed in the .NET Framework via Client layer logic to Business layer logic integration (case c). We can consider interaction between these two applications from these perspectives:

► *Perspective c1: WebSphere to .NET*

A runtime artifact executing within a Fat Java Client application, such as a Swing client application or an AWT client application (possibly deployed in the WebSphere Client Container), interacting with a runtime artifact executing within the Business layer of a .NET application deployed in the .NET Framework.

► *Perspective c2: .NET to WebSphere*

A runtime artifact executing within a Fat .NET client application, such as a Windows Form application, interacting with a runtime artifact executing within

the Business layer of a WebSphere application (possibly an EJB or an MDB) deployed in the WebSphere EJB container.

Thin Java Clients and Thin .NET clients communicate directly with the Presentation layer using HTTP. It probably makes little sense to consider thin client to Business layer interactions. It *may* be possible for a thin client with client side logic, such as a Java Applet or an ActiveX control, to interact directly with business logic. However, with this scenario, technical difficulties such as overcoming the restrictions imposed by the browser sandbox may prove challenging.

These perspectives are illustrated in Figure 4-28.

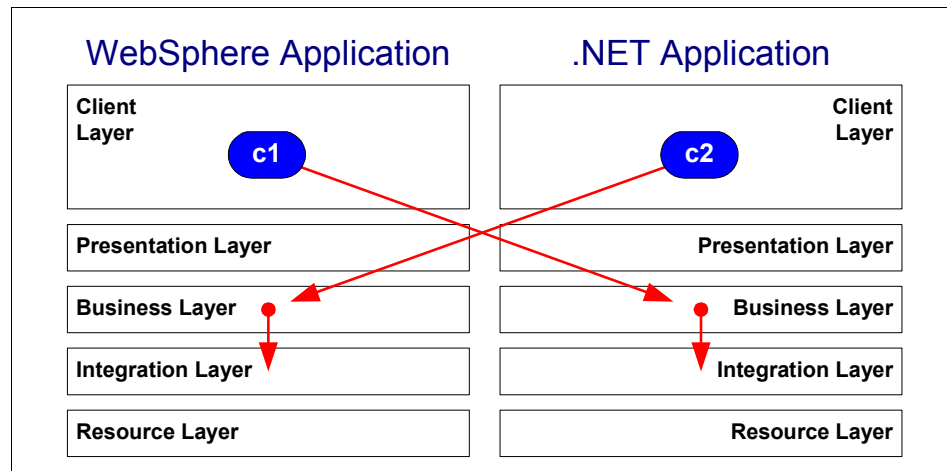


Figure 4-28 interaction cases c1 (WebSphere application's perspective) and c2 (.NET application's perspective)

Interaction case c: structure and dynamic considerations

Consider the scenario where a WebSphere application Business layer exposes some specialized calculator functionality. Say that a .NET client application requires this functionality, and that it is considered an appropriate solution for the .NET client application to invoke the WebSphere applications business functionality. Figure 4-29 on page 154 illustrates a high-level objective model for case c1 (Figure 4-30 on page 155 illustrates an equivalent high-level objective model for case c2).

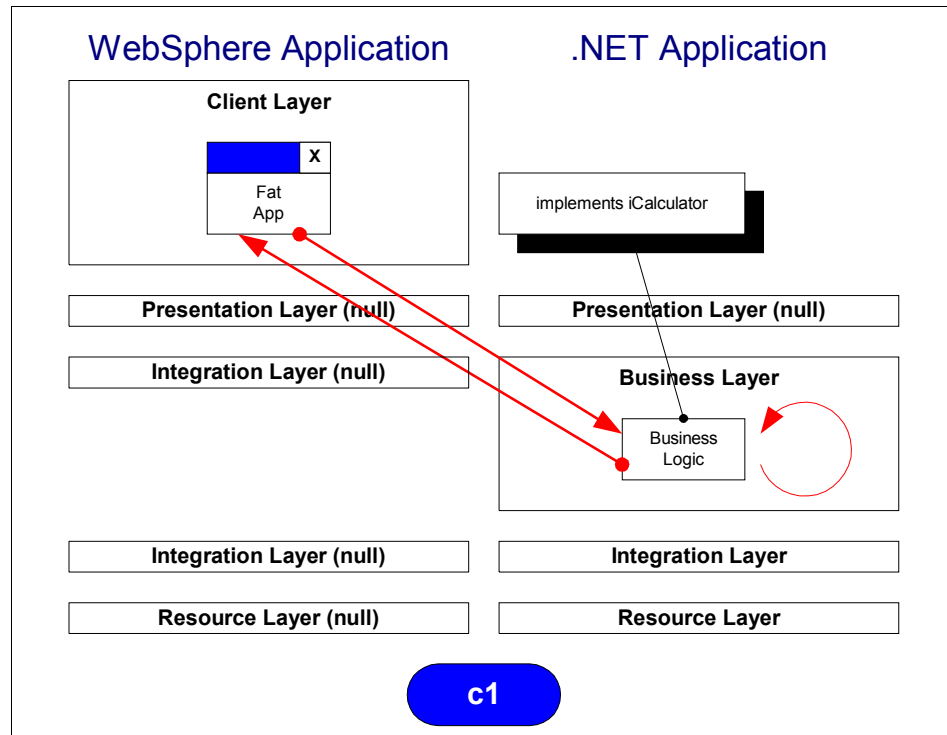


Figure 4-29 A high-level objective model for case c1

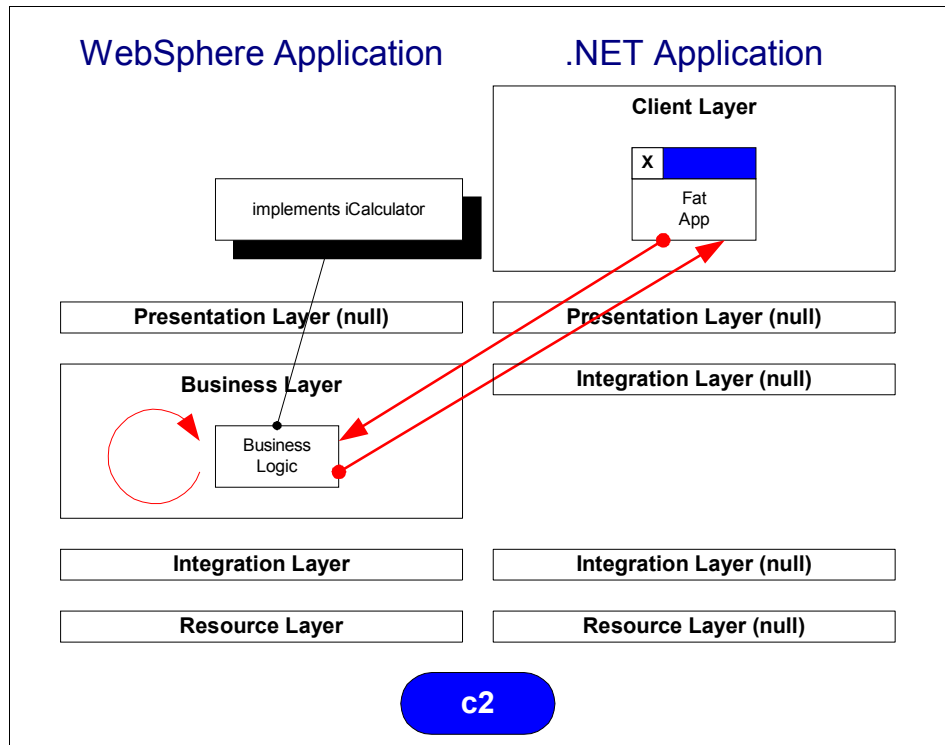


Figure 4-30 A high-level objective model for case c2

Figure 4-31 on page 156 illustrates a candidate solution model for interaction case c1. This model uses a proxy-stub pattern between the fat client and the Business layer code. A Java proxy presents a business interface (`iCalculator`) to the client, and abstracts the integration solution for the client. A .NET stub represents the client to the .NET service implementation, and abstracts the service from the integration solution.

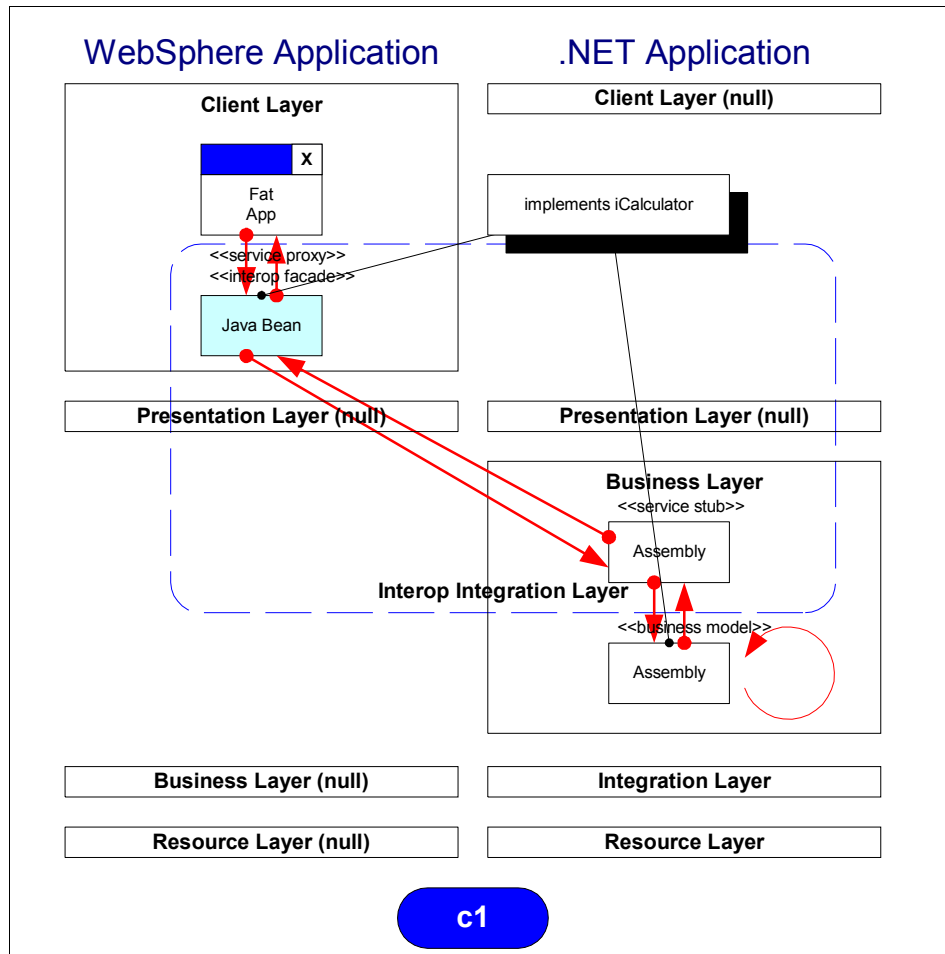


Figure 4-31 Interaction case c1: candidate solution model

Figure 4-32 on page 157 illustrates a candidate solution model for interaction case c2. This model also uses a proxy-stub pattern between the fat client and the Business layer code. A .NET proxy presents a business interface (`ICalculator`) to the client, and abstracts the integration solution for the client. A Java stub represents the client to the service's EJB quality of service (QOS) decorator, which in turn delegates to the Java service implementation.

Definitions:

Decoration: using one object to attach additional responsibilities to another object.

QoS decoration: attaching quality of service responsibilities to another object. In this case, using an EJB to attach declarative qualities of service (potentially: security, object pooling, transactions, workload management).

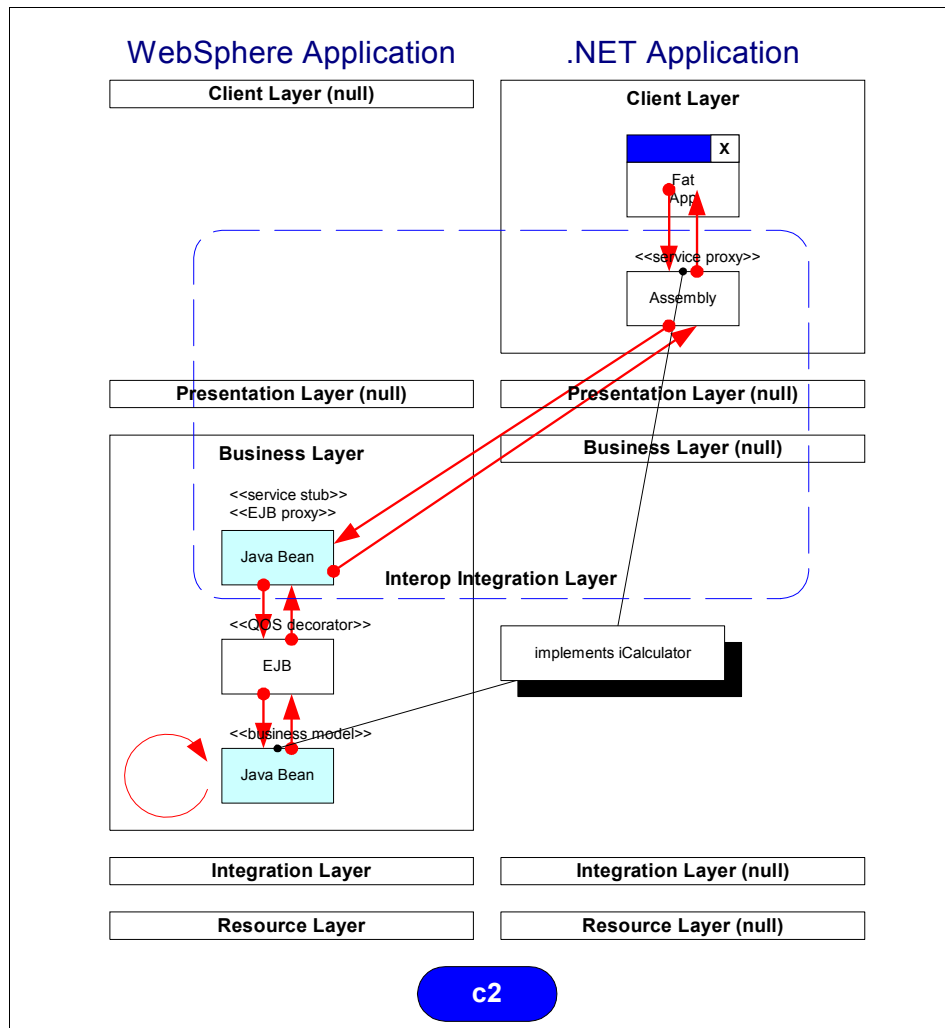


Figure 4-32 Interaction case c2: candidate solution model

Interaction case c: component interaction classifications

For each of these interaction perspectives (case c1 and case c2), the interaction between components can be considered to have one of the following integration classifications:

- ▶ *a*: Stateful Synchronous Integration.
- ▶ *b*: Stateless Synchronous Integration.
- ▶ *c*: Stateless Asynchronous Integration.
- ▶ *d*: Stateful Asynchronous Integration (not considered further in this book).

It is possible to further decompose these interactions into finer levels of detail, but these classifications will be adequate for modeling the technical challenges and technical solutions addressed in this book.

This give us six potential interaction cases to consider:

- ▶ *Case c1.a*: Stateful Synchronous Integration from WebSphere to .NET.
- ▶ *Case c1.b*: Stateless Synchronous Integration from WebSphere to .NET.
- ▶ *Case c1.c*: Stateless Asynchronous Integration from WebSphere to .NET.
- ▶ *Case c2.a*: Stateful Synchronous Integration from .NET to WebSphere.
- ▶ *Case c2.b*: Stateless Synchronous Integration from .NET to WebSphere.
- ▶ *Case c2.c*: Stateless Asynchronous Integration from .NET to WebSphere.

These cases are illustrated in Figure 4-33 on page 159 and Figure 4-34 on page 159.

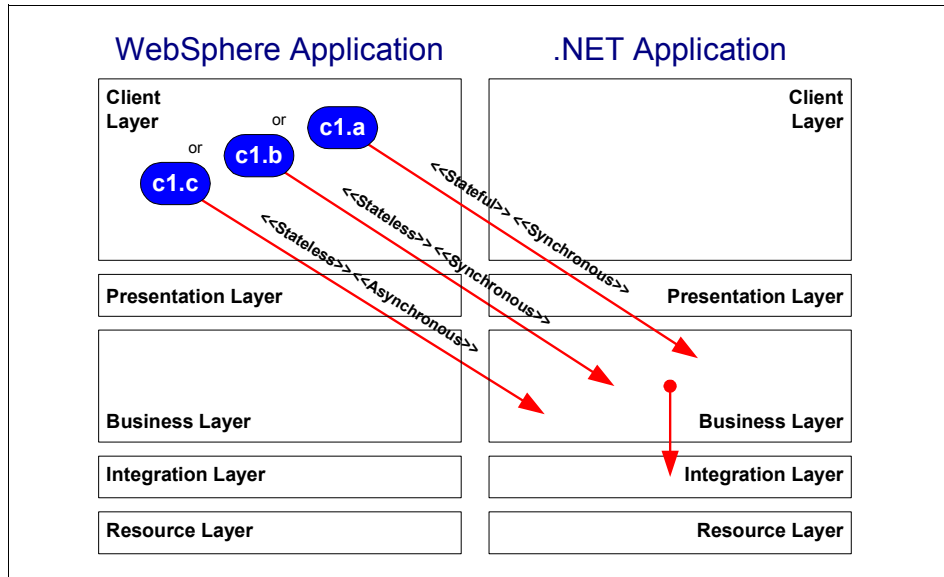


Figure 4-33 Interaction cases c1.a, c1.b and c1.c (WebSphere application's perspective)

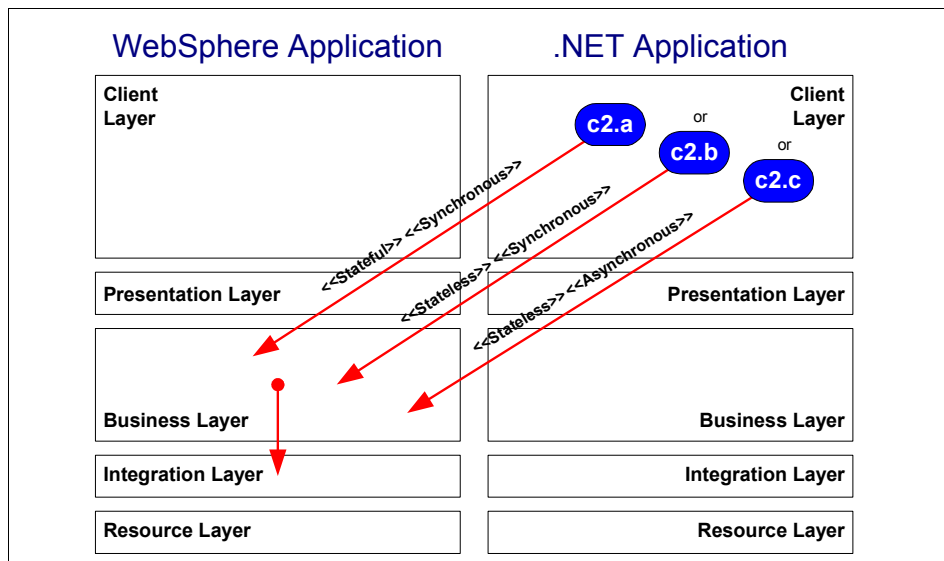


Figure 4-34 Interaction cases c2.a, c2.b and c2.c (.NET application's perspective)

4.4, “Technical solution mapping” on page 202 provides guidance on identifying potential technical solutions for each of these interaction cases (c1.a, c1.b, c1.c, c2.a, c2.b and c2.c).

4.3.4 Interaction case d: presentation logic to presentation logic

Let's consider the coexistence of an application deployed in WebSphere and an application deployed in the .NET Framework via Presentation layer logic to Presentation layer logic integration (case d). We can consider interaction between these two applications from these perspectives:

► *Perspective d1: WebSphere to .NET*

A runtime artifact executing within the Presentation layer of a WebSphere application (possibly a JSP, servlet, or an underpinning Java Bean) deployed in the WebSphere Web container, interacting with a runtime artifact executing within the Presentation layer of a .NET application (possibly an ASP.NET artifact) deployed in IIS.

► *Perspective d2: .NET to WebSphere*

A runtime artifact executing within the Presentation layer of a .NET application (possibly an ASP.NET or an underpinning assembly) deployed in IIS, interacting with a runtime artifact executing within the Presentation layer of a WebSphere application (possibly a JSP, or a servlet) deployed in WebSphere Web Container.

These perspectives are illustrated in Figure 4-35.

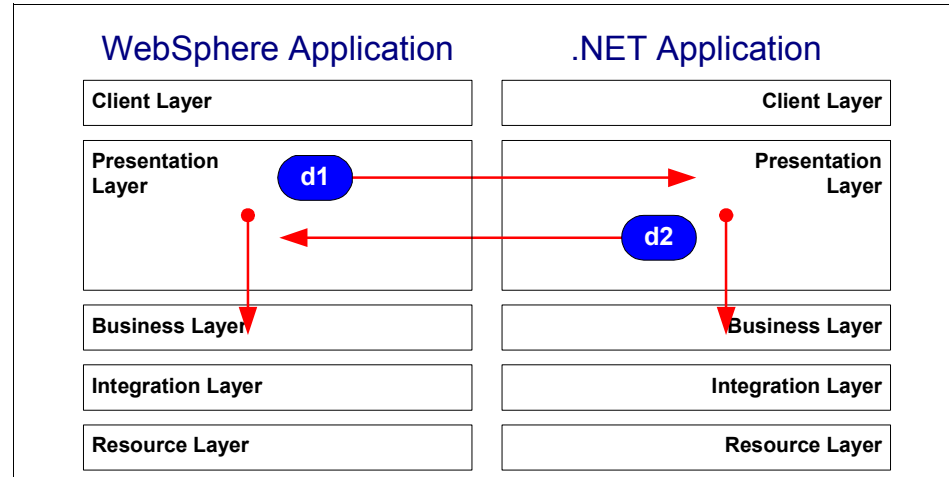


Figure 4-35 Interaction cases d1 (WebSphere application's perspective) and d2 (.NET application's perspective)

Interaction case d: structure and dynamic considerations

In this case, the presentation logic from WebSphere interacts with the .NET presentation logic, and vice-versa. The Presentation layer here represents the

Web application layer; note that the client is called thin client in both the d1 and d2, cases. This interaction model is a bit more complex than other cases; not only do the Web applications interact with each other, but the user can be involved in the communication.

Definition: The *Model-View-Controller* (MVC) pattern allows an abstraction of the components of the user interface (GUI) to support easy development, maintenance, and extensibility. The *model* is the program itself, the *view* represents the input and the output of the program, the *controller* links the view and the model.

Case d1 represents the situation where the client accesses the WebSphere Web application and WebSphere interacts with the .NET Web application in order to present the response.

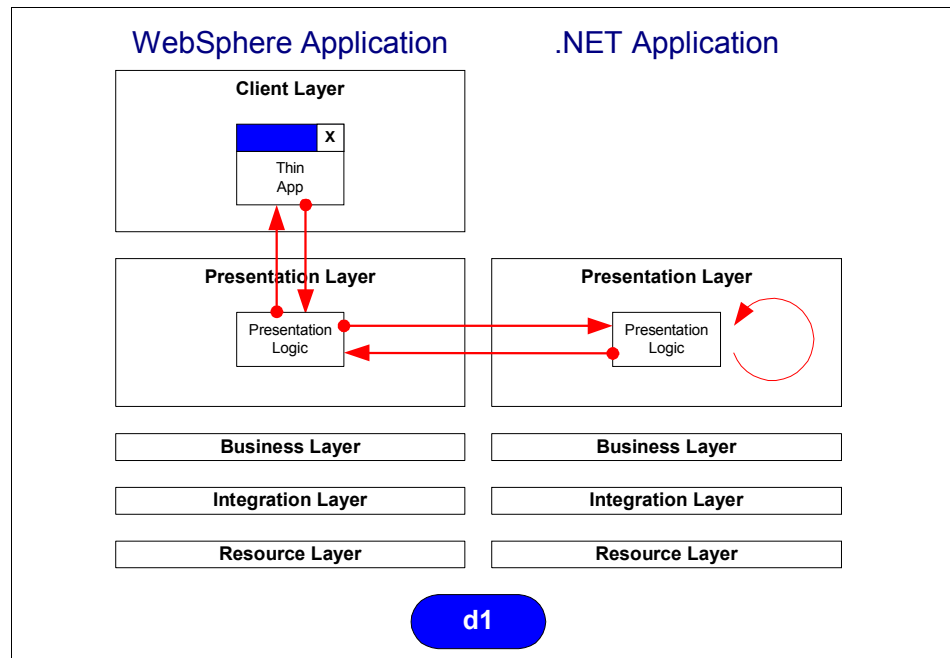


Figure 4-36 A high-level objective model for case d1

Case d2 represents the situation where the client accesses the .NET Web application and .NET interacts with the WebSphere Web application in order to present the response.

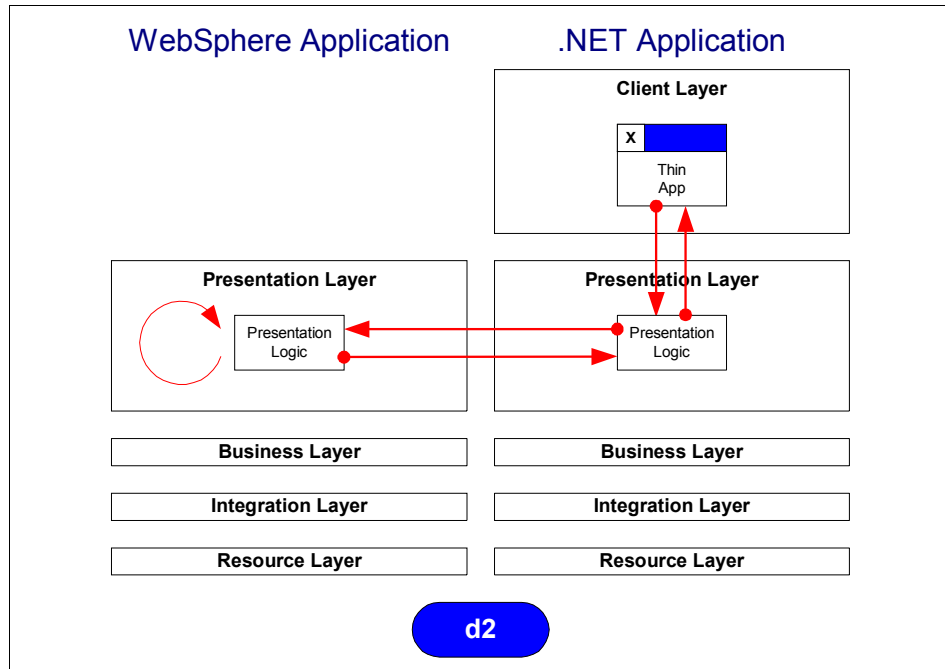


Figure 4-37 A high-level objective model for case d2

Potential value:

- ▶ This solution model should enable migration of an application Business layer code to WebSphere, allowing Presentation layer code to be migrated to WebSphere in a subsequent phase.

Potential downsides:

- ▶ Both Microsoft IIS and the WebSphere Web Container need to be deployed (until the .NET Presentation layer is migrated to WebSphere).

Expected issues with this solution model are:

- ▶ Sharing content (mostly static content).
- ▶ Session maintenance.
- ▶ Session state representation.
- ▶ Security between WebSphere and .NET.

The following interaction model is exercising redirection between Web applications. The user accesses a Web application on one platform and the response arrives from another Web application on the other platform. Redirection involves further client interactions. The request does not flow directly from one

application to the other but it goes back to the client application with a special redirection response.

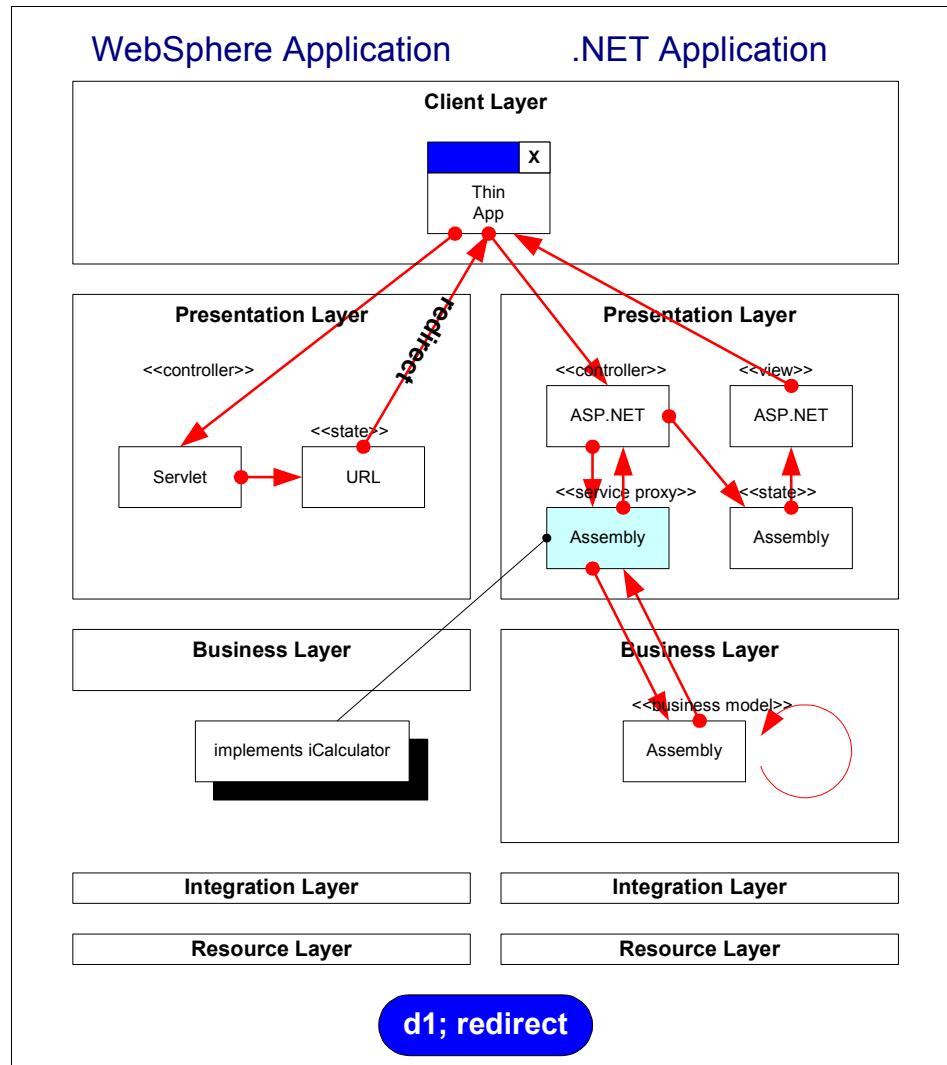


Figure 4-38 Interaction case d1: candidate solution model

In the alternative solution, both applications are involved in the process as opposed to the original solution where one application just redirects based on the requested URL.

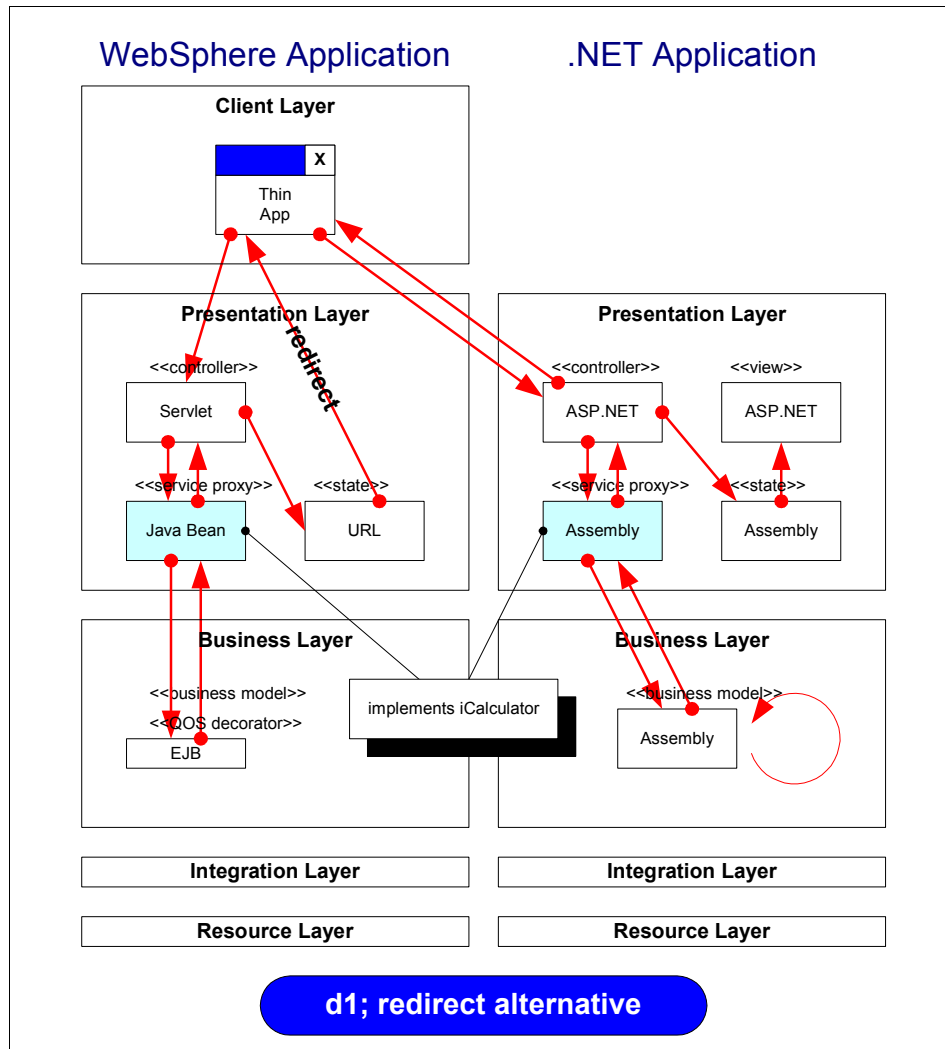


Figure 4-39 Interaction case d1: candidate solution model

Note: The other interaction alternative, where the applications forward the request from one Web application on one platform to the other application on the other platform, is not valid. Forwarding between two different platforms is not supported by either platforms.

Interaction case d: component interaction classifications

For each of these interaction perspectives (case d1 and case d2), the interaction between components can be considered to have one of the following integration classifications:

- ▶ *a*: Stateful Synchronous Integration.
- ▶ *b*: Stateless Synchronous Integration.
- ▶ *c*: Stateless Asynchronous Integration.
- ▶ *d*: Stateful Asynchronous Integration (not considered further in this book).

It is possible to further decompose these interactions into finer levels of detail, but these classifications will be adequate for modeling the technical challenges and technical solutions addressed in this book.

This give us six potential interaction cases to consider:

- ▶ *Case d1.a*: Stateful Synchronous Integration from WebSphere to .NET.
- ▶ *Case d1.b*: Stateless Synchronous Integration from WebSphere to .NET.
- ▶ *Case d1.c*: Stateless Asynchronous Integration from WebSphere to .NET.
- ▶ *Case d2.a*: Stateful Synchronous Integration from .NET to WebSphere.
- ▶ *Case d2.b*: Stateless Synchronous Integration from .NET to WebSphere.
- ▶ *Case d2.c*: Stateless Asynchronous Integration from .NET to WebSphere.

These cases are illustrated in Figure 4-40 and Figure 4-41 on page 166.

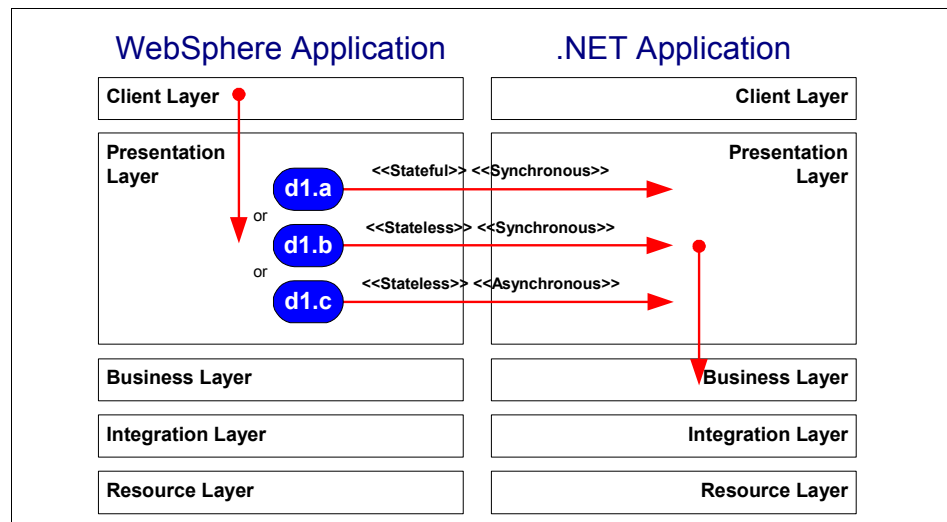


Figure 4-40 Interaction cases d1.a, d1.b and d1.c (WebSphere application's perspective)

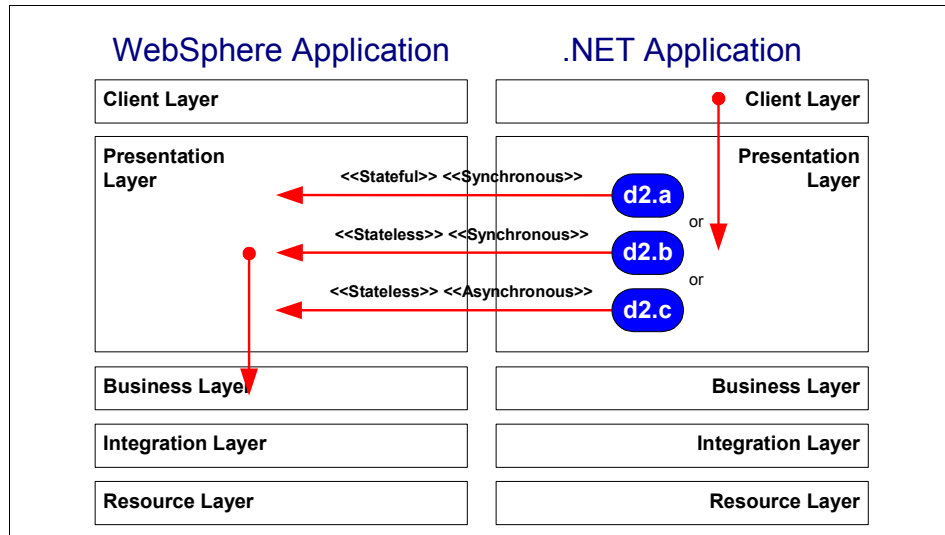


Figure 4-41 Interaction cases d2.a, d2.b and d2.c (.NET application's perspective)

4.4, "Technical solution mapping" on page 202 provides guidance on identifying potential technical solutions for each of these interaction cases (d1.a, d1.b, d1.c, d2.a, d2.b and d2.c).

4.3.5 Interaction case e: presentation logic to business logic

Let's consider the coexistence of an application deployed in WebSphere and an application deployed in the .NET Framework via Presentation layer logic to Business layer logic integration (case e). We can consider interaction between these two applications from these perspectives:

► *Perspective e1: WebSphere to .NET*

A runtime artifact executing within the Presentation layer of a WebSphere application (possibly a JSP, servlet, or an underpinning Java Bean) deployed in the WebSphere Web Container, interacting with a runtime artifact executing within the Business layer of a .NET application (possibly an assembly or a COM+ component) deployed in the .NET Framework.

► *Perspective e2: .NET to WebSphere*

A runtime artifact executing within the Presentation layer of a .NET application (possibly an ASP.NET artifact or an underpinning assembly) deployed in IIS, interacting with a runtime artifact executing within the Business layer of a WebSphere application (possibly an EJB or MDB) deployed in WebSphere EJB Container.

These perspectives are illustrated in Figure 4-42.

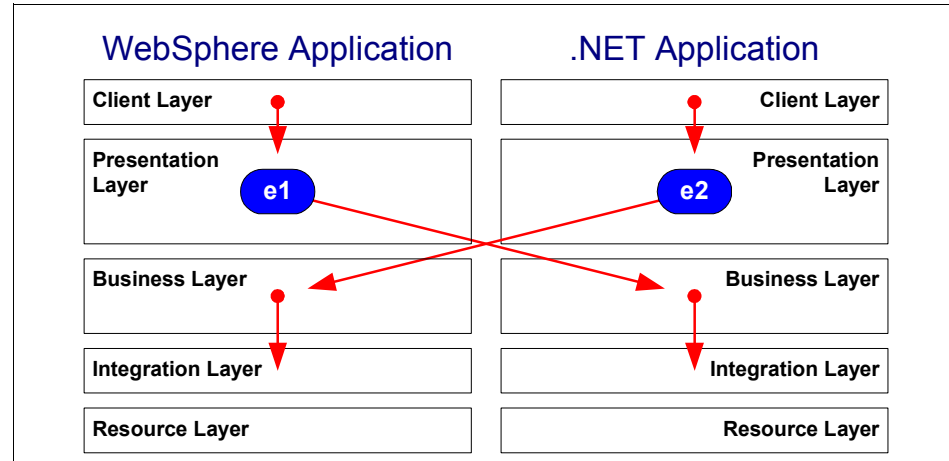


Figure 4-42 Interaction cases e1 (WebSphere application's perspective) and e2 (.NET application's perspective)

Interaction case e: structure and dynamic considerations

Consider the scenario where a .NET application Business layer exposes some specialized calculator functionality. Say that a WebSphere application

Presentation layer requires this functionality, and that it is considered an appropriate solution for the WebSphere application to invoke the .NET applications business functionality. Figure 4-43 illustrates a high-level objective model for case e1 (the WebSphere calling application's perspective).

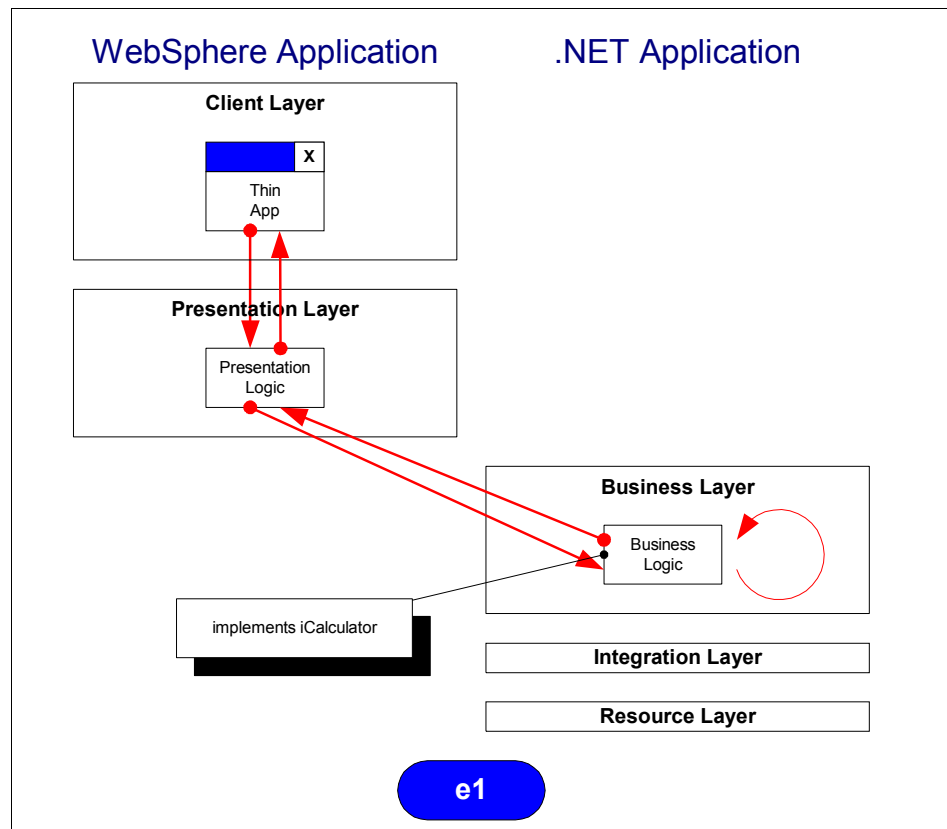


Figure 4-43 A high-level objective model for case e1

Figure 4-44 on page 169 illustrates a high-level objective model for case e2 (the .NET calling application's perspective).

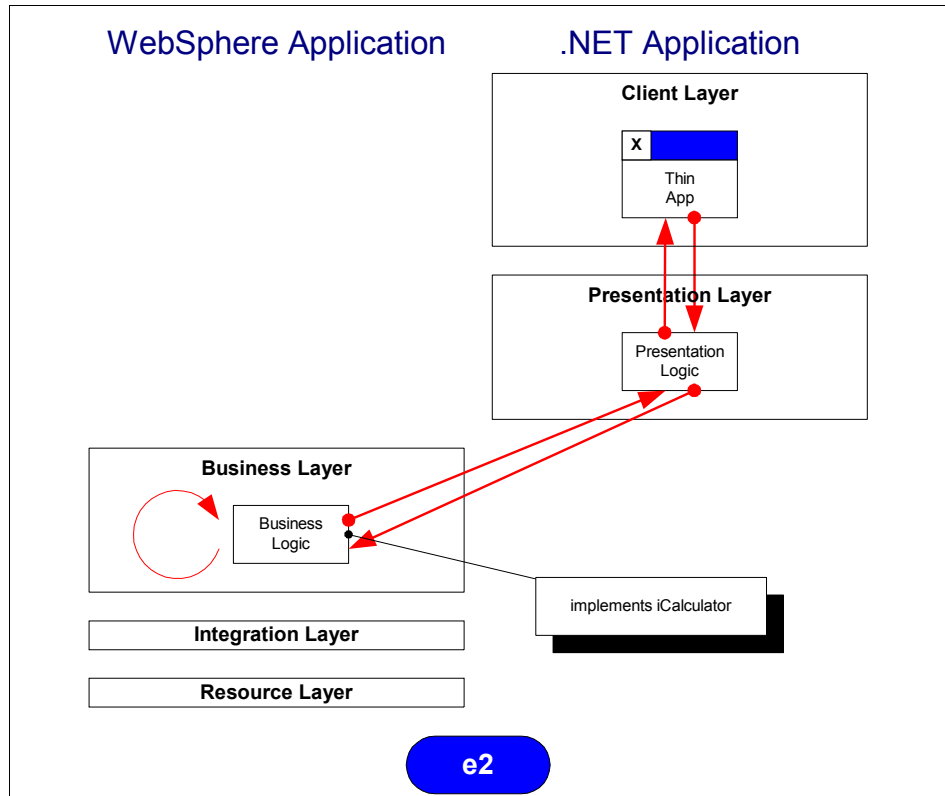


Figure 4-44 A high-level objective model for case e2

The model illustrated in Figure 4-45 on page 170 presents a candidate solution for interaction case e1. This model combines a *model-view-controller* (MVC) and a *proxy-stub* pattern. The MVC pattern provides the structure and separation of concerns for the Presentation layer of the WebSphere application. The proxy-stub pattern provides the structure, separation of concerns and abstraction for the integration solution. The Java proxy presents a business interface (*iCalculator*) to the MVC *controller* (Java Servlet). The .NET stub represents the client to the .NET service implementation, and abstracts the service from the integration solution.

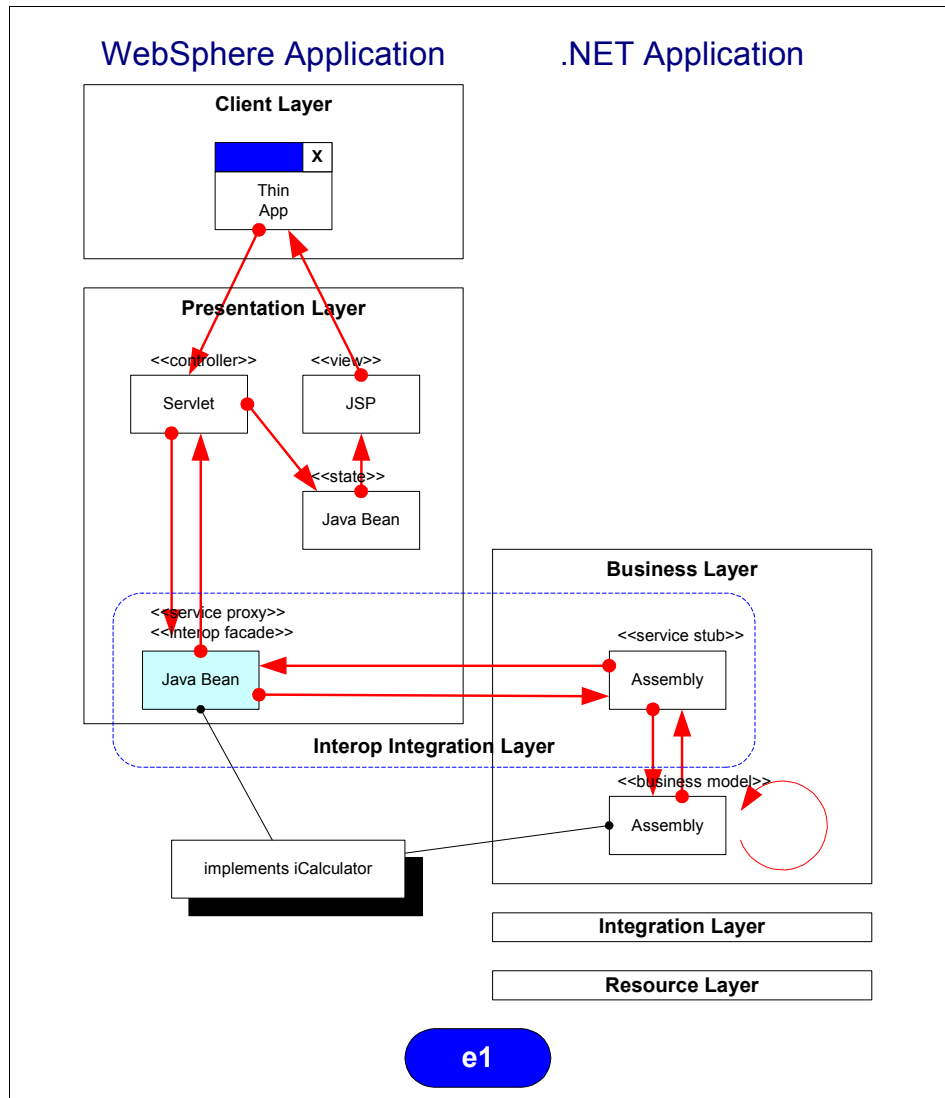


Figure 4-45 Interaction case e1: candidate solution model

The solution model illustrated in Figure 4-46 on page 171 extends the model illustrated in Figure 4-45 by pushing integration down into the Business layer to give a Business layer to Business layer interaction scenario (case f). This model uses an EJB as a quality of service (QOS) decorator for the service proxy.

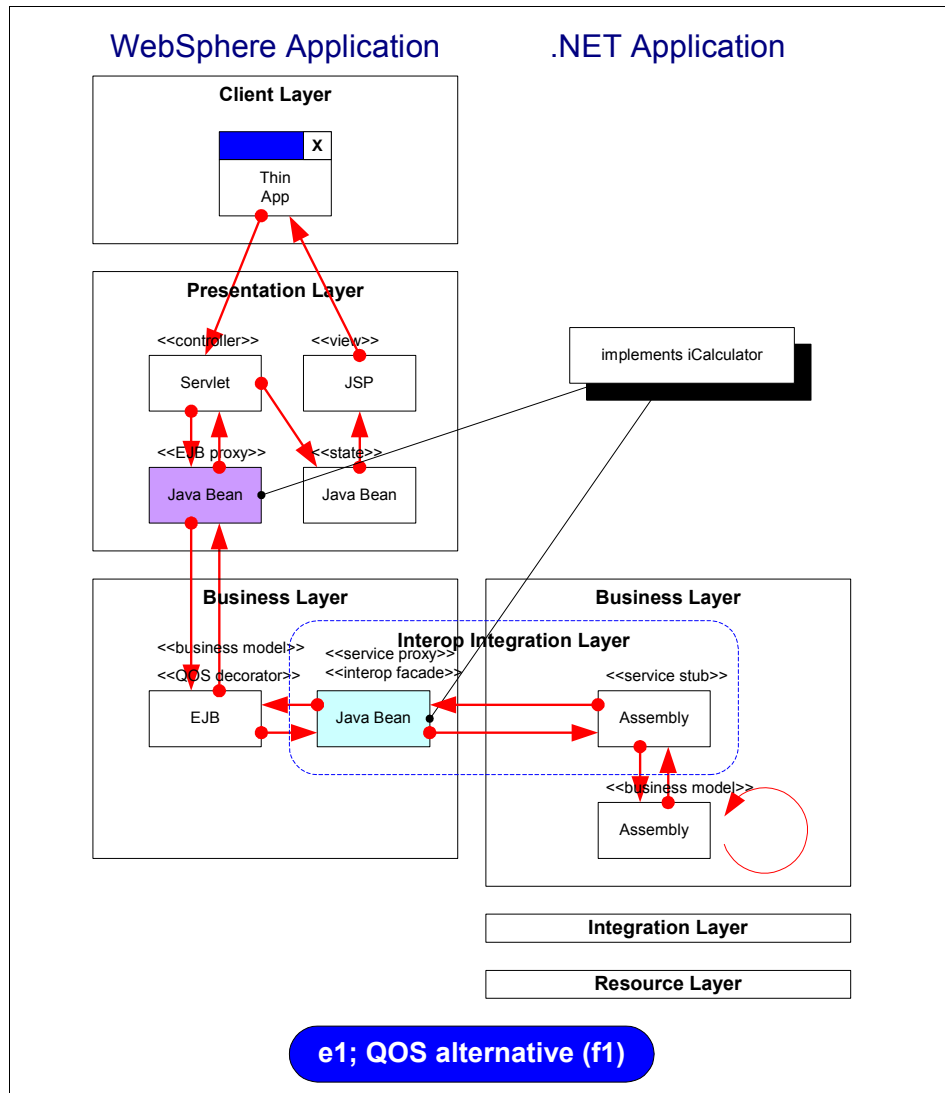
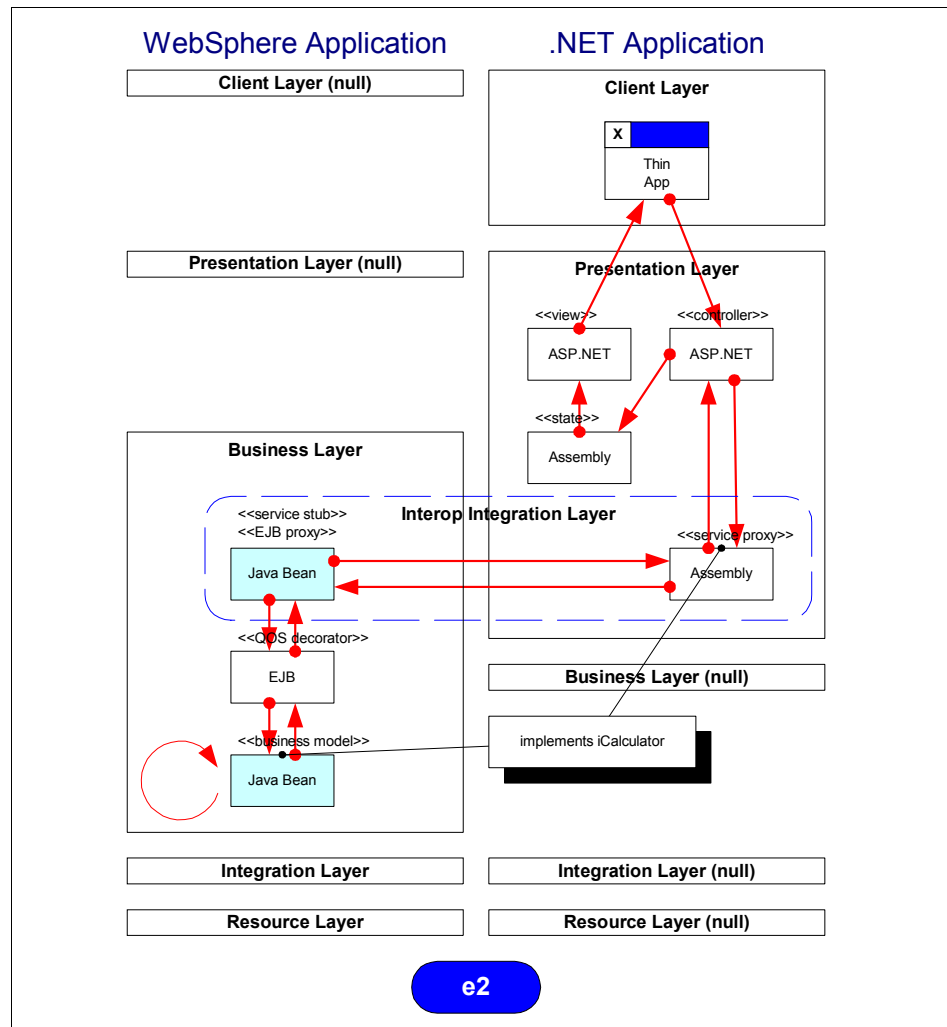


Figure 4-46 Interaction case e1 (f1); candidate solution model

Figure 4-47 on page 173 illustrates a candidate solution model for interaction case e2. As with the model presented in Figure 4-45 on page 170T, this also model combines a *model-view-controller* (MVC) and a *proxy-stub* pattern. The MVC pattern provides the structure and separation of concerns for the Presentation layer of the .NET application. The proxy-stub pattern provides the structure, separation of concerns and abstraction for the integration solution. The .NET proxy presents a business interface (*iCalculator*) to the MVC *controller*

(ASP.NET). The Java stub represents the client to the service's EJB quality of service (QOS) decorator, which in turn delegates to the Java service implementation.



Interaction case e: component interaction classifications

For each of these interaction perspectives (case e1 and case e2), the interaction between components can be considered to have one of the following integration classifications:

- ▶ *a*: Stateful Synchronous Integration.
- ▶ *b*: Stateless Synchronous Integration.

- *c*: Stateless Asynchronous Integration.
- *d*: Stateful Asynchronous Integration (not considered further in this book).

It is possible to further decompose these interactions into finer levels of detail, but these classifications will be adequate for modeling the technical challenges and technical solutions addressed in this book.

This give us six potential interaction cases to consider:

- *Case e1.a*: Stateful Synchronous Integration from WebSphere to .NET.
- *Case e1.b*: Stateless Synchronous Integration from WebSphere to .NET.
- *Case e1.c*: Stateless Asynchronous Integration from WebSphere to .NET.
- *Case e2.a*: Stateful Synchronous Integration from .NET to WebSphere.
- *Case e2.b*: Stateless Synchronous Integration from .NET to WebSphere.
- *Case e2.c*: Stateless Asynchronous Integration from .NET to WebSphere.

These cases are illustrated in Figure 4-47 and Figure 4-48 on page 174.

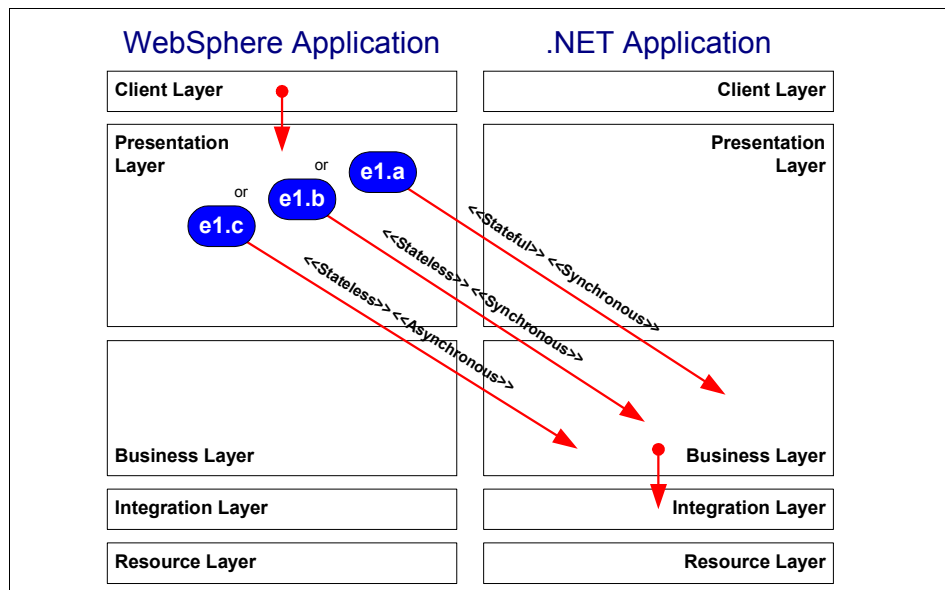


Figure 4-47 Interaction cases e1.a, e1.b and e1.c (WebSphere application's perspective)

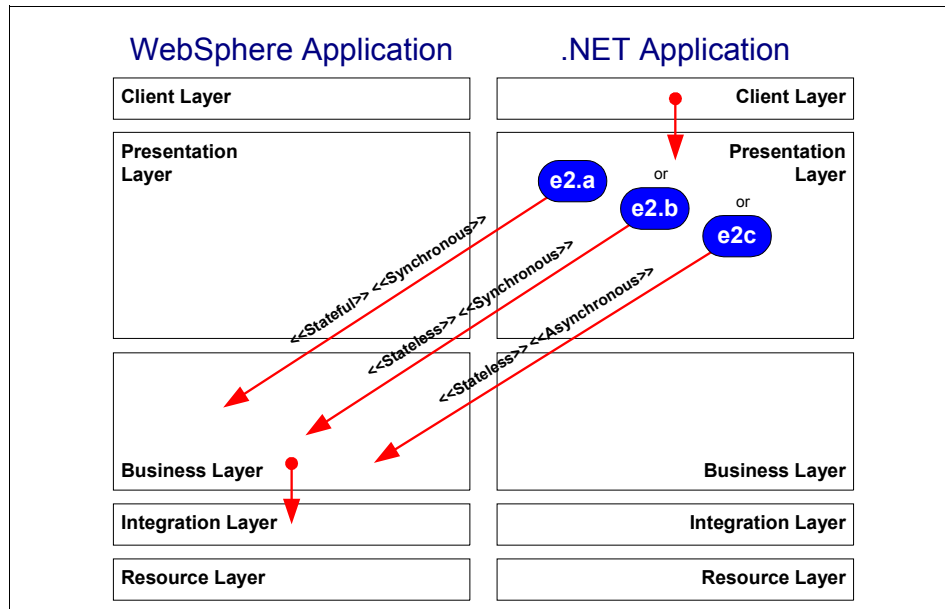


Figure 4-48 Interaction cases e2.a, e2.b and e2.c (.NET application's perspective)

4.4, “Technical solution mapping” on page 202 provides guidance for identifying potential technical solutions for each of these interaction cases (e1.a, e1.b, e1.c, e2.a, e2.b and e2.c).

4.3.6 Interaction case f: business logic to business logic

Let’s consider the coexistence of an application deployed in WebSphere and an application deployed in .NET via Business layer logic to Business layer logic integration (case f). We can consider interaction between these two applications from these perspectives:

► *Perspective f1: WebSphere to .NET*

A runtime artifact executing within the Business layer of a WebSphere application (possibly an EJB or underpinning Java Bean) deployed within a the WebSphere EJB Container, interacting with a runtime artifact executing within the Business layer of a .NET application (possibly an assembly or a COM+ component) deployed in the .NET Framework.

► *Perspective f2: .NET to WebSphere*

A runtime artifact executing within the Business layer of a .NET application (possibly an assembly or a COM+ component) deployed within the .NET Framework, interacting with a runtime artifact in the Business layer of a

WebSphere application (possibly an EJB or MDB) deployed in WebSphere EJB Container.

These two perspectives are illustrated in Figure 4-49.

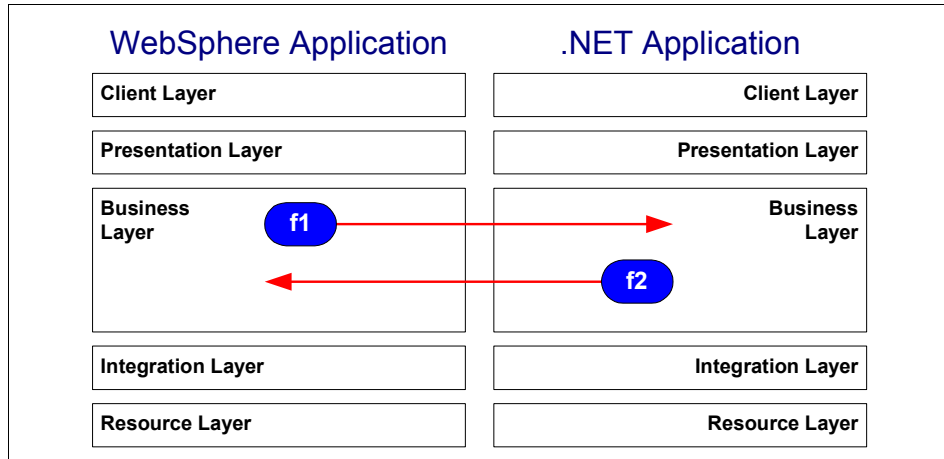


Figure 4-49 interaction cases f1 (WebSphere application's perspective) and f2 (.NET application's perspective)

Interaction case f: structure and dynamic considerations

Consider the scenario where a WebSphere application's Business layer exposes some specialized calculator functionality. Say that a .NET application Presentation layer requires this functionality, and that it is considered an appropriate solution for the WebSphere application to invoke the .NET application's business functionality. Figure 4-50 on page 176 illustrates a high-level objective model for case f2 (the .NET calling application's perspective). Figure 4-51 on page 177 illustrates a high-level objective model for case f1 (the WebSphere calling application's perspective).

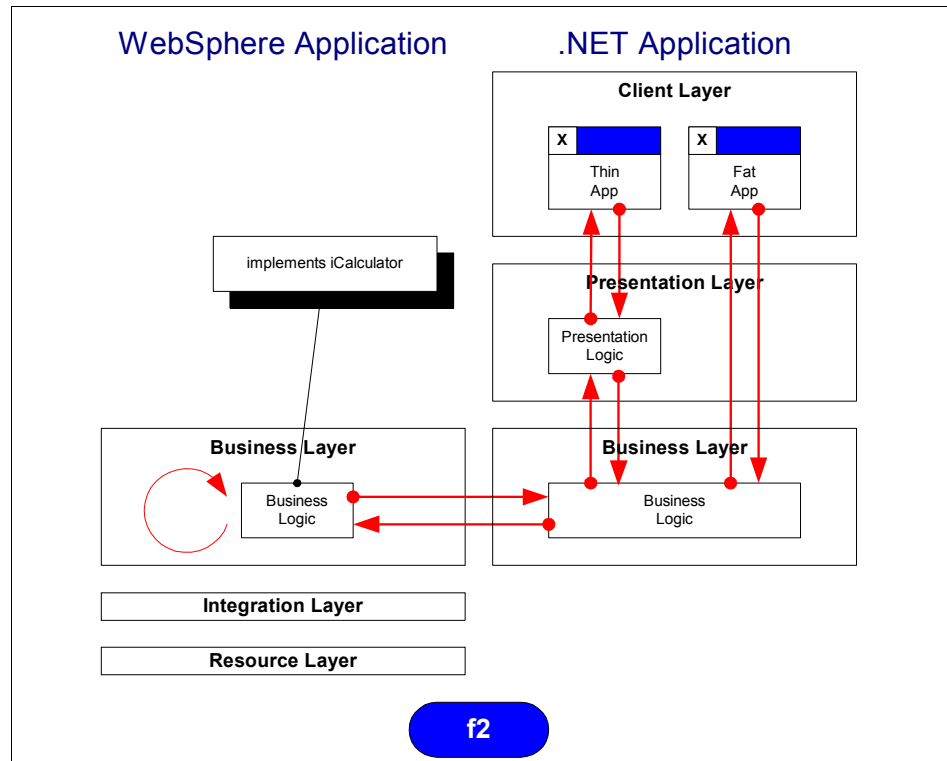


Figure 4-50 A high-level objective model for case f2

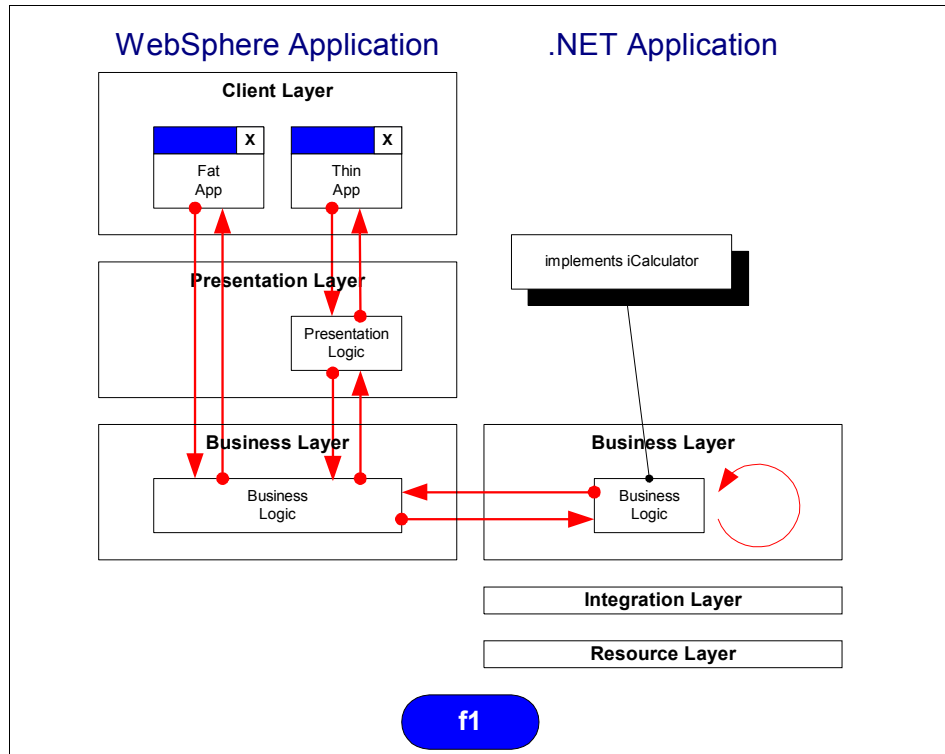


Figure 4-51 A high-level objective model for case f1

Figure 4-52 on page 178 illustrates a candidate solution model for interaction case f2. This uses a proxy-stub pattern to provide structure, separation of concerns, and abstraction for the integration solution. The .NET proxy presents a business interface (*iCalculator*) to an adjacent business tier artifact. The Java stub in the WebSphere application presents the client to the service's EJB quality of service (QOS) decorator, which in turn delegates to the Java service implementation.

For completeness, this model also illustrates both a fat client application binding to the .NET business model, and thin client binding to the same (logical) .NET assembly via its Presentation layer MCV controller ASP.NET.

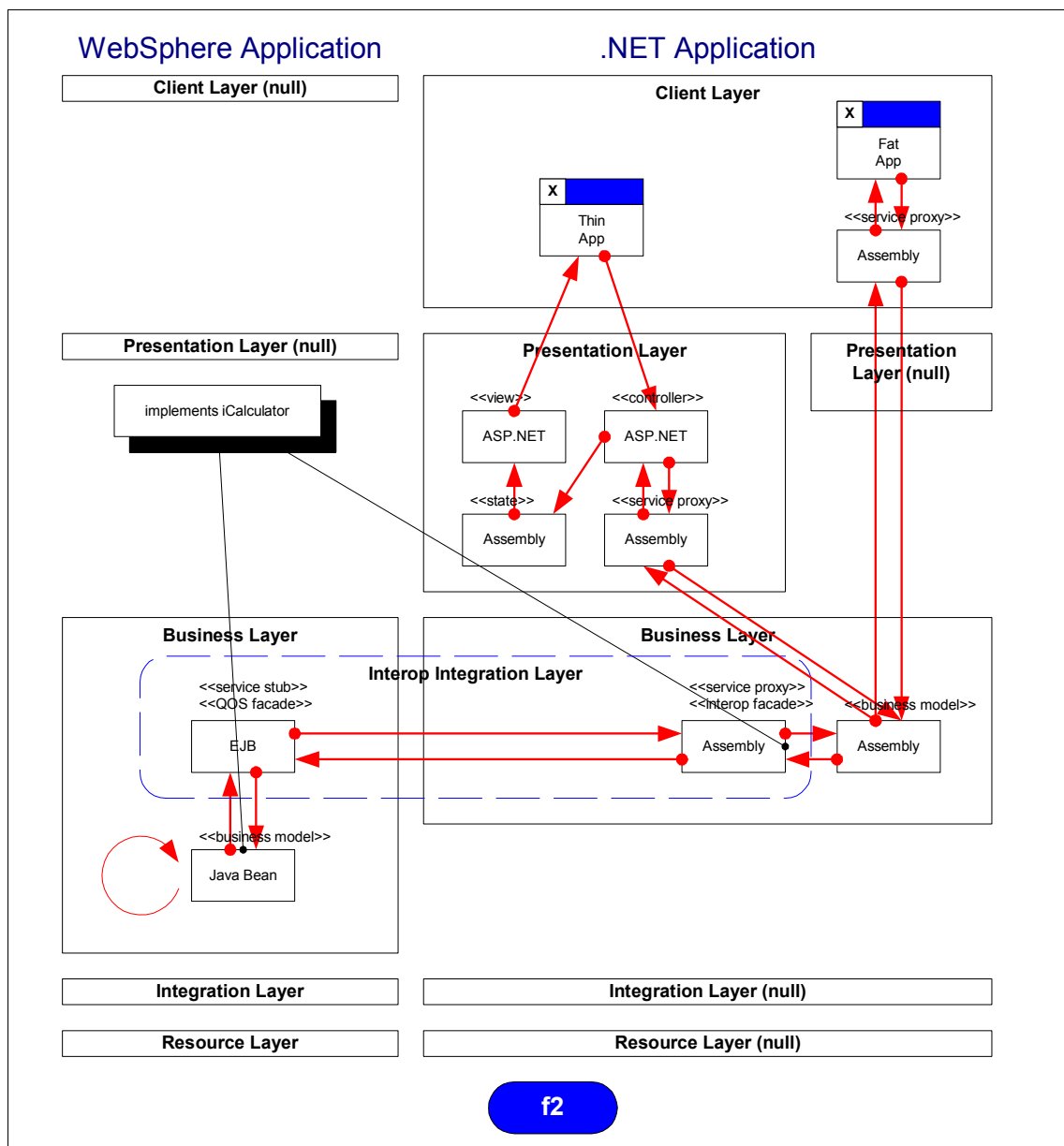


Figure 4-53 on page 180 illustrates a candidate solution model for interaction case f1. This uses a proxy-stub pattern to provide structure, separation of concerns, and abstraction for the integration solution. The Java proxy presents a

business interface (iCalculator) to an adjacent Business layer artifact. The .NET stub represents the client to the .NET service implementation.

For completeness, this model also illustrates both a fat client application binding to a business model EJB, and a thin client binding to the same business model EJB via its Presentation layer MCV controller servlet. Notice that two instances of the same client side Java proxy implementation (as opposed to the single interop proxy) are illustrated. Each Java proxy is deployed adjacent to its client (that is, the Client Tier for the fat client, and the Presentation Tier for the thin client). Both of these proxies bind to the same service EJB (this assumes the same integration mechanism, such as RMI/IIOP between the WebSphere application layers).

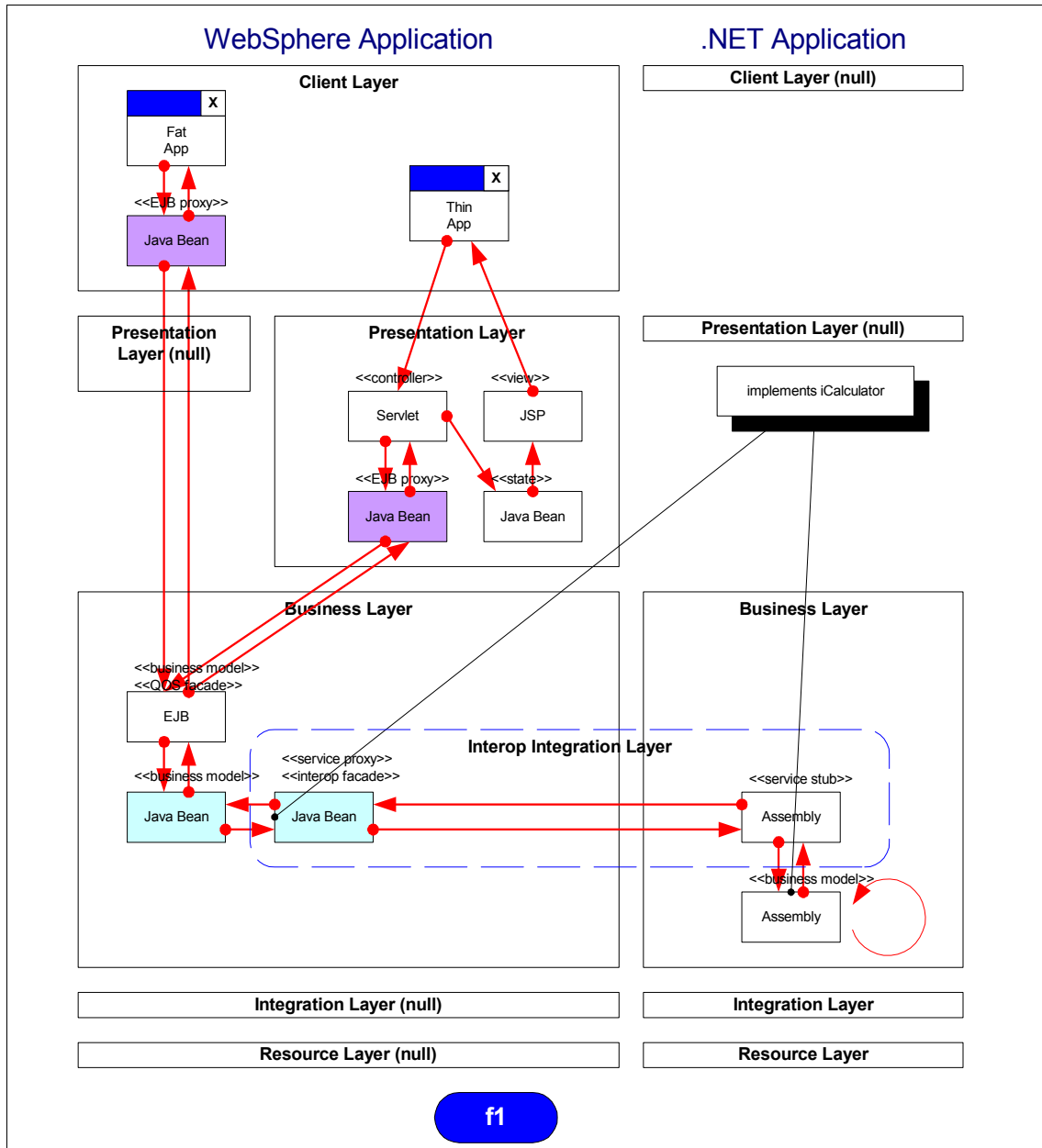


Figure 4-53 Interaction case f1: candidate solution model

Interaction case f: component interaction classifications

For each of these interaction perspectives (case f1 and case f2), the interaction between components can be considered to have one of the following integration classifications:

- ▶ *a*: Stateful Synchronous Integration.
- ▶ *b*: Stateless Synchronous Integration.
- ▶ *c*: Stateless Asynchronous Integration.
- ▶ *d*: Stateful Asynchronous Integration (not considered further in this book).

It is possible to further decompose these interactions into finer levels of detail, but these classifications will be adequate for modeling the technical challenges and technical solutions addressed in this book.

This give us six potential interaction cases to consider:

- ▶ *Case f1.a*: Stateful Synchronous Integration from WebSphere to .NET.
- ▶ *Case f1.b*: Stateless Synchronous Integration from WebSphere to .NET.
- ▶ *Case f1.c*: Stateless Asynchronous Integration from WebSphere to .NET.
- ▶ *Case f2.a*: Stateful Synchronous Integration from .NET to WebSphere.
- ▶ *Case f2.b*: Stateless Synchronous Integration from .NET to WebSphere.
- ▶ *Case f2.c*: Stateless Asynchronous Integration from .NET to WebSphere.

These cases are illustrated in Figure 4-54 and Figure 4-55 on page 182.

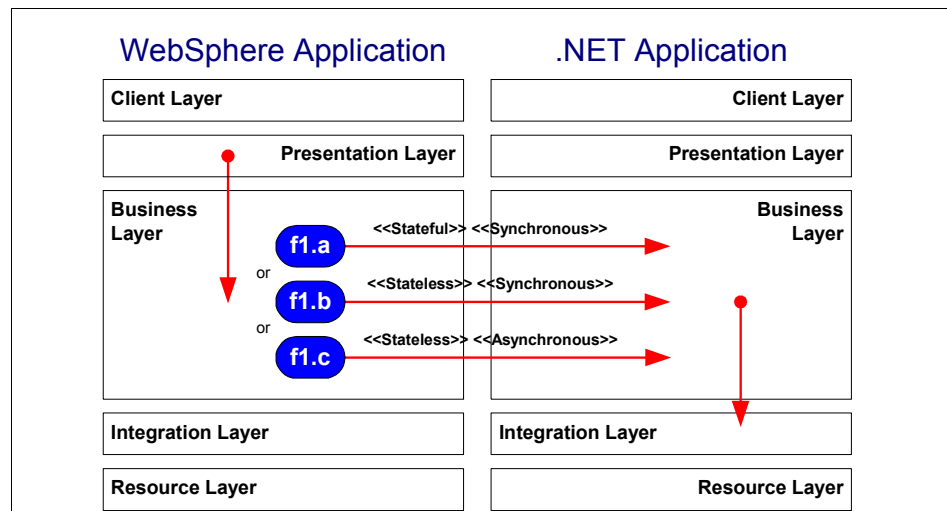


Figure 4-54 interaction cases f1.a, f1.b and f1.c (WebSphere application's perspective)

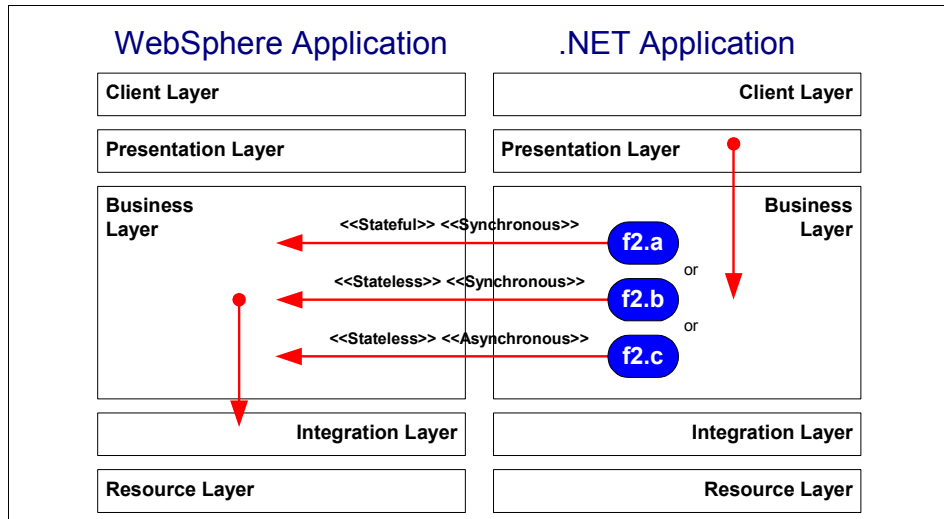


Figure 4-55 Interaction cases f2.a, f2.b and f2.c (.NET application's perspective)

4.4, “Technical solution mapping” on page 202 provides guidance on identifying potential technical solutions for each of these interaction cases (f1.a, f1.b, f1.c, f2.a, f2.b and f2.c).

4.3.7 Interaction case g: business logic to resource

Let's consider the coexistence of an application deployed in WebSphere and an application deployed in .NET via Business layer logic to resource integration. We can consider interaction between these two applications from these perspectives:

- *Perspective g1 isolated: WebSphere to .NET*
A runtime artifact executing within the Business layer of a WebSphere application (possibly an EJB or underpinning Java Bean) deployed within a the WebSphere EJB Container, interacting with a runtime resource of a .NET application.
- *Perspective g2 isolated: .NET to WebSphere*
A runtime artifact executing within the Business layer of a .NET application (possibly an assembly or a COM+ component) deployed within the .NET Framework, interacting with a runtime resource of a WebSphere application.
- *Perspective g1/g2 concurrent isolated: both WebSphere and .NET*
Runtime artifacts executing within the Business layer of a WebSphere application and runtime artifacts executing within the Business layer of a

.NET application both the access the same shared resource concurrently to perform separate and distinct units of work.

- *Perspective g1/g2 concurrent conjoined:* both WebSphere and .NET Runtime artifacts executing within the Business layer of a WebSphere application and runtime artifacts executing within the Business layer of a .NET application both the access the same shared resource concurrently (or in close synchronization) to perform different aspects of the same logical unit of work.

The high-level objective models for these perspectives are illustrated in Figure 4-56, Figure 4-57 on page 184, Figure 4-58 on page 184 and Figure 4-59 on page 185.

Resources can simplistically divide resources, and the integration layer technologies used expose these resource to Business layer components, into the following categories:

- Relational Database: JDBC, ADO .NET, ODBC, API
- Enterprise Applications: JCA, MQ, MSMQ, API, RPC
- SOA Services: Web Services
- Operating System Services: API

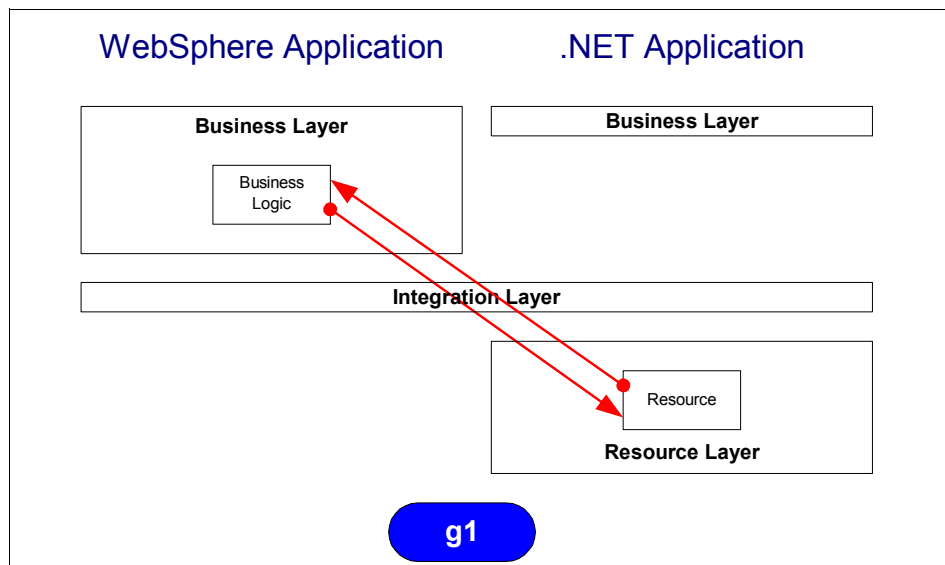


Figure 4-56 A high-level objective model: case g1 isolated

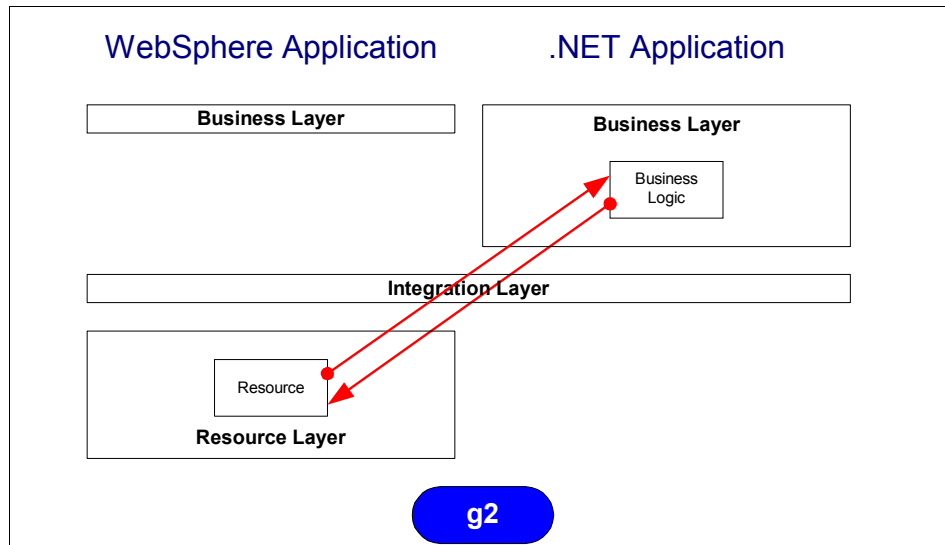


Figure 4-57 A high-level objective model: case g2 isolated

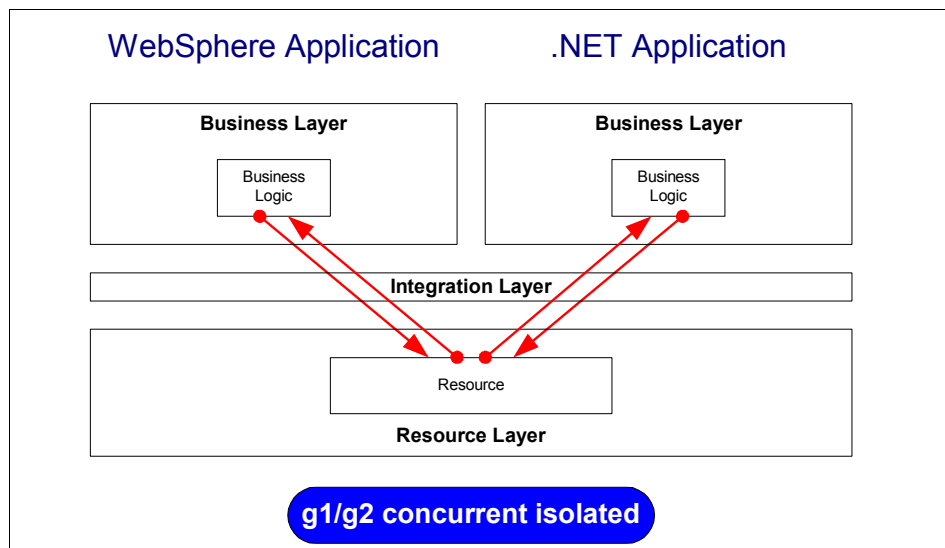


Figure 4-58 A high-level objective model: case g1 and g2 concurrent isolation

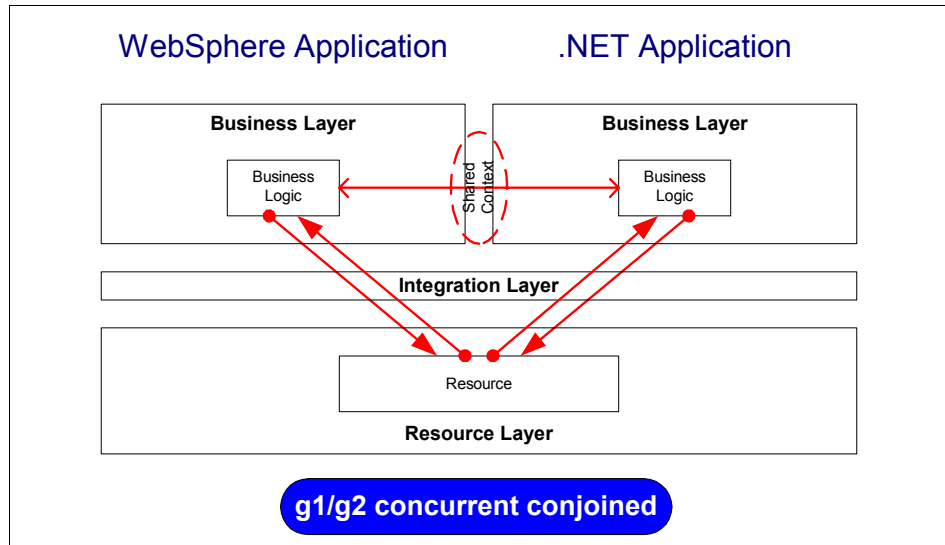


Figure 4-59 A high-level objective model: case *g1* and *g2* concurrent conjoined

Interaction case g: structure and dynamic considerations

Consider the scenario illustrated in Figure 4-60 on page 186. Both a WebSphere application and .NET application share the same relational database resource. When we consider each interaction in isolation (such as interaction *case g1 isolated* and interaction *case g2 isolated*), this is business as usual for each application. The WebSphere application can use JDBC to operate on the database data set in isolation. The .NET application can use ADO.NET to operate on the database data set in isolation. In this scenario, isolation is provided by the RDBMS in cooperation with the JDBC or ADO.NET driver.

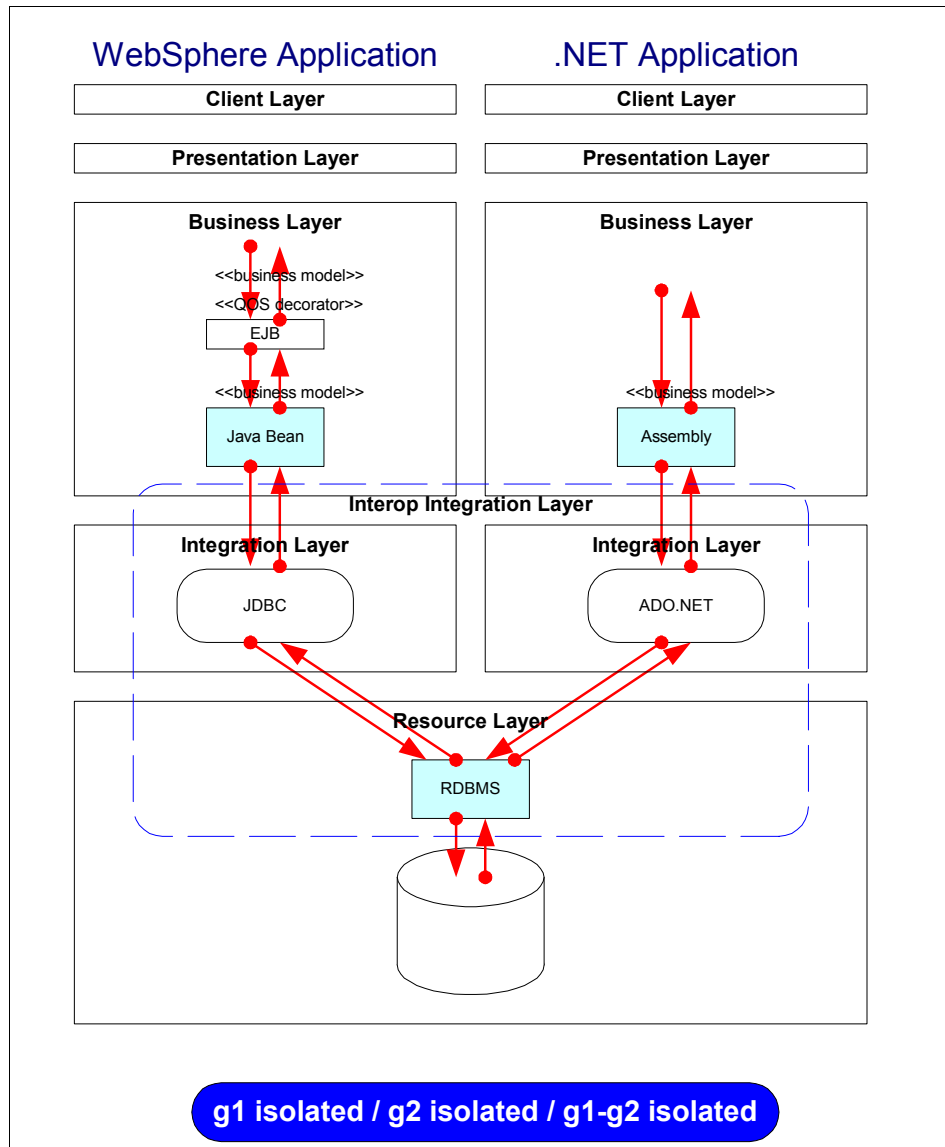


Figure 4-60 Isolated interaction case g1 and g2: a shared database resource

The challenge for this model comes when pushing the coexistence scenario up into the Business layer to give interaction *case g1-g2 conjoined*. Consider the scenario where a single database resource is shared by coexistent WebSphere and .NET Business layer implementations. Say both the WebSphere Business layer component and the .NET Business layer component need to combine to fulfill a single unit of work. This is illustrated in Figure 4-61 on page 187 from the

perspective of a WebSphere application (analogous to case f1 extended), but it is easy to imagine a similar model from a .NET applications perspective (analogous to case f2 extended). The challenge for this scenario becomes the following: how do we share runtime context between the Business layers and how can we ensure that work performed by each Business layer is treated by the RDBMs as a single unit of work?

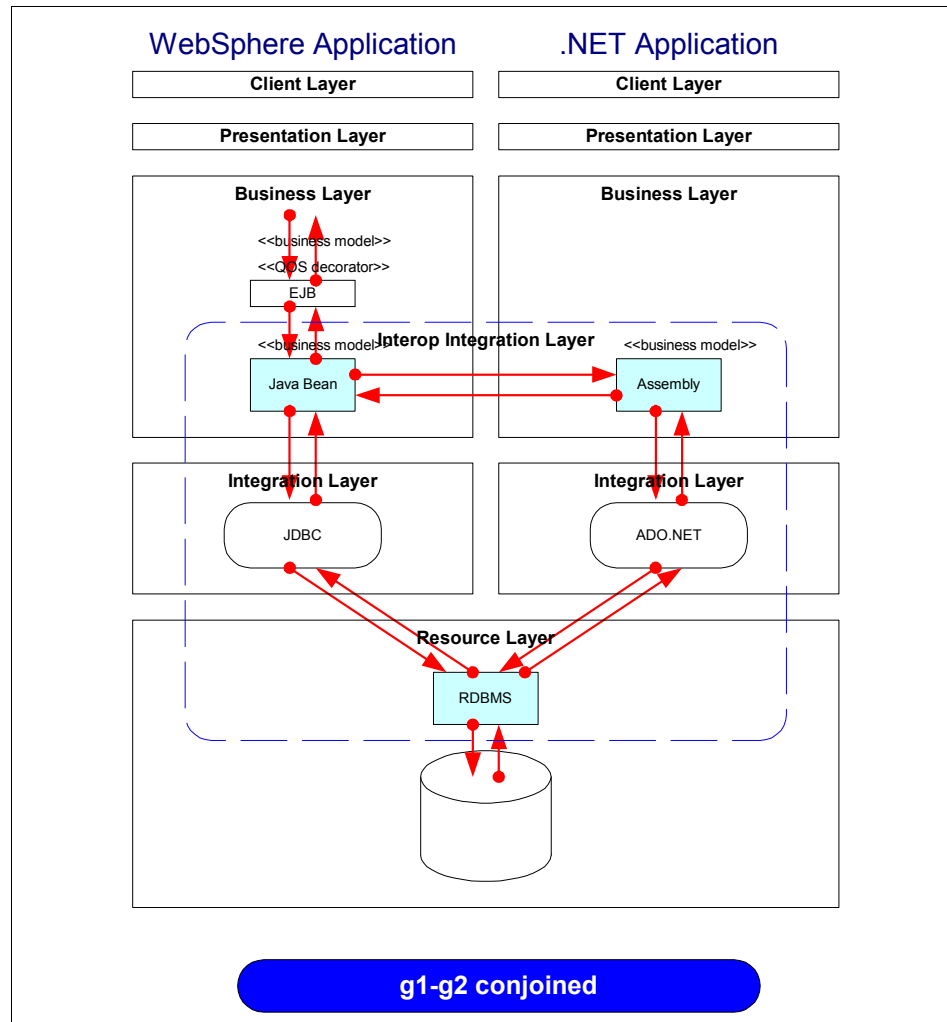


Figure 4-61 Conjoined interaction case g1 and g2: a shared database resource as an extension of case f1

Consider the scenario illustrated Figure 4-62 on page 188. Both a WebSphere application and .NET application share the same application resource. The

WebSphere application is illustrated integrating with the resource application via a JCA connector, or via a bespoke client API. The .NET Framework has no technical equivalent to JCA, so we have illustrated the .NET application integrating with the resource application via a bespoke API. When we consider each interaction in isolation (such as interaction *case g1 isolated* and interaction *case g2 isolated*) this should be business as usual for the API or JCA implementation.

Once again, the more challenging interaction case is *case g1-g2 conjoined*. This interaction case is illustrated in Figure 4-63 on page 189 (analogous to case f1 extended).

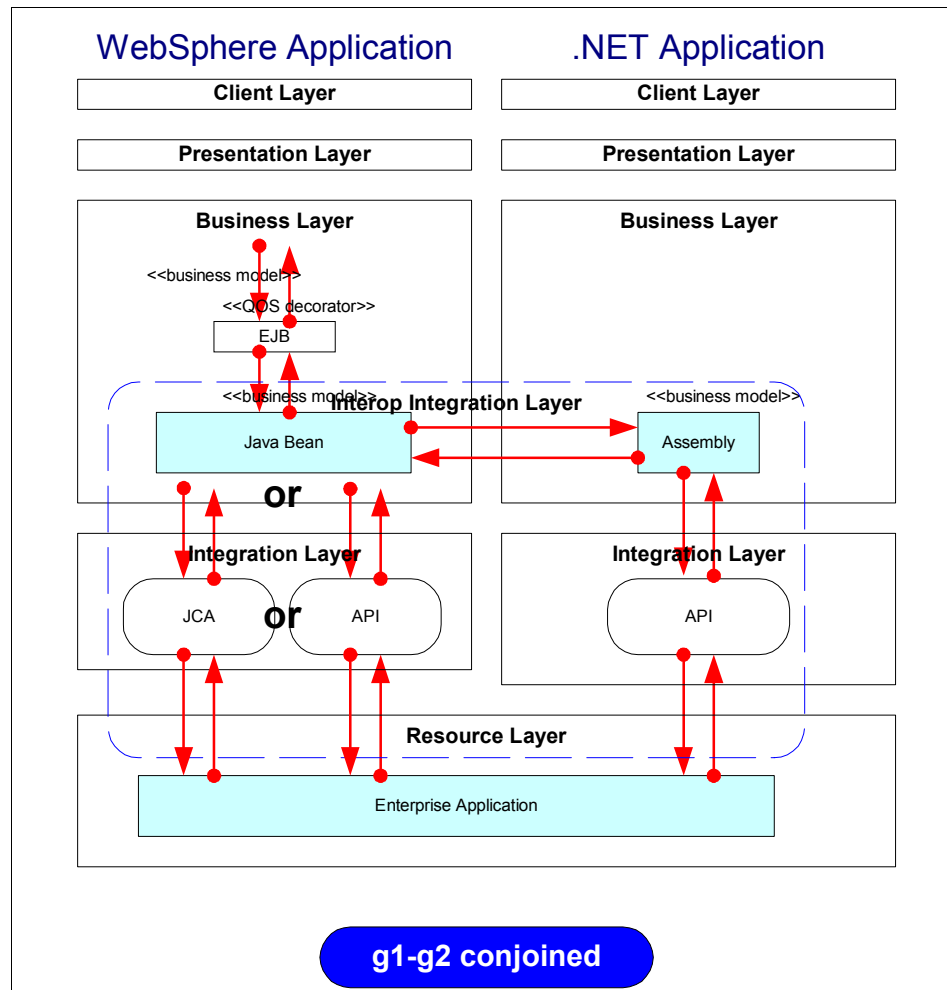


Figure 4-62 Isolated interaction case g1 and g2: a shared application resource

An interesting candidate solution model for interaction case g1-g2 conjoined is to make a decision to perform all interaction with a given resource through a single integration layer. This is illustrated in Figure 4-64 on page 190 from both a WebSphere applications perspective (case g1/f1 extended) and a .NET applications perspective (case g2 / f2 extended). This removes the challenge of both WebSphere application's and .NET application's integration layers needing to cooperate to deliver a single unit of work, by simply removing one or another of the integration layers from the solution. The challenge of sharing context between the WebSphere application and the .NET application at the Business layer remains.

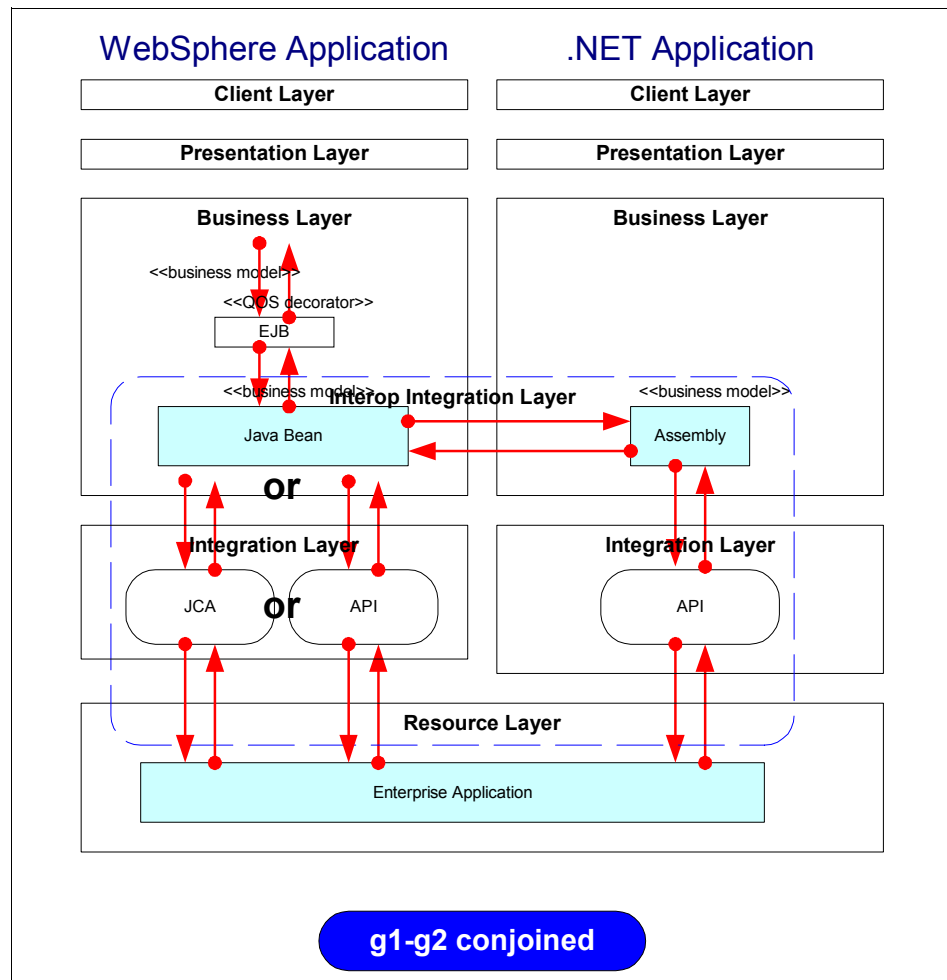


Figure 4-63 Conjoined interaction case g1 and g2: a shared application resource as an extension of case f1

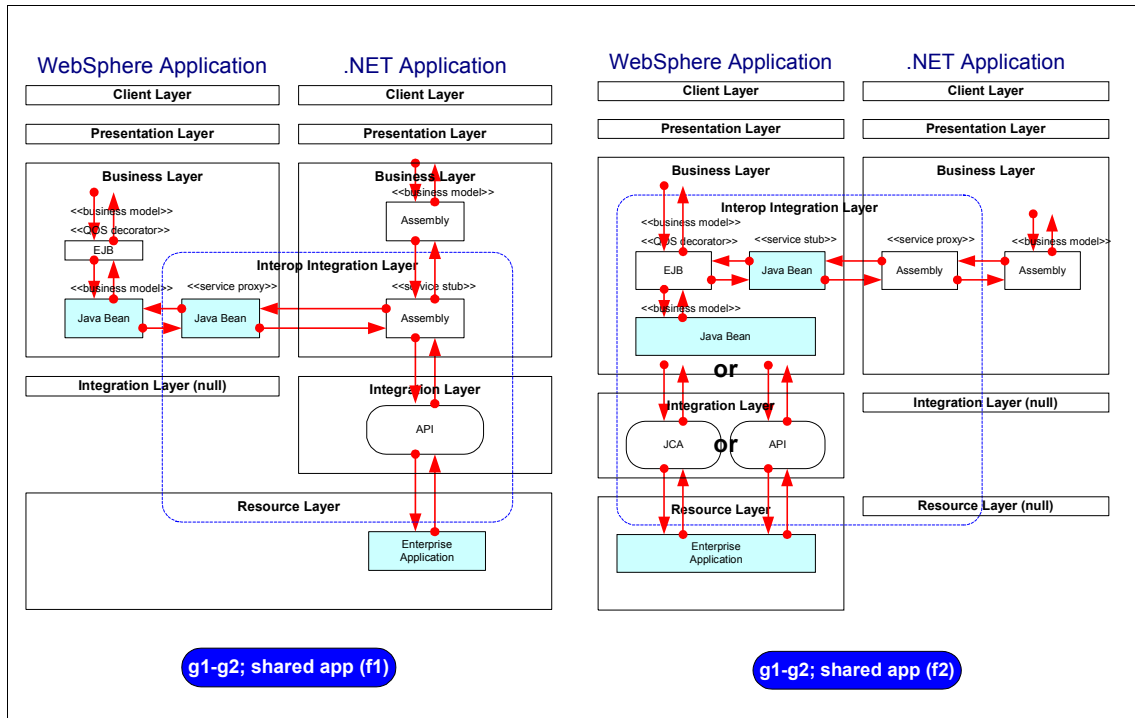


Figure 4-64 Interaction case g1 and g2: candidate solution model(s) for a shared application

Consider the scenario illustrated in Figure 4-65 on page 191, a WebSphere application and a .NET application both sharing the same MQ enabled resource. MQ is a proven *strategic* solution for the common challenge of integrating enterprise applications. The message-oriented solution model delivered by WebSphere MQ automatically addresses the needs of interaction case g1 isolated, case g2 isolated and case g1-g2 concurrent isolated. Once again, the most challenging interaction case will be interaction *case g1-g2 conjoined*. The difference is that with WebSphere MQ messaging, the context always flows as part of the message.

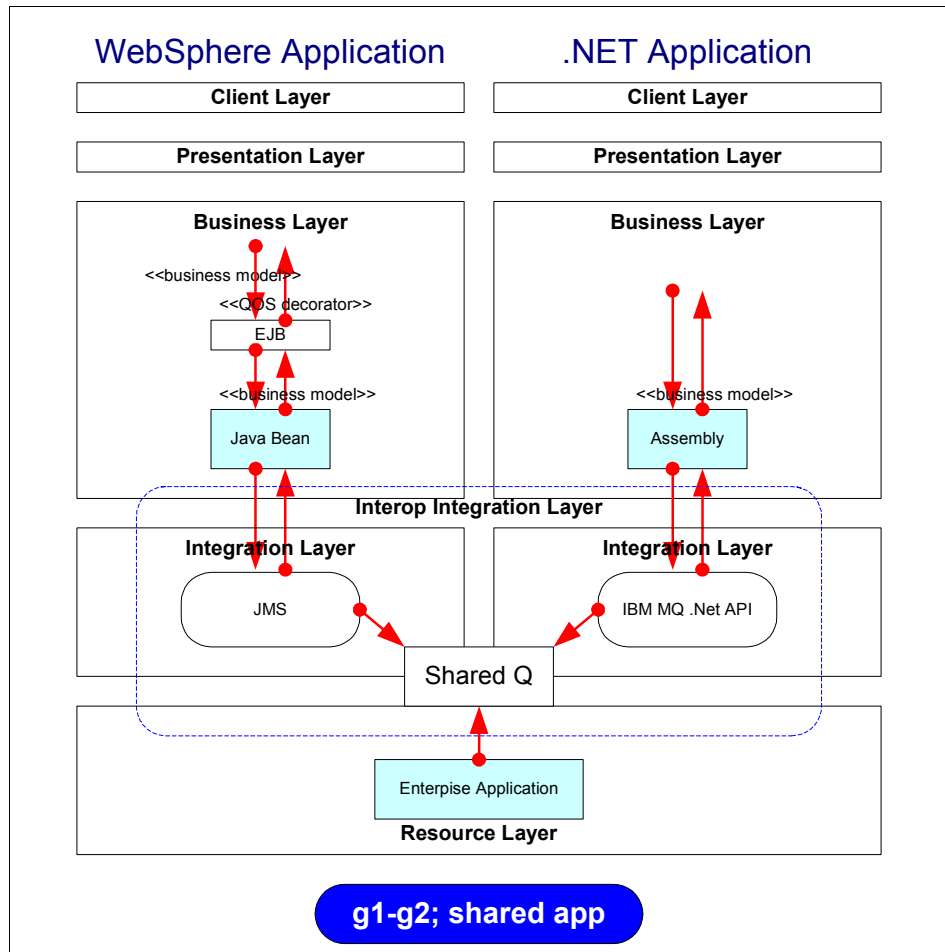


Figure 4-65 Isolated interaction case g1 and g2: a shared queue to an IBM WebSphere MQ enabled application

Consider the scenario illustrated in Figure 4-66 on page 192. It shows a WebSphere application and .NET application sharing the same Web Services enabled resource. Web Services are rapidly emerging as a prime *strategic* solution for providing an implementation technology neutral integration solution. The industry backing for these technologies, and the standardization effort being exerted by IBM, Microsoft, and others, suggest that Web Services is going to be the dominant strategic solution for integration and interoperability. This is especially the case for solutions requiring integration and interoperability between WebSphere applications and .NET applications.

Once again, the challenges facing the *g1-g2 conjoined* interaction case (Figure 4-67 on page 193) present the biggest challenge for coexistence. However, the WS-Security and WS-Transactions Web Services specifications are specifically addressing the issue of context sharing between implementation technologies. As these specifications and implementations of these specifications mature, Web Services will become the simplest way to solve conjoined coexistence scenarios across any of our coexistent WebSphere and .NET application layers.

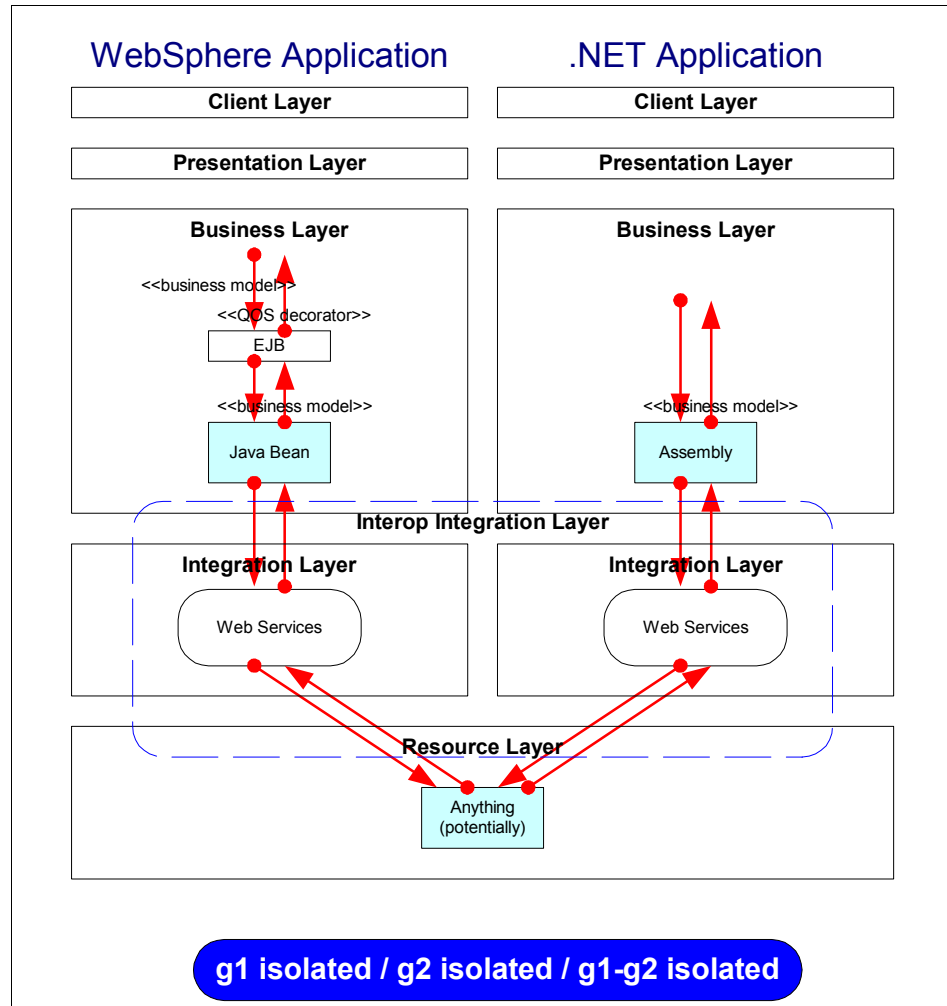


Figure 4-66 Isolated interaction case g1 and g2: a shared Web Service enabled resource

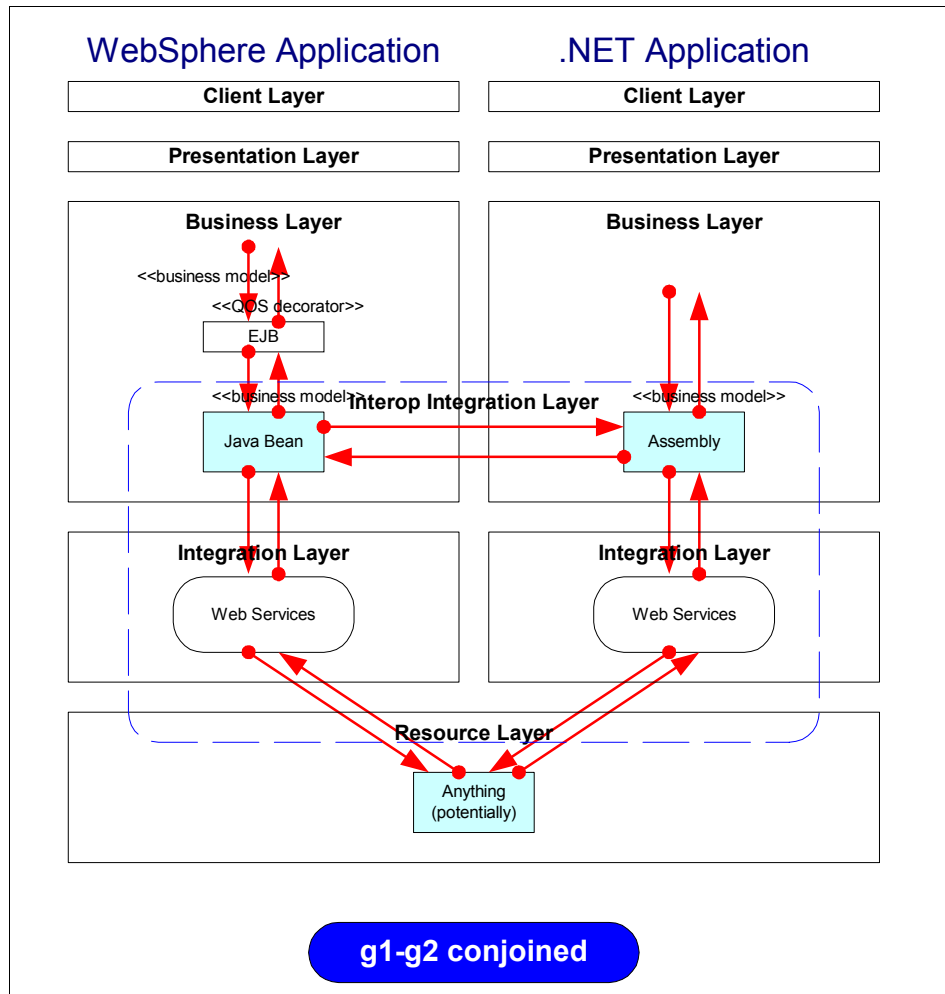


Figure 4-67 Conjoined interaction case g1 - g2: a shared Web Service enabled resource

Interaction case g: component interaction classifications

For each of these interaction perspectives (case g1 and case g2), the interaction between components can be considered to have one of the following integration classifications:

- ▶ *a*: Stateful Synchronous Integration.
- ▶ *b*: Stateless Synchronous Integration.
- ▶ *c*: Stateless Asynchronous Integration.
- ▶ *d*: Stateful Asynchronous Integration (not considered further in this book).

It is possible to further decompose these interactions into finer levels of detail, but these classifications will be adequate for modeling the technical challenges and technical solutions addressed in this book.

This gives us six potential interaction cases to consider:

- ▶ *Case g1.a:* Stateful Synchronous Integration from WebSphere to .NET.
- ▶ *Case g1.b:* Stateless Synchronous Integration from WebSphere to .NET.
- ▶ *Case g1.c:* Stateless Asynchronous Integration from WebSphere to .NET.
- ▶ *Case g2.a:* Stateful Synchronous Integration from .NET to WebSphere.
- ▶ *Case g2.b:* Stateless Synchronous Integration from .NET to WebSphere.
- ▶ *Case g2.c:* Stateless Asynchronous Integration from .NET to WebSphere.

These cases are illustrated in Figure 4-68 and Figure 4-69 on page 195.

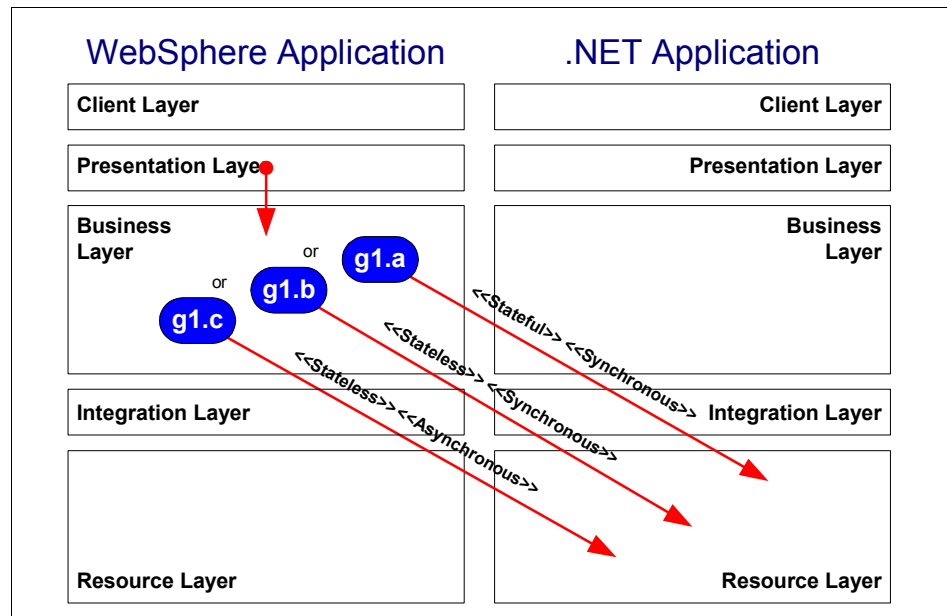


Figure 4-68 interaction cases g1.a, g1.b and g1.c (WebSphere application's perspective)

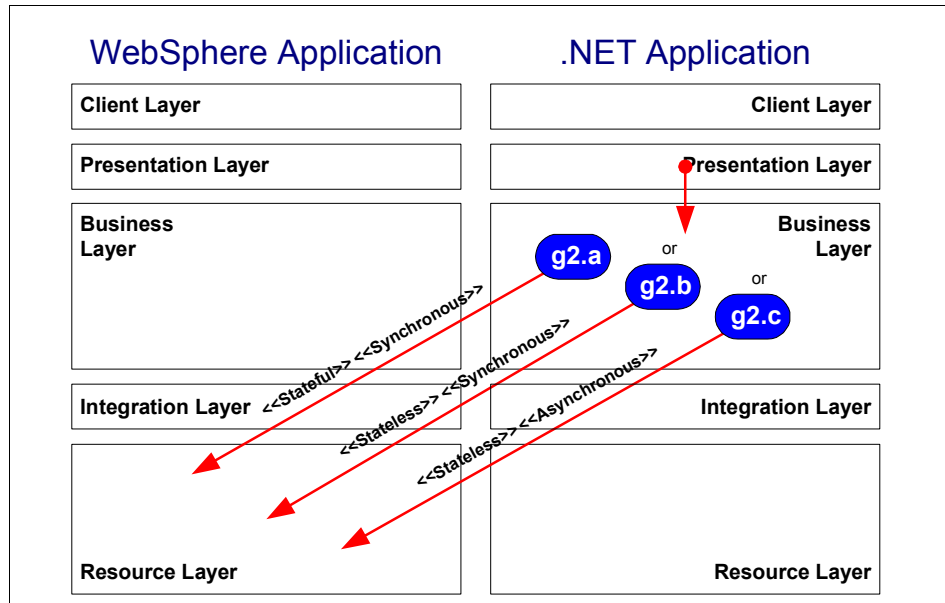


Figure 4-69 interaction cases g2.a, g2.b and g2.c (.NET application's perspective)

4.4, “Technical solution mapping” on page 202 provides guidance on identifying potential technical solutions for each of these interaction cases (g1.a, g1.b, g1.c, g2.a, g2.b and g2.c).

4.3.8 Interaction case h: resource to resource

The term *resource* is very vague. In order to effectively analyze resource to resource interactions, we first must define what we consider to be a resource. The general definition of a resource is “something that may be populated and consumed.” Resources generally reside a level above the system software and provide some shared service. This service may provide storage and retrieval of data, a messaging system, or execute some action. The key things to remember when defining a resource are purpose and generality. Simply, resources should not be specific to any single business process and should provide some general service or set of services. A line should be drawn between the business process and the resources the process consumes with some sort of integration layer to facilitate interaction between them.

Some examples of the most common resources are Relation Database Management Systems, Message Queueing Systems, and directory services. All these resources are very general and may be shared by multiple applications. When we use the term *resources*, these are the types of resources we are referring to.

Resource-level interaction must be triggered by some event or sequence of events that occurs in one of the layers above it. In some cases, it may even be a timed event. An example of resource interaction is a trigger in a relational database system. A trigger executes some code whenever a specific database operation occurs. The code that is executed may require some external resource or push some data to an external resource.

Now that we've discussed resources, let's consider the coexistence of an application deployed in WebSphere and an application deployed in the dotNet environment with Resource layer logic integration. We can consider interaction between these two resources from two possible perspectives:

- ▶ *Perspective h1*: A runtime artifact executing within the Resource layer of an application deployed within an instance of WebSphere, initiating a communication to an application artifact executing within the Resource layer of an application deployed within an instance of the .NET Framework.
- ▶ *Perspective h2*: A runtime artifact executing within the Resource layer of an application deployed within an instance of the .NET Framework, initiating a communication to an application artifact executing within the Resource layer of an application deployed within an instance of WebSphere.

These two perspectives are illustrated in Figure 4-70.

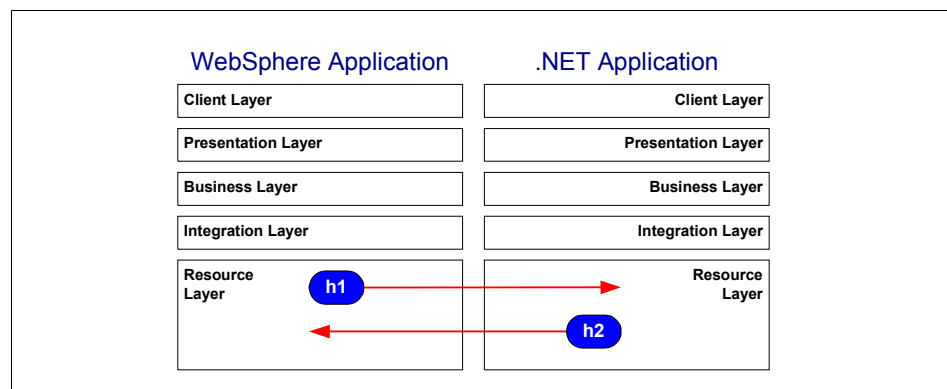


Figure 4-70 Resource to resource interaction

For each of these perspectives (h1 and h2), the interaction style can be considered have one of the following classifications:

- ▶ *a*: Stateful Synchronous Integration.
- ▶ *b*: Stateless Synchronous Integration.
- ▶ *c*: Stateless Asynchronous Integration.

It is possible to further decompose these interactions into finer detail, but these classifications will be adequate for modeling the technical challenges and technical solutions addressed in this book.

This give us six potential interaction cases to consider:

- ▶ *Case h1.a:* Stateful Synchronous Integration from WebSphere to .NET.
- ▶ *Case h1.b:* Stateless Synchronous Integration from WebSphere to .NET.
- ▶ *Case h1.c:* Stateless Asynchronous Integration from WebSphere to .NET.
- ▶ *Case h2.a:* Stateful Synchronous Integration from .NET to WebSphere.
- ▶ *Case h2.b:* Stateless Synchronous Integration from .NET to WebSphere.
- ▶ *Case h2.c:* Stateless Asynchronous Integration from .NET to WebSphere.

These cases are illustrated in Figure 4-71 on page 198 and Figure 4-72 on page 199.

Let's consider each of these interaction cases in turn, first from the perspective of calls from WebSphere applications, then from the perspective of .NET applications.

WebSphere application to .NET application

Let's start with an example of resource to resource interaction, specifically from a WebSphere to a .NET application. A scenario where this type of interaction may occur is centered around the typical 'Allow us to send you more information regarding this product or service' item seen on many Web forms when registering a product or for a service. In this example, the registration application is hosted by WebSphere Application Server. The Resource layer of this application is the IBM DB2 relational database system. The database system contains database tables containing registration information. Each time a new user is registered, their personal information is stored in the database. The database system also contains a trigger. A trigger is analogous to an event listener with a specific set of criteria. When these criteria are met, the trigger is fired. When the trigger is fired, a set of instructions is executed.

The criteria for firing the above trigger is a user choosing to receive more information about the product or service they are registering. An outside marketing company has been contracted to perform this service. The application they use to send addition information uses .NET and their contact information is stored using Microsoft SQL Server. The DB2 trigger performs a resource to resource interaction, forwarding the user information to the SQL Server database. The SQL Server database also has a trigger. This trigger informs the application that a new customer is requesting its services. This request is

propagated up the logic chain and at the endpoint, the information is sent to the user.

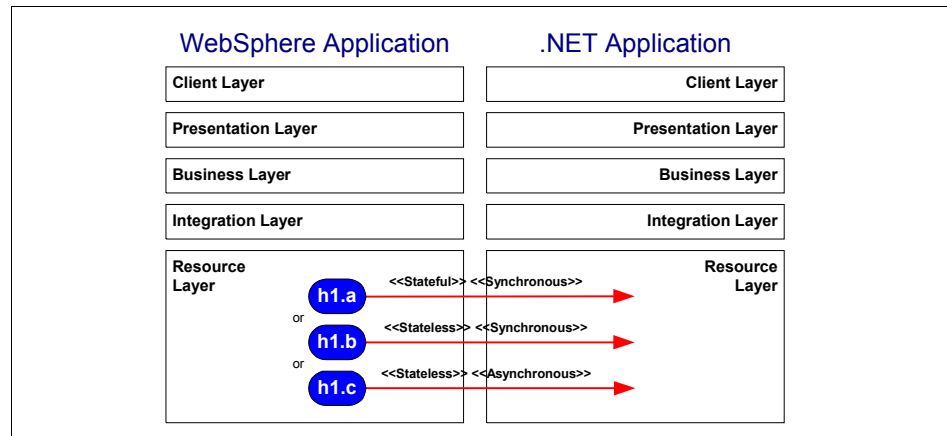


Figure 4-71 WebSphere application to .NET application

Of interest in this WebSphere and .NET coexistence scenario is not the process itself, but the technologies and interactions used to perform resource to resource interactions of this nature. The interaction cases below discuss these topics in more detail.

Interaction case h1.a

This may be the least common scenario for resource to resource interaction. However, this interaction case is possible. While each resource maintains its own state, state of the interaction normally is not required. Cases where state is important for resource to resource interaction might be database to database replication, synchronization, or back-up. In these cases, each resource needs to know what state it is in during the process.

Another situation where state may be important is for the registration scenario above. The WebSphere application database may wish to store the user selection for receiving more information. The application may allow the user to change this value at a later date. By maintaining state, the trigger would know not to send information to a user who already received information.

Interaction case h1.b

This interaction case, stateless synchronous, is the most likely for resource to resource interaction. Typically, the resource requesting the interaction sends a single request or multiple stateless requests to the secondary resource. This case matches the interaction of our registration system. The DB2 database trigger fires a single, stateless request to the SQL Server database.

Interaction case h1.c

Asynchronous resource to resource interaction between a .NET and a WebSphere application normally involves the use of messaging services. In a coexistence scenario, each platform may be using the same message queue resource provider or they may be using different providers. In either case, messaging middleware is provided by each vendor to access messaging resources. In a resource to resource interaction, the messaging provider, such as IBM WebSphere MQ, may receive a message prompting some local processing. This local processing may call a Java application, which in turn interacts with the messaging resource on a .NET platform. This may be Microsoft MQ or WebSphere MQ.

.NET application to WebSphere application

Resource to resource interaction from a Microsoft .NET application to a WebSphere application may occur very similarly to the scenarios discussed above. The same scenarios are possible, only the start and end points are different. Each application has its own Resource layer which can be a relational database system, messaging system, or some other resource. Using a commercial resource provider such as IBM WebSphere MQ or IBM DB2 UDB is one of the best ways to ensure there is adequate middleware (by the same vendor or a third party vendor) to ensure cross-platform interoperability.

In addition to using commercial products for resource management, applications may use and provide custom resource management. A Microsoft Windows service or even a IBM WebSphere Web Service could be considered a resource. The interaction cases below discuss some of the possibilities for custom resource to resource interactions.

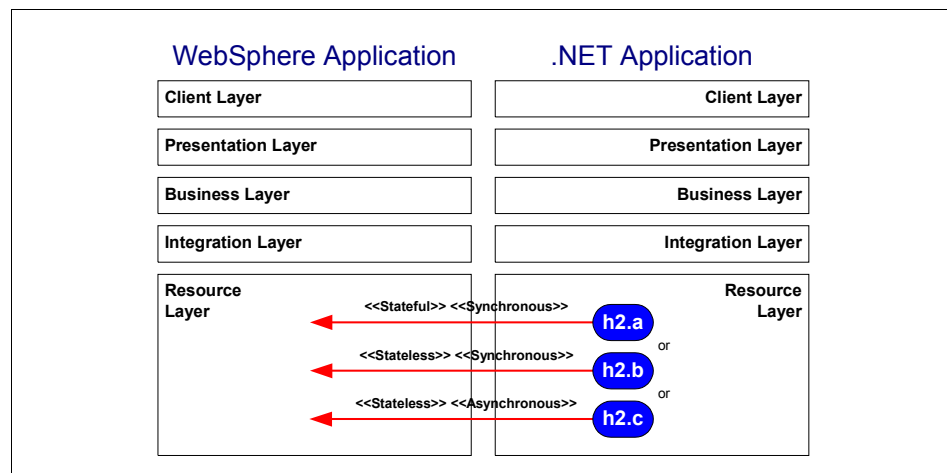


Figure 4-72 .NET application to WebSphere application

Interaction case h2.a

Similar to interaction case h1.a above, this interaction combination is unlikely in a resource to resource scenario. Interaction case h2.b is more likely. However, this interaction is possible and may be required in some situations. An example of this type of interaction might be a Windows service that issues software and feature licenses to its clients. The service needs to know the state of the license and clients must renew licenses and inform the server of the features it is using. A WebSphere application, executed from some resource, may need to access this service. Hence, stateless synchronous resource to resource interaction results. Other scenarios are also possible for stateful synchronous resource to resource interaction.

Interaction case h2.b

Stateless, synchronous interactions are the most likely when performing resource to resource interactions from a .NET application to a WebSphere application. An example of this type of interaction might be a WebSphere Web Service that gets called by the Resource layer of the .NET application. For example, let's say the .NET application accesses a custom resource that provides driving route information for a given location. A WebSphere application collects and compiles traffic information. This information is also available via a Web Service provided by the Resource layer. The Resource layer for the .NET application could also access this resource and return it along with route information. This is just one example of resource to resource, stateless synchronous interaction.

Interaction case h2.c

Similar to interaction case h1.c, asynchronous stateless interaction in a resource to resource coexistence scenario should normally be done using a messaging provider. There are currently few to no other good solutions available for this type of interaction. One future possibility is one-way calls over IIOP from .NET to Java, but there is currently no preferred IIOP provider for .NET.

Recommendations

The following are some recommendations and considerations for resource to resource interaction between applications deployed across IBM WebSphere and Microsoft .NET.

- ▶ Intranet versus Internet boundaries
 - If resources are within an intranet, it is often easiest and most beneficial to share them, if possible. For example, sharing a relational database management system will allow simpler resource to resource interactions than having two separate, dissimilar database systems.

- Resource to resource connectivity can be a problem over a firewalled Internet connection. Web Services and other specialized middleware can overcome this problem. However, it should be kept in mind that many resource providers normally do not provide a canned solution for this problem.
- ▶ Resource interfaces
 - What types of interfaces are available to each resource? Do they allow direct resource to resource interactions? This is a common problem between resource providers produced by different vendors. Often, some sort of middleware must be used.
 - Are the interfaces available for .NET and Java? Java is a more mature technology so native interfaces are more likely. With .NET, some sort of bridging technology may be required until a native version is provided.
- ▶ Interactions
 - Typically, stateless synchronous interactions are the most common and are suited to most situations. Maintaining state adds an extra burden to both resource systems and in some cases, may not be an option for one or both resources.
 - If asynchronous interactions are required messaging software such as IBM WebSphere MQ or Microsoft MQ should be used. There are few other solutions available.

4.3.9 Conclusion and recommendations

In this chapter, we have identified many possible interaction cases. Some of these interaction cases will be encountered frequently (perhaps case e, case f and case g), some of these will be encountered infrequently (perhaps case a, case c and case h), some are interesting but improbable (perhaps case d) and some *may* not require much consideration (perhaps case b).

The recurring theme across most of the candidate solution models presented in this section is the proxy-stub pattern. The proxy-stub pattern *can* be used to provide the structure for separation of concerns between business interfaces, client implementations, service implementations, adaption, and integration. The proxy-stub is especially valuable in our scenarios of coexistence between WebSphere applications and .NET applications. It is also especially valuable when we need to consider deploying one *tactical* solution in the short term, and a different *strategic* solution in the long term.

Whether or not you choose to use one of these candidate solution models, it is our recommendation that you design an interoperation integration layer (using the proxy-stub pattern, or some alternative pattern) to hide the integration

implementation from both the client implementation and the service implementation.

One of the more difficult interaction cases to solve will probably always be interaction case *g1-g2 concurrent conjoined*. As Web Services support for context sharing matures, it is likely to become a standard solution for this interaction case. In the meantime, prototyping and testing are going to remain important techniques for proving, or disproving, alternative solution candidates for this interaction case.

4.4 Technical solution mapping

This section is not intended to be prescriptive, but it is intended to indicate the technical solutions that should be considered for the previously identified cross-layer integration cases. The correct choice of technical solution for a given coexistence challenge may be influenced by many factors, such as, but not limited to:

- ▶ Targeted deployment environment: what middleware is already there?
- ▶ Strategic direction: is a given technical solution more appropriate for the chosen strategy of the customer? Will you need to provide different solutions to meet different strategic directions of different customers?
- ▶ Available time: do you need a quick solution, and are the developers already familiar with one of these solutions?
- ▶ Available budget: what are the build, manage and maintenance costs of one solution compared to another?
- ▶ Qualities of service: is one solution better suited than another to your performance, security, scalability (etc.) needs?
- ▶ Manifold solutions: real coexistence problems may be compounds of more than one of the interoperability cases identified in this chapter. If this is the case, then it may make sense to deploy several (optimal) solutions.
- ▶ Solution rationalization: it may make sense to try and rationalize several potential (optimal) solutions into a common (acceptable) solution.
- ▶ Technical solution maturity: your ideal (strategic) technical solution may still be maturing. It may be deficient in a required characteristic, but on the cusp of delivering all your requirements. In this scenario, if the interoperability integration layer is well designed, it will normally be possible to totally abstract the integration technology from both the client implementation code and the service implementation code. This implies that, if it is appropriate, you should be able to choose a tactical short term solution, then, with a minimum of disruption, substitute it for the ideal (strategic) solution at a later date.

Let's consider some candidate technical integration solutions, and map them to the interaction characteristics and interaction case we identified in section 4.2, "Fundamental interaction classifications" on page 111 and section 4.3, "Layer interaction classifications" on page 132.

4.4.1 Stateful synchronous integration solution candidates

This section discusses the stateful synchronous integration solution candidates from two perspectives:

- ▶ "WebSphere applications perspective" on page 203
- ▶ ".NET applications perspective" on page 209

WebSphere applications perspective

Let's consider stateful synchronous integration from the perspective of a WebSphere application artifact invoking a .NET application artifact. This is illustrated in Figure 4-73.

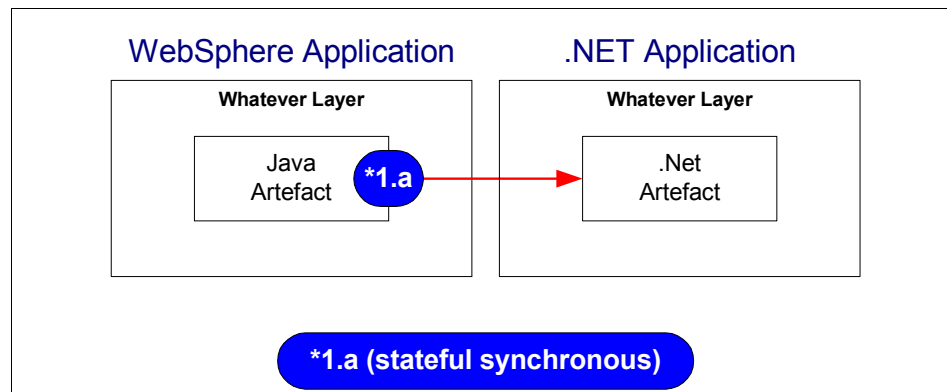


Figure 4-73 stateless asynchronous integration cases *1.a

Solution candidate: IBM Interface Tool for Java

The solutions we suggest for consideration will be relevant for the following interaction cases.

- ▶ **Case a1.a:** Stateful Synchronous Integration between Client layer and Client layer, from WebSphere to .NET, a recurrent solution candidate for this interaction case.
- ▶ **Case c1.a:** Stateful Synchronous Integration between Client layer and Business layer, from WebSphere to .NET, a recurrent solution candidate for this interaction case.

- **Case e1.a:** Stateful Synchronous Integration between Presentation layer and Business layer from WebSphere to .NET, a recurrent solution candidate for this interaction case.
- **Case f1.a:** Stateful Synchronous Integration between Business layer and Business layer from WebSphere to .NET, a recurrent solution candidate for this interaction case.
- **Case g1.a:** Stateful Synchronous Integration between Business layer and Resource layer from WebSphere to .NET, an occasional solution candidate for this interaction case if the resource is exposed locally as a .NET or COM implementation.

Figure 4-74 illustrates an overview of IBM Interface Tool for Java. IBM Interface Tool for Java leverages .NET COM interop features to provide stateful Java proxies to .NET assemblies.

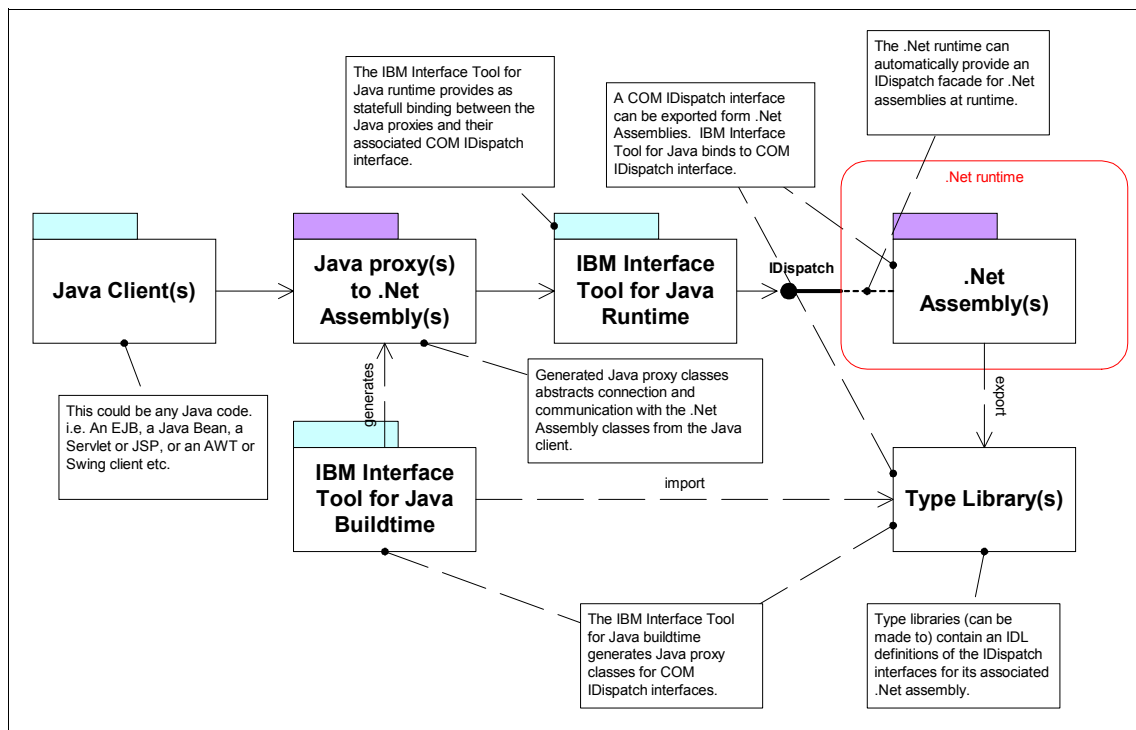


Figure 4-74 IBM Interface Tool for Java overview

In 4.2, "Fundamental interaction classifications" on page 111, we identified some characteristics for consideration when classifying your given interaction cases. Table 4-1 on page 205 identifies how/if the IBM Interface Tool for Java delivers these characteristics.

Note: *IBM Interface Tool for Java* is the technology formerly know as Bridge2Java. See:

<http://www.alphaworks.ibm.com/tech/bridge2java>

Table 4-1 *IBM Interface Tool For Java interaction characteristic matching*

Characteristic	Supported	Comments
By Value argument paradigm	yes	-
By reference argument paradigm	yes	-
Document interface style	yes	-
RPC interface style	yes	-
Distributed object paradigm	potentially	Yes, if used in combination with an inter-process communication mechanism. See Figure 4-76 and Figure 4-77.
Service oriented paradigm	no	-
Message-oriented paradigm	no	-
Call context propagation	no	-
Proxy Head Protocol	API	The proxies built by the IBM Interface Tool for Java build-time use this API.
Stub Tail Protocol	encapsulated	Not exposed. Uses .NET's in-built COM interop services.
Service discovery mechanism	none	Location is configured via the Windows Registry and the .NET GAC.
Transport mechanism	none	IBM Interface Tool for Java is an in process solution, but can be combined with inter-process communication mechanisms. See Figure 4-76 and Figure 4-77.

Characteristic	Supported	Comments
Communication protocol	encapsulated	-
Service binding/hosting	-	Binds via the COM runtime. Hosted using a .NET CLR shim defined in the registry.
Open standards	no	-
Longevity	Tactical	As Web Services matures, it will probably become the strategic choice for this interaction characteristic.

The IBM Interface Tool for Java provides in-process binding to .NET assemblies. As Figure 4-75 illustrates, the simplest solution candidate loads the Interface Tool for Java runtime, and the assembly (and its CLR instance) into the the Java clients JVM process.

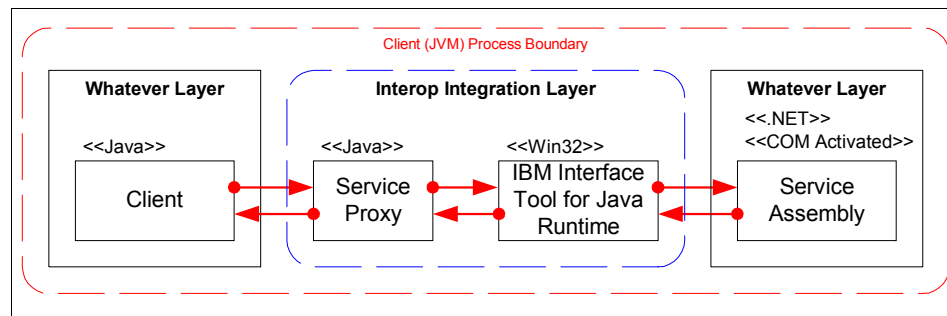


Figure 4-75 IBM Interface Tool for Java: in-process solution candidate

If the .NET assembly you need to bind to is not local (not on the same machine), or you want to keep the .NET assembly out of the JVM process, then you will need a some sort of inter-process communication mechanism between the Interface Tool for Java runtime, and the .NET assembly (the Interface Tool for Java runtime is always loaded into the client's JVM process).

Figure 4-76 on page 207 illustrates a solution candidate that uses .NET remoting as the inter-process communication mechanism. This model extends Figure 4-75 by adding a .NET remoting proxy and stub to the interop integration layer. However, with this solution, you still need a .NET assembly in the client process to host the .NET remoting proxy. This means that a CLR instance will still be loaded in the client's JVM process. So this solution candidate delivers inter-process communication, but it still puts a .NET CLR instance in the JVM process, and it also adds extra build complexity.

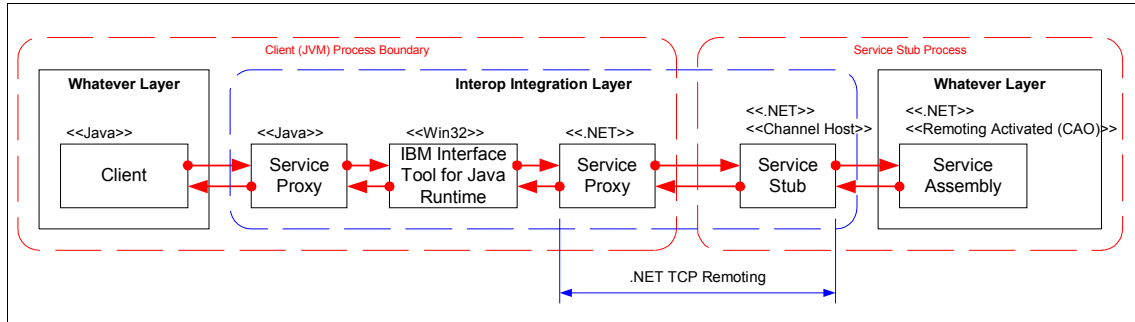


Figure 4-76 IBM Interface Tool for Java: inter-process solution candidate: .NET remoting

Figure 4-77 illustrates a solution candidate that uses DCOM as the inter-process communication mechanism. This solution model is simpler than that illustrated in Figure 4-76 because we do not need to build .NET remoting proxies or stubs. It also has the benefit of not requiring a .NET CLR to be loaded into the client's JVM process. It delivers complete separation between Java and .NET.

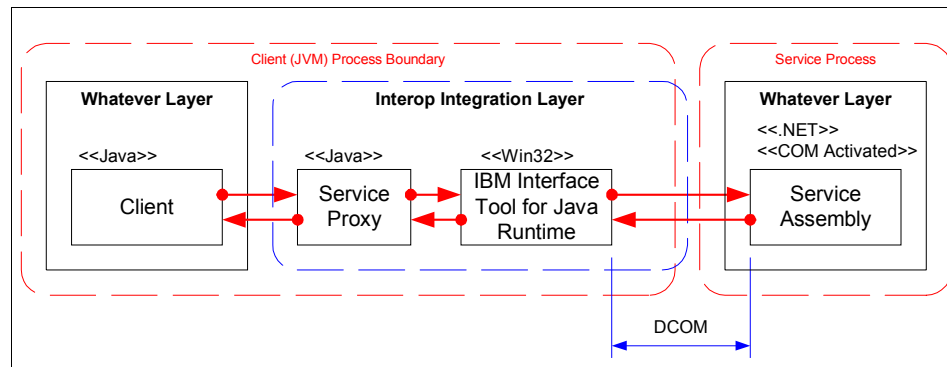


Figure 4-77 IBM Interface Tool for Java: inter-process solution candidate: DCOM

So far, each of these solution candidates has a dependency on the Windows OS. Both the .NET assembly and the Integration Tool for Java runtime require Windows. So, what do we do if our Java client code is deployed on another platform?

Figure 4-78 on page 208 illustrates a solution candidate that extends the model in Figure 4-75 on page 206, showing that we can design a solution where one JVM (this needs to be on a Windows OS) is used to isolate interop integration from the real Java clients executing on any Java supported OS. The interop JVM could be a regular JVM or a WebSphere JVM, depending on whether or not you use EJBs in your interop solution. Figure 4-79 on page 209 illustrates a similar solution candidate that extends the model in Figure 4-77.

Points to remember with the solution candidates in Figure 4-78 and Figure 4-79 on page 209 is that each time we put inter-process communication into our solution model, we add complexity and latency. Do not forget that this *can* be a pass by reference solution model. This *could* lead to some intensive chatty runtime interactions, which is probably not a good thing.

The solution candidate in Figure 4-78 uses EJBs as a QoS decorator to the .NET assembly. With this model, because the interop is in process, we can use WebSphere security to control access to the assembly. The solution candidate in Figure 4-79 on page 209 separates the JVM and the CLR, but may require both WebSphere and Java security models, which introduces another potential challenge.

As is so often true with software engineering problems, it may be a case of trading one feature for another to arrive at a satisfactory solution.

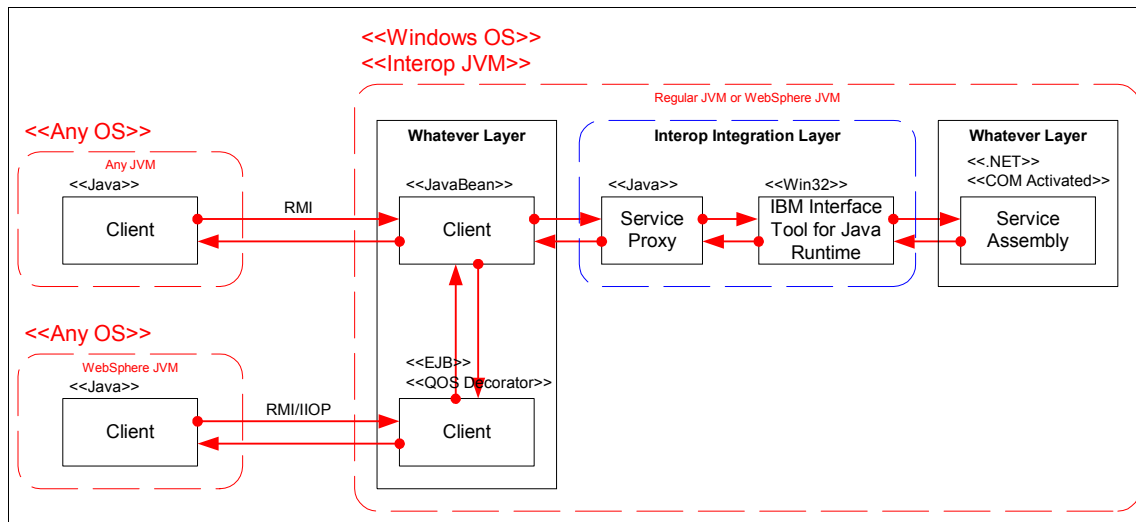


Figure 4-78 IBM Interface Tool for Java: inter-process solution candidate: separation of concerns by JVM

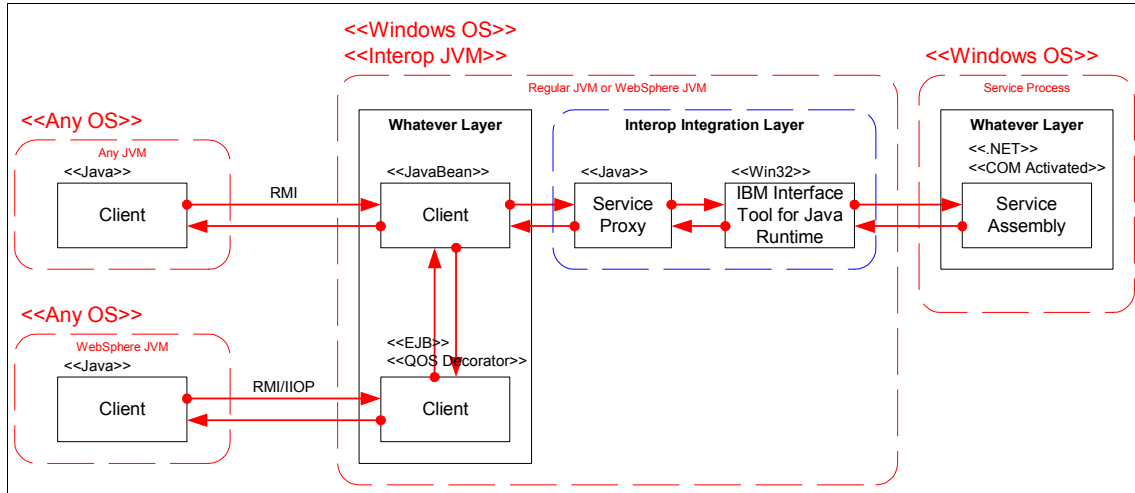


Figure 4-79 IBM Interface Tool for Java: inter-process solution candidate: separation of concerns by JVM.

Emerging solution candidate: Web Services WS-Addressing

Today, Web Services infrastructure (middleware) implementations do not have an interoperable standard for stateful synchronous interaction between a client and a service. It is probable that in the future, the WS-Addressing specification will provide this standard. This implies that Web Services are likely to become a strategic choice for stateful synchronous interaction for coexistent WebSphere and .NET applications.

See <http://www.ibm.com/developerworks/webservices/library/ws-add/> for more information on WS-Addressing.

Potential candidate: IIOP remoting channels

See 4.4.5, “Some last resource integration technologies” on page 226.

.NET applications perspective

Let’s consider stateful synchronous integration from the perspective of a .NET application artifact invoking a WebSphere application artifact. This is illustrated in Figure 4-80 on page 210.

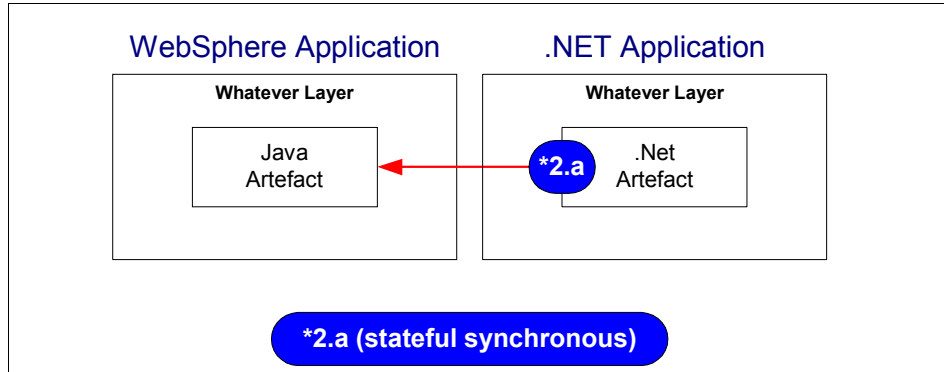


Figure 4-80 stateless asynchronous integration cases *2.a

The solutions we suggest for consideration will be relevant for the following interaction cases.

- ▶ **Case a2.a:** Stateful Synchronous Integration between Client layer and Client layer, from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case c2.a:** Stateful Synchronous Integration between Client layer and Business layer, from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case e2.a:** Stateful Synchronous Integration between Presentation layer and Business layer from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case f2.a:** Stateful Synchronous Integration between Business layer and Business layer from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case g2.a:** Stateful Synchronous Integration between Business layer and Resource layer from .NET to WebSphere: an occasional solution candidate for this interaction case if the resource is exposed locally as a Java or EJB implementation.

Solution candidate: IBM WebSphere ActiveX Bridge

Figure 4-81 on page 211 illustrates an overview of WebSphere ActiveX Bridge. WebSphere ActiveX Bridge provides a COM interface to the WebSphere Client Container. We can leverage .NET COM interop features to provide stateful .NET proxies to WebSphere Services and applications.

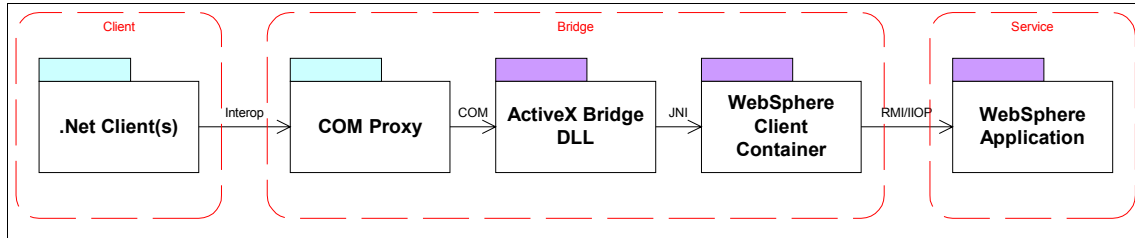


Figure 4-81 WebSphere ActiveX Bridge overview

In 4.2, “Fundamental interaction classifications” on page 111, we identified some characteristics for consideration when classifying the given interaction cases between coexistent WebSphere applications and .NET applications. Table 4-2 identifies if, and how, WebSphere ActiveX Bridge delivers these characteristics.

Table 4-2 WebSphere ActiveX Bridge characteristic matching

Characteristic	Supported	Comments
By Value argument paradigm	yes	-
By reference argument paradigm	yes	-
Document interface style	yes	-
RPC interface style	yes	-
Distributed object paradigm	yes	EJB
Service-oriented paradigm	no	-
Message-oriented paradigm	no	
Call context propagation	some	WebSphere ActiveX Bridge APIs exposes the J2EE client programming model to the client code. This enables the client to participate in WebSphere security.

Proxy Head Protocol	API	WebSphere ActiveX Bridge APIs exposes the J2EE client programming model to the client code.
Stub Tail Protocol	-	WebSphere ActiveX Bridge APIs exposes the J2EE client programming model to the client code.
Service Discovery Mechanism	JNDI	WebSphere ActiveX Bridge APIs exposes the J2EE client programming model to the client code. This enables the client to use JNDI.
Transport Mechanism	IIOP	-
Communication Protocol	RMI/IIOP	-
Service Binding/Hosting	WebSphere	WebSphere ActiveX Bridge APIs exposes the J2EE client programming model to the client code. Services are hosted as EJBs in WebSphere.
Open Standards	Java Community Standards	Exposes Java J2EE standards to the client.
Longevity	Tactical	As Web Services matures, it will probably become the strategic choice for this interaction characteristic.

Figure 4-82 on page 213 illustrates a candidate solution model using WebSphere ActiveX Bridge as the integration solution between .NET client code, and an application deployed in WebSphere. This solution model uses a COM service proxy to abstract WebSphere ActiveX Bridge from the .NET client code. The .NET client code binds to the COM service proxy using .NET COM interop. As you can see, with this solution, both a CLR instance and a JVM instance live inside the client process.

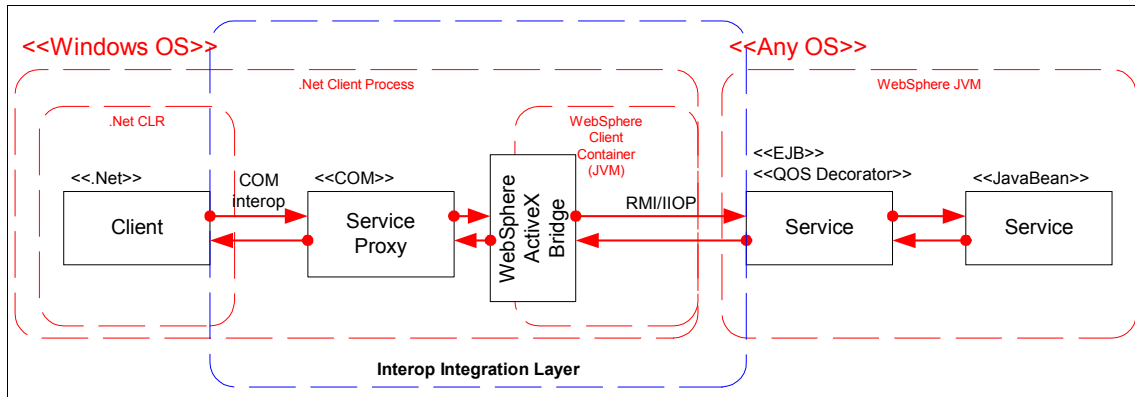


Figure 4-82 WebSphere ActiveX Bridge as the integration solution between .NET client code, and an application deployed in WebSphere

Figure 4-83 illustrates an alternative candidate solution model that separates the .NET CLR and COM into separate processes. As usual, separating these technologies into different processes can potentially have both benefits and disadvantages. Once again, you may need to trade one feature for another to arrive at a satisfactory solution.

Note: Readers may be interested to know that the CLR is implemented as a COM component. See the section “CLR Hosting” in the book *Applied Microsoft .NET Framework Programming*, by Jeffrey Richter, ISBN 0-7356-1422-9.

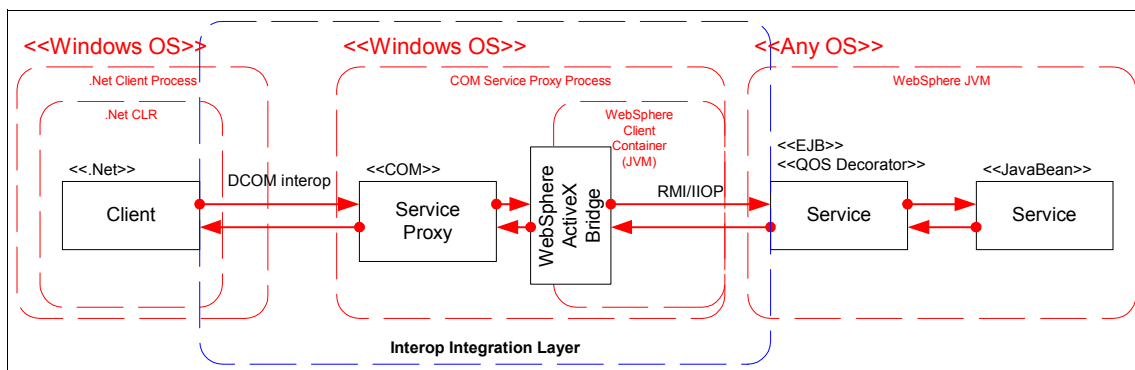


Figure 4-83 Alternative candidate solution model, separating the .NET CLR and COM processes

For other solution candidates, refer to the following sections:

- ▶ “Emerging solution candidate: Web Services WS-Addressing” on page 209
- ▶ “Potential candidate: IIOP remoting channels” on page 209

4.4.2 Stateless synchronous integration solution candidates

This section discusses the Stateless Synchronous integration solution candidates.

WebSphere and .NET applications

Let's consider stateless synchronous integration from the perspective of a WebSphere application artifact invoking a .NET application artifact, and from the perspective of a .NET application artifact invoking a WebSphere application artifact. These perspectives are illustrated in Figure 4-84 and Figure 4-85.

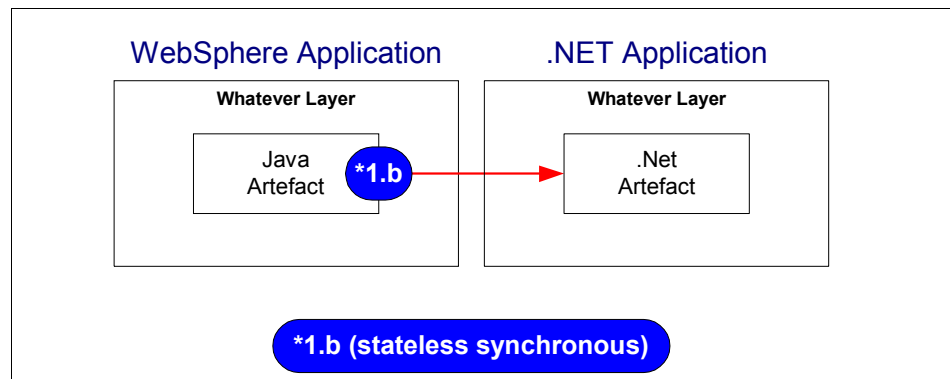


Figure 4-84 Stateless synchronous integration cases *1.b

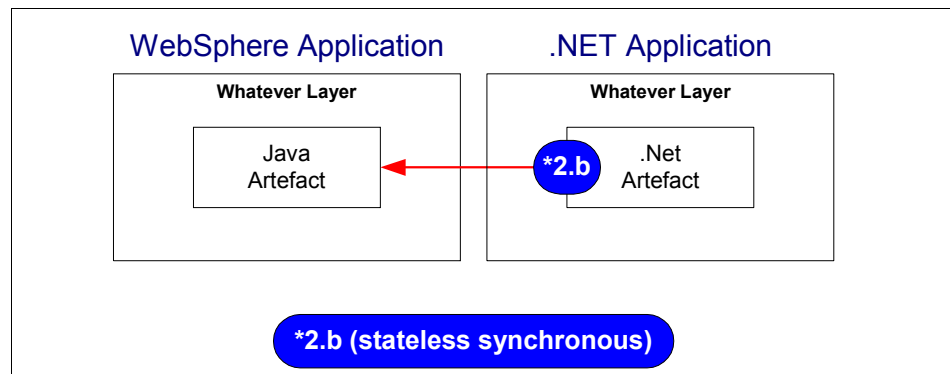


Figure 4-85 Stateless synchronous integration cases *2.b

The solutions we suggest for consideration will be relevant for the following interaction cases.

- ▶ **Case a1.b:** Stateless Synchronous Integration between Client layer and Client layer, from WebSphere to .NET: a recurrent solution candidate for this interaction case.
- ▶ **Case a2.b:** Stateless Synchronous Integration between Client layer and Client layer, from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case c2.b:** Stateless Synchronous Integration between Client layer and Business layer, from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case c1.b:** Stateless Synchronous Integration between Client layer and Business layer, from WebSphere to .NET: a recurrent solution candidate for this interaction case.
- ▶ **Case e1.b:** Stateless Synchronous Integration between Presentation layer and Business layer from WebSphere to .NET: a recurrent solution candidate for this interaction case.
- ▶ **Case e2.b:** Stateless Synchronous Integration between Presentation layer and Business layer from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case f1.b:** Stateless Synchronous Integration between Business layer and Business layer from WebSphere to .NET: a recurrent solution candidate for this interaction case.
- ▶ **Case f2.b:** Stateless Synchronous Integration between Business layer and Business layer from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case g1.a:** Stateless Synchronous Integration between Business layer and Resource layer from WebSphere to .NET: a frequent solution candidate for this interaction case.
- ▶ **Case g2.a:** Stateless Synchronous Integration between Business layer and Resource layer from .NET to WebSphere: a frequent solution candidate for this interaction case.

Solution candidate: Web Services

You can find a description of the Web Services technology in Chapter 10, “Supporting technologies” on page 429.

Figure 4-86 on page 216 illustrates an overview of Web Services from the perspective of Java client code deployed in WebSphere invoking a .NET service implementation deployed in the .NET Framework. Figure 4-87 on page 216

illustrates a similar overview from the perspective of .NET client code invoking a Java service implementation deployed in WebSphere.

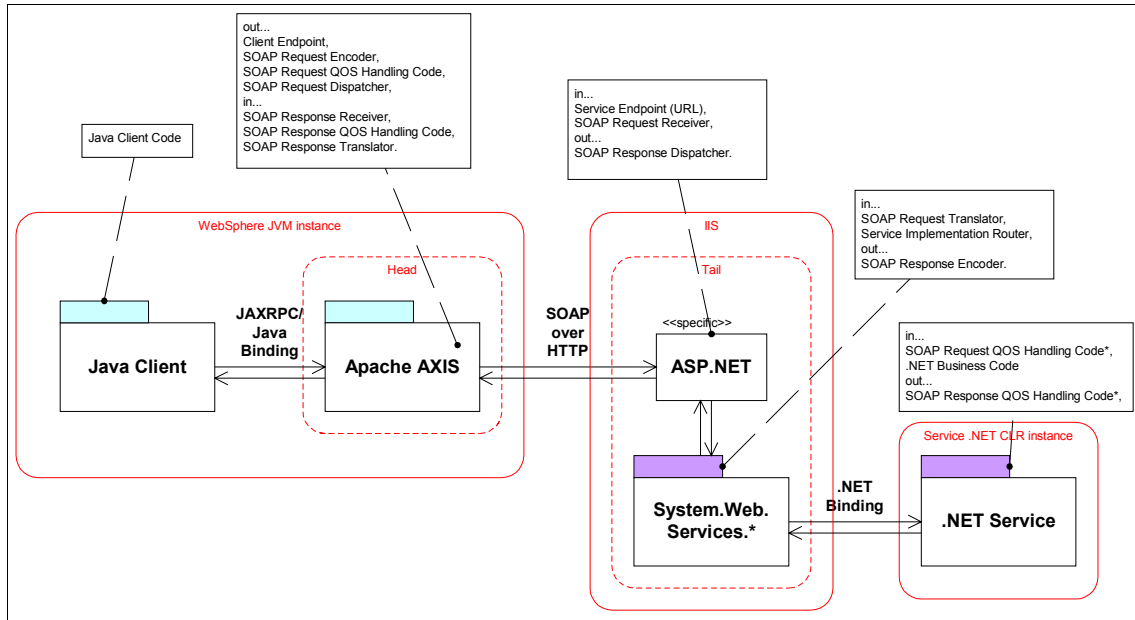


Figure 4-86 Web Services overview: WebSphere clients perspective

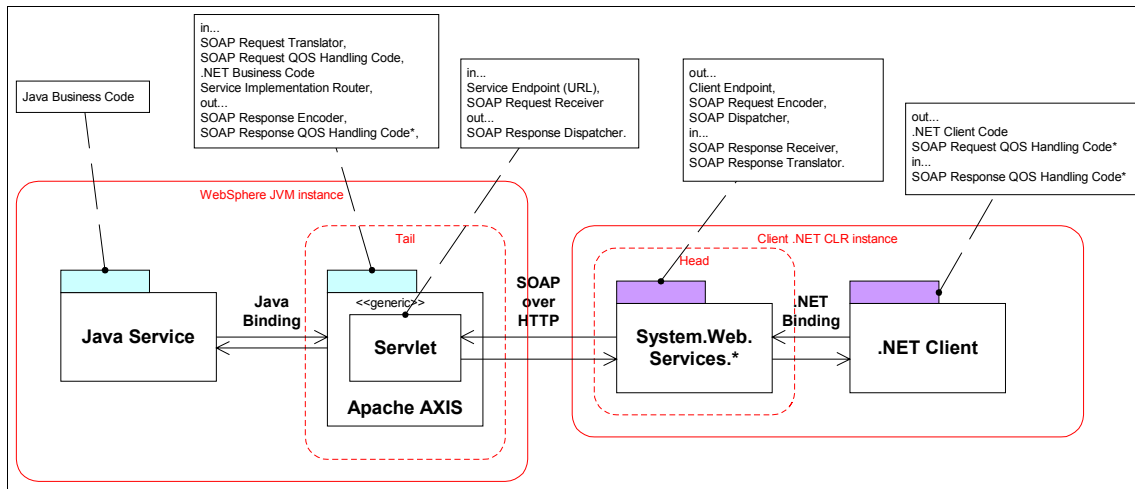


Figure 4-87 Web Services overview: .NET clients perspective

In 4.2, “Fundamental interaction classifications” on page 111, we identified some characteristics for consideration when classifying your given interaction cases

between coexistent WebSphere applications and .NET applications. Table 4-3 identifies if, and how, Web Services delivers these characteristics.

Table 4-3 Web Services characteristic matching

Characteristic	Supported	Comments
By Value Argument Paradigm	yes	Via [in] arguments (that is, using input message parts in WSDL).
By Reference Argument Paradigm	semantically	Web Services are not an object remoting technology, so they do not support true bidirectional references or callbacks. However, Web Services can use pass by value mechanisms to deliver pass by reference semantics via [in,out] arguments (that is, using paired input message parts and output message parts in WSDL).
Document Interface Style	yes	Web Services support compound types as operation arguments (message parts). Document style SOAP delivers messages on the wire as documents, but this is distinct from document interfaces (you can pass documents as arguments to a Web Service using RPC style SOAP).
RPC interface Style	yes	-
Distributed object paradigm	no	Web Services are a client and service implementation neutral integration technology. Objects are not relevant, they are implementation choices which should not be exposed at a WSDL interface.
Service-oriented paradigm	yes	-
Message-oriented paradigm	no	However, SOAP messages can be delivered using WebSphere MQ as a transport mechanism instead of HTTP.
Call context propagation	some / emerging	WS-Security is implemented in the most recent toolkit from IBM and Microsoft. WS-Transactions and WS-Coordination are evolving standards that will probably be implemented in future toolkits.
Proxy Head Protocol	API	WebSphere: JAX-RPC, Apache SOAP, Apache AXIS, Apache WSIF. .NET: System.Web.Service
Stub Tail Protocol	-	WebSphere: JAX-RPC, Apache SOAP, Apache AXIS, Apache WSIF. .NET: System.Web.Service

Characteristic	Supported	Comments
Service Discovery Mechanism	UDDI WSIL DISCO?	
Transport Mechanism	various	HTTP HTTPS WebSphere MQ, JMS others
Communication Protocol	SOAP	-
Service Binding/Hosting	-	WebSphere: the WebSphere Web Container hosts a single generic Apache routing Servlet, which binds to the service implementation using the Apache implementation. .NET: IIS host multiple service specific ASP.NET endpoints which bind to the specific service implementation.
Open Standards	yes	Both WebSphere and .NET interoperate at standards level with XML, XML Schema, SOAP and HTTP. WebSphere: the proxy programming model is to Apache.org open standards and the client programming model is to JAX-RPC Java standards. .NET: the proxy and the client programming model are Microsoft standards.
Longevity	Strategic	Web Services look certain to be the industry standard solution for integration between heterogeneous implementations.

Note: Web Services are still an evolving technology. Basic SOAP Web Services over HTTP have a formal measure of functional interoperability between WebSphere and .NET (see: <http://www.ws-i.org/>). At the time of writing, no formal measure of WS-Security interoperability between WebSphere and .NET exists, nor do we yet have implementations of WS-Coordination or WS-Transaction.

See <http://www.ws-i.org/> for WS-I Basic Profile guidelines for delivering interoperability between Web Service implementations, and tools for measuring the interoperability of Web Services.

Alternative solution candidate: WebSphere MQ

An alternative to the SOAP over HTTP solution is SOAP over MQ. Web Services do not provide all the capabilities that a solution may require, for example transactionality, delivery assurance, security, and so on. Switching to WebSphere MQ on the transport layer, we can get this quality of service from the messaging middleware.

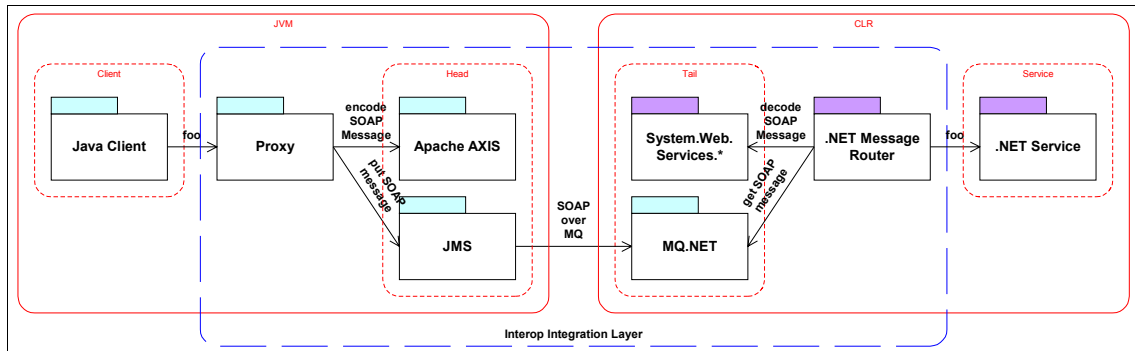


Figure 4-88 SOAP over MQ alternative solution

4.4.3 Stateful asynchronous integration solution candidates

This section discusses the solution candidates for asynchronous integration solutions.

WebSphere and .NET applications

Let's consider stateless synchronous integration from the perspective of a WebSphere application artifact invoking a .NET application artifact, and the perspective of a .NET application artifact invoking a WebSphere application artifact, as illustrated in Figure 4-89 on page 220 and Figure 4-90 on page 220.

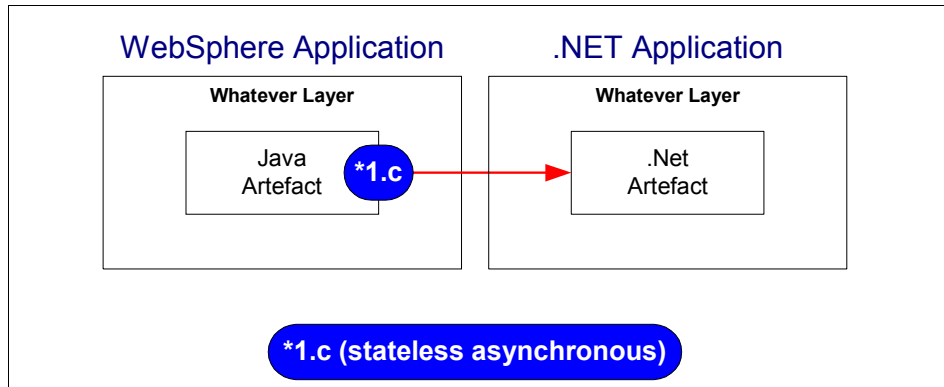


Figure 4-89 Stateless asynchronous integration cases *1.c

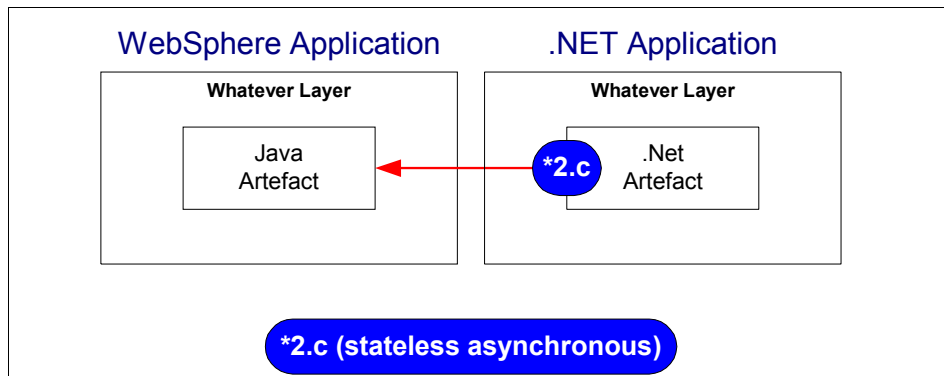


Figure 4-90 Stateless asynchronous integration cases *2.c

The solutions we suggest for consideration will be relevant for the following interaction cases.

- ▶ **Case a1.c:** Stateless Asynchronous Integration between Client layer and Client layer, from WebSphere to .NET: a recurrent solution candidate for this interaction case.
- ▶ **Case a2.c:** Stateless Asynchronous Integration between Client layer and Client layer, from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case c2.c:** Stateless Asynchronous Integration between Client layer and Business layer, from .NET to WebSphere: a recurrent solution candidate for this interaction case.

- ▶ **Case c1.c:** Stateless Asynchronous Integration between Client layer and Business layer, from WebSphere to .NET: a recurrent solution candidate for this interaction case.
- ▶ **Case e1.c:** Stateless Asynchronous Integration between Presentation layer and Business layer from WebSphere to .NET: a recurrent solution candidate for this interaction case.
- ▶ **Case e2.c:** Stateless Asynchronous Integration between Presentation layer and Business layer from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case f1.c:** Stateless Asynchronous Integration between Business layer and Business layer from WebSphere to .NET: a recurrent solution candidate for this interaction case.
- ▶ **Case f2.c:** Stateless Asynchronous Integration between Business layer and Business layer from .NET to WebSphere: a recurrent solution candidate for this interaction case.
- ▶ **Case g1.c:** Stateless Asynchronous Integration between Business layer and Resource layer from WebSphere to .NET: an frequent solution candidate for this interaction case.
- ▶ **Case g2.c:** Stateless Asynchronous Integration between Business layer and Resource layer from .NET to WebSphere: an frequent solution candidate for this interaction case.

Solution candidate; IBM WebSphere MQ

WebSphere MQ is the ideal solution for asynchronous integration of applications. WebSphere MQ is available on several different platforms and provides all the necessary functions and services required for asynchronous messaging. For more information about WebSphere MQ, refer to the IBM Web site at:

<http://www-306.ibm.com/software/integration/wmq/>

You can also search for WebSphere MQ Redbooks on:

<http://www.redbooks.ibm.com>

In 4.2, “Fundamental interaction classifications” on page 111, we identified some characteristics for consideration when classifying your given interaction cases between coexistent WebSphere applications and .NET applications. Table 4-4 on page 222 identifies if, and how, WebSphere MQ delivers these characteristics.

Table 4-4 WebSphere MQ characteristic matching

Characteristic	Supported	Comments
By value argument paradigm	yes	-
By reference argument paradigm	no	Decoupled systems are not prepared for this feature
Document interface style	yes	Messaging by default can handle this feature
RPC interface style	no	Not standardized
Distributed object paradigm	no	These are decoupled systems, distributed applications are not the primary objective
Service-oriented paradigm	no	-
Message-oriented paradigm	yes	-
Call context propagation	possibly	It can be implemented
Proxy Head Protocol	-	
Stub Tail Protocol	-	
Service discovery mechanism	-	
Transport mechanism	-	
Communication protocol	-	
Service binding/hosting	-	
Open standards	no	Java standards (JMS) on the Java side. MQ de facto standards on the .NET side.

Characteristic	Supported	Comments
Longevity	Strategic	-

Solution candidate: Web Services - asynchronous façade

Web Services would be an ideal solution candidate for asynchronous communication between different platforms. As of today, the technology is not there yet, and asynchronous Web Services as a standard implementation do not exist. It is the implementor's responsibility to develop Web Services that support asynchronous communication. For more information about asynchronous Web Services, refer to:

<http://www-106.ibm.com/developerworks/webservices/library/ws-asynch1.html>

4.4.4 Other potential candidate technical solutions (to be proven)

These technical solutions may be worth further consideration, but have not been exercised in this redbook.

Apache WSIF

WSIF is an Apache.org open source project which enables developers to interact with abstract representations of Web Services through their WSDL descriptions instead of working directly with the SOAP APIs, which is the usual programming model. With WSIF, developers can work with the same programming model regardless of how the Web Service is implemented and accessed. Consequently, WSIF delivers a *Java to Many* model to a Java client implementation, with bridging and routing provided by WSIF, and configured via WSDL.

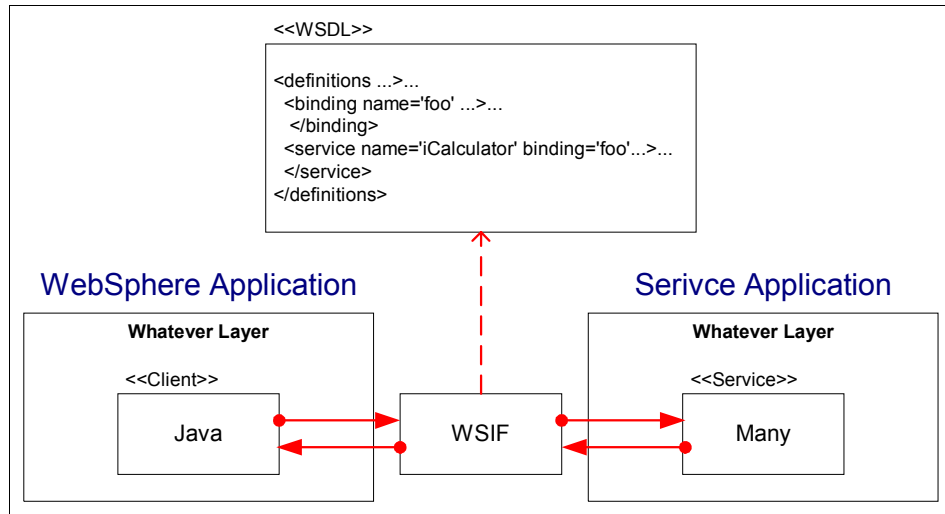


Figure 4-91 WSIF delivers a Java to Many model

See: <http://ws.apache.org/wsif/> for more details about WSIF.

Because WSIF abstracts the client from the services via WSDL, WSIF has the potential to help with both coexistence cases and managed migration cases. Figure 4-92 on page 225 illustrates WSIF used as a coexistence bridge between Java and .NET. Figure 4-93 on page 225 illustrates that WSIF can be used as part of a migration strategy to defend the client implementation from change when a service is migrated to WebSphere.

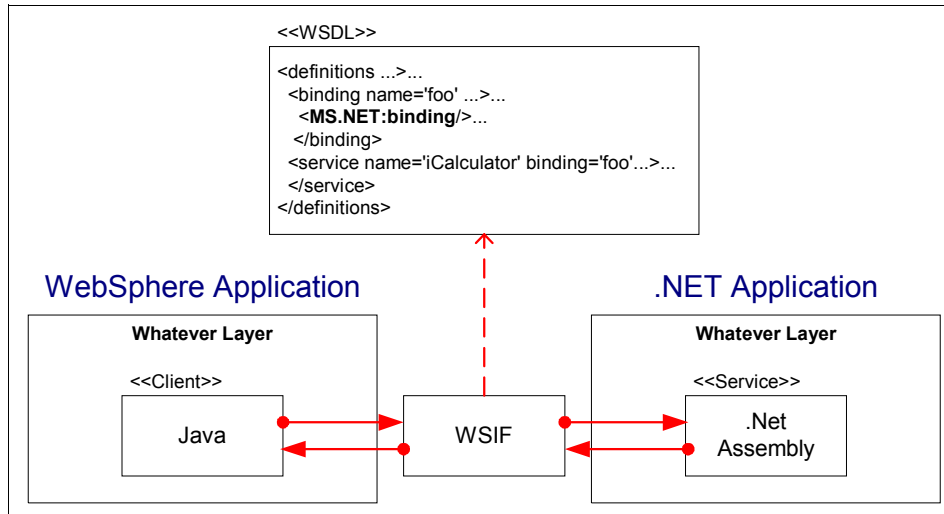


Figure 4-92 WSIF used as a bridge, routing between a Java client and a .NET Service implementation via WSDL

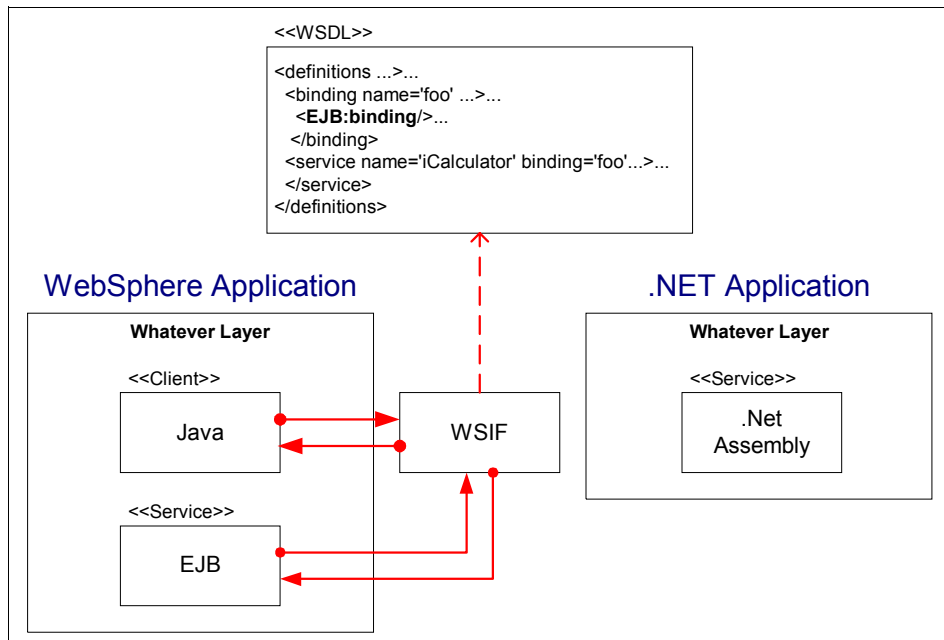


Figure 4-93 WSIF used as a bridge, routing between a Java client and an EJB Service implementation via WSDL (without modification of the client implementation)

IIOp.NET IIOp.NET is a SourceForge.net open source project which provides bidirectional interoperability between .NET, CORBA and J2EE distributed objects. IIOp.NET implements a CORBA/IIOp remoting channel for the .NET Framework.

See <http://iio-net.sourceforge.net/> for more information about IIOp.NET.

IIOp.NET is released under the LGPL license; see:

<http://www.gnu.org/copyleft/lesser.html>

Ja.NET

Ja.NET is a commercial product from Intrinsic. Ja.NET includes a .NET remoting implementation for .NET clients to bridge to J2EE and CORBA implementations over IIOp.

See <http://ja.net.intrinsyc.com/ja.net/info/> for more information about Ja.NET.

The JNBridge **JNBridge**

JNBridge is a commercial product JNBridge. JNBridge includes a .NET remoting implementation for .NET clients to bridge to J2EE and CORBA implementations over IIOp.

See <http://www.jnbridge.com> for more information about JNBridge.

Janeva

Janeva is a commercial product from Borland. Janeva includes a .NET remoting implementation for .NET clients to bridge to J2EE and CORBA implementations over IIOp.

See <http://www.borland.com/janeva/> for more information about Janeva.

SpiritWave

SpiritWave is a commercial product from SpiritSoft. SpiritWave includes a JMS provider for Microsoft MSMQ.

See <http://www.spirit-soft.com/products/wave/introducing.shtml> for more information about SpiritWave.

4.4.5 Some last resource integration technologies

This list of technologies is provided for completeness. If you use any of these technologies to provide a technical solution, then you are writing your own middleware. This means that you will be responsible for providing and maintaining

all (or most) of your required solution characteristics, as shown in Figure 4-13 on page 137.

- ▶ Raw Sockets (both directions)
- ▶ RMI to JNI/NIO (Java to .NET)
- ▶ RMI/IIOP to JNI / NIO (WebSphere EJBs to .NET)
- ▶ JNI/NIO to RMI (.NET to Java)
- ▶ JNI/NIO to RMI/IIOP (.NET to WebSphere EJBs)

Please do not underestimate what might be involved in delivering a solution using these low-level technologies, even for single case, short term, very tactical solutions.



Scenario: Asynchronous

This chapter deals with the asynchronous interaction scenarios defined in 4.2.3, “Stateless asynchronous interaction” on page 117 and 4.2.4, “Stateful asynchronous interaction” on page 120 where the consumer is a .NET application component and the service provider is a WebSphere component.

We will explore the problem spaces associated with these types of interactions, define the solutions that exist to meet these problems and provide a few basic examples of these solutions in action.

A great deal has been written about exactly how to code applications that communicate asynchronously between these platforms. Our discussion will, therefore, focus on the pros and cons of various methods from a solution point of view rather than on the mechanics of the code in most instances.

For more details on asynchronous interactions between .NET and WebSphere applications, the reader is referred to the IBM Redbook *WebSphere MQ Solutions in a Microsoft .NET Environment*, SG24-7012.

5.1 Problem definition

We have been discussing interactions in the context of .NET and WebSphere coexistence. However, in the business and technology worlds alike, there are many places where asynchronous interactions are intuitively useful. Wherever there are needs, solutions inevitably rise to meet them and these needs are no different.

Message-oriented middleware has been used for decades now to provide a mechanism for multi-platform integration which, by its nature, lends itself to asynchronous use. WebSphere MQ is the most prolific middleware product in the market in terms of market share, platform and language support. Companies all over the world use MQ for business-critical integration and e-business and to preserve and reuse past investments in systems that work. WebSphere MQ is defined in more detail in “WebSphere MQ” on page 464.

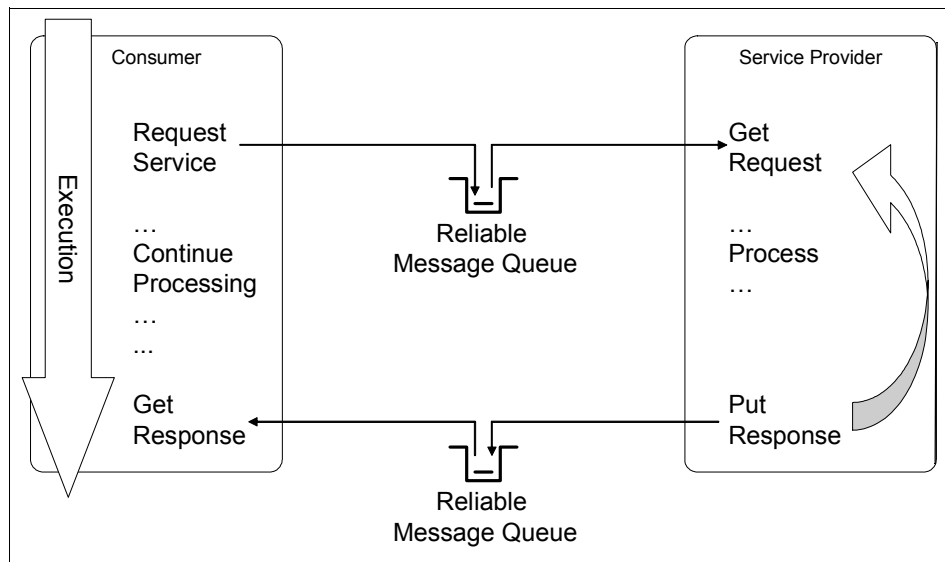


Figure 5-1 Asynchronous message queuing

WebSphere MQ implements highly reliable asynchronous message queuing and takes on the role of ensuring that messages sent from one program to another arrive at their destination exactly once, without loss or corruption of the message data.

Outside of message-oriented middleware and message queuing, asynchronous programming techniques have also been around a long time and have specific uses, mostly related to performance. As a programming technique,

asynchronous interaction requires the use of threads. *Callback functions* and *delegation threads* are the usual tools of the trade for implementation.

A callback function is simply a function that handles the response of an asynchronous service provider. The consumer or service provider will create a separate delegation thread via some mechanism and then start that thread off executing some code. Once the thread is off and running, the service provider immediately returns control of the main thread back to the consumer. The consumer then continues processing further work while the service provider works on the request.

The service provider then calls the consumer's callback method as the last (or nearly the last) step in the process of providing the requested service. Sometimes, the callback function is passed the response, sometimes it is used as a mechanism to alert the consumer that the response is ready somewhere else.

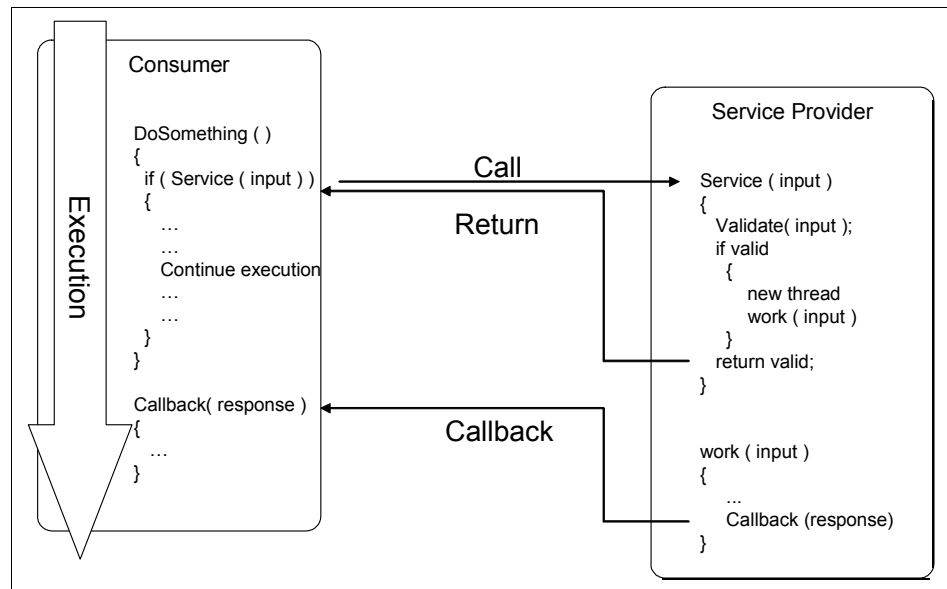


Figure 5-2 Asynchronous calling paradigm

Most people associate asynchronous interactions with *stateless* interactions, and, in fact, most asynchronous interactions are indeed stateless. However, *asynchronous* does not imply statelessness by necessity and sometimes there are very good reasons to maintain state information within asynchronous interactions.

Interactions are considered to be stateful when information is kept about the current state by some mechanism and operations being performed by any given actor at any given time depend upon that information in some way. For instance, in the stateful implementation, the calculator application keeps the current total in a private variable in local memory during execution and uses each invocation of each method. This common technique for implementing statefulness requires two things:

1. That the service provider continue to exist between invocations.
2. That the same service provider be used for each `add()` operation.

But state is, in the end, merely information and order. If the information representing state can be stored in the messages through which interaction occurs, then some new possibilities emerge with regard to state. Take our calculator, for example. If we represented the current total in the message itself, then the fact that the service provider's *act* of calculation occurs asynchronously (at the same time that the consumer is doing other things) is not really relevant.

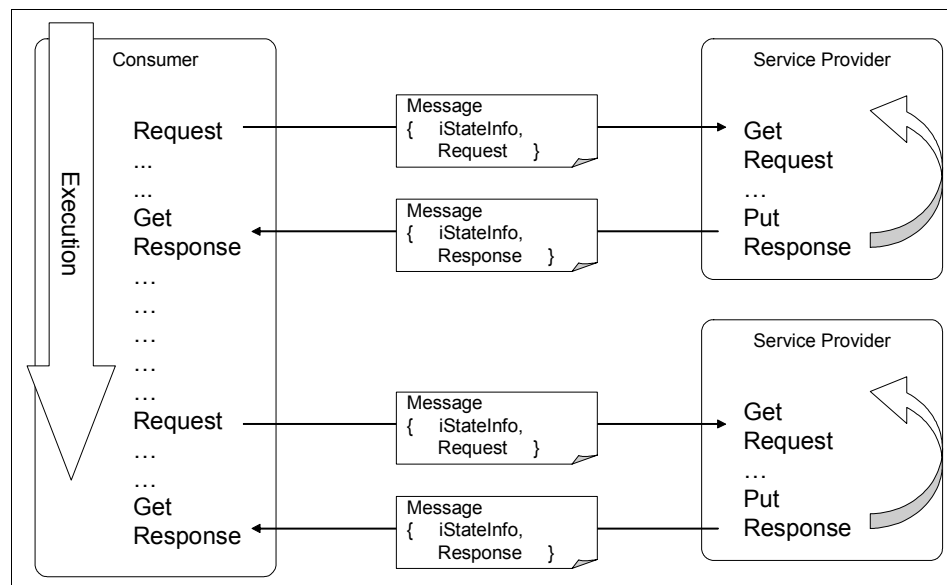


Figure 5-3 Stateful asynchronous works across multiple service providers

Consider Figure 5-3 and realize that it is not even relevant now which instance of the service provider we utilize as long as the service itself is consistent. You may notice that our service provider is stateless now. If we define stateless as “having all information necessary to perform the business operation,” then the service provider might be considered stateless in this case.

definition of asynchronous interactions between service consumers and providers.

From here, our focus will be to provide some details related to asynchronous interoperability between WebSphere and .NET.

5.1.1 Description of the problem

This chapter addresses how application artifacts deployed in a .NET environment may request the services of application artifacts deployed in a WebSphere Application Server environment using asynchronous interaction and vice-versa.

We will focus on stateless services, show a simple example of message-oriented asynchronous interaction between a .NET application using the classes for WebSphere MQ and a message-driven bean implemented under WebSphere using JMS.

We will focus on reuse of the Calculator class written in Java for WebSphere that we have been using in these scenario examples. We will choose to utilize one of the stateless interfaces we created and access it in an asynchronous fashion. Our consumer will be a .NET application. Whether it is an Active Server Page.NET or a fat client really is not relevant to the interaction.

We will then discuss the same scenario, but accessing the .NET implementation of Calculator from a J2EE client.

5.1.2 Considerations

This section describes the considerations should be taken into account when designing a stateful asynchronous application.

Transports

The transports available for asynchronous communication between .NET and WebSphere application artifacts include the following:

- ▶ WebSphere MQ provides a highly reliable messaging transport and middleware for over 35 platforms.
- ▶ The MQ Transport for SOAP provides a transport for SOAP requests used in Web Services. This topic is covered extensively in the IBM Redbook *WebSphere MQ Solutions in the Microsoft .NET Environment*, SG24-7012.
- ▶ HTTP: the Web Services Description Language (WSDL) supports definition of interaction between the service consumer and provider. By using document invocation style and the operation modes one-way, as well as notification to

define the interaction messages within the WSDL definition, Web Services interaction can be implemented in an asynchronous fashion.

Interfaces

The following two interfaces are available for implementation.

Java Messaging Service

The Java Messaging Service (JMS) provides an interface into WebSphere MQ for Java programs. It is discussed in more detail in “Java Messaging Service” on page 235.

MQ Classes for .NET

IBM provides an unsupported (Category 2) SupportPac™ for WebSphere MQ that contains .NET Classes for accessing MQ from within managed .NET assemblies. This support pack requires MQSeries V5.2, V5.2.1 or WebSphere MQ V5.3.

This SupportPac, **MA7P**, and instructions for download and installation, can be found at the following link:

<http://www-3.ibm.com/software/integration/support/supportpacs/individual/ma7p.html>

The MQ Classes for .NET provide an easy to use interface for .NET applications written in Visual Basic.NET, C# as well as C++ with Managed Extensions. In addition, an ActiveX wrapper is provided to help migrate existing Visual Basic V6 (pre-.NET) MQ applications that use the MQ ActiveX interface. For more information, see the documentation that is provided with the service pack.

Note: The MQ Classes for .NET do not include any JMS implementation. You either have to use WebSphere MQ messaging or you have to create your own JMS messages programming with the existing classes.

If you want to create your own JMS messages, header and payload, refer to the WebSphere MQ product guide: *WebSphere MQ Using Java*, SC34-6066-01.

Security

If it is worth doing, it is probably worth protecting. Security for asynchronous interaction is dependent somewhat upon the technologies used to implement the interactions. Therefore, if security is important for your application, it should be one of the key considerations for choosing such technology.

WebSphere MQ, for example, has advanced security features to protect messages being transmitted between application artifacts, for authentication, authorization and encryption which extend across platforms. For more information on security for asynchronous interactions using WebSphere MQ, please see the IBM RedBook *WebSphere MQ Security in and Enterprise Environment*, SG24-6814.

Transactions

In a synchronous interaction, where the consumer waits for the service to be provided, it is relatively easy to imagine transaction boundaries that make intuitive sense as being, “from the time the request is successfully placed to the time the response is successfully received.” However, in an asynchronous interaction, where the request and response can be separated by days and responses may not even be necessary or desired, transaction boundaries are not so obvious.

In an asynchronous interaction, the scope of the transaction is not managed by the consumer, but by each actor in the interaction, with the boundaries being the interaction point itself.

Asynchronous interaction is primarily accomplished by allowing each interaction between any application artifacts to be an entirely self-contained unit. As such, each interaction can become a stand-alone “Unit Of Work” in the transactional sense. In other words, the objective of the Transaction Coordinator being used changes from “ensuring all the work was done” to something like “ensuring that the request was placed successfully for the consumer” and “ensuring that the request was received, the work was done and the response was sent successfully for the provider.”

So if, in asynchronous interactions, transactions boundaries no longer cover the entire interaction as they do in synchronous interactions, then does not that leave a gap? The answer is, *yes*.

If you are using a reliable transport such as WebSphere MQ, which provides assured message delivery, then this kind of transactional support is meaningful and provides powerful and flexible transactional support across platforms.

If you are using other technologies to implement asynchronous interactions, then corresponding choices will be available for transactional support and corresponding transactional boundaries are affected very little. That is, an asynchronous Web Service, for instance, that is implemented under WebSphere, might perform a series of functions that are bound within a transactional Logical Unit of Work, but the entire interaction with the consumer is not covered by this transaction, only the work performed by the single Web Service.

In either case, you must recognize that it is now possible for part of the interaction to succeed and part of it to fail. When a service consumer successfully submits a request and this request fails to process, additional action must be taken to notify the consumer of this failure asynchronously if the consumer cares about success.

5.2 Solution model

This section discusses the design and implementation of our solution to the stateless asynchronous interaction from application artifacts in a WebSphere Application Server application to application artifacts in a Microsoft .NET application and vice versa.

The solution provided in this section restricts itself to providing a simplified implementation that illustrates how asynchronous interoperability can be achieved between WebSphere Application Server and Microsoft .NET Business layer components. As such, we will reuse the Calculator class we created in previous scenarios and access it in an asynchronous fashion.

So, our problem domain is this: allowing a Business layer .NET assembly to invoke our Java Calculator via an asynchronous mechanism, and allowing a Business layer Java application to invoke our .NET Calculator via an asynchronous mechanism.

This section contains the following subsections:

1. A solution to the problem

Here we illustrate our solution design, including the rationale for some of the design decisions taken. We then continue by presenting the details of our implementation.

2. Simple scenario details

Here we describe the scenario we will be implementing as our solution.

3. .NET Consumer to WebSphere Service Provider

Here we detail the implementation of the scenario from the .NET consumer making an asynchronous call to a WebSphere service.

4. WebSphere Consumer to .NET Service Provider

Here we detail the reverse of the above scenario - a WebSphere consumer making an asynchronous call to a .NET service.

5.2.1 A solution to the problem

For our transport, we have chosen to illustrate asynchronous messaging between these environments using WebSphere MQ. Since the MQ Transport for SOAP was covered in the redbook *WebSphere MQ Solutions in the Microsoft .NET Environment*, SG24-7012, we will not use Web Services technologies, but stick to simple asynchronous messaging. This leaves us with the following remaining design decisions:

- ▶ The interface that will be used to access WebSphere MQ in each case.
- ▶ Message payload formats.

Interface

On the .NET side, the interface between a client and service provider will use the *MQ Classes for .NET*.

On the WebSphere side, since J2EE provides value-added mechanisms for the Service Provider with regard to messaging, we will use different techniques for the consumer and service providers. The WebSphere consumer will use the Java Messaging Service to put a request message and then obtain the response. The WebSphere service provider will be implemented as a Message-Driven Bean.

Message format

Since our goal is not to provide a complex messaging tutorial, we will keep our message format simple for our interaction. We will focus on stateless interaction using the `add(a1, a2)` method of our calculator where `a` and `b` are the two numbers we wish to add together.

We design two messages; a *request* and a *response*.

A request message is sent from the consumer to the service provider in order to request service. It should contain all the information required to complete the service being requested. Our request will be a simple comma-delimited string, for example: "2,2". Everything before the comma is the first argument. Everything after the comma is the second argument. The message "2,2" will cause two to be added to two.

A response message is sent from the service provider back to the consumer in order to provide service. It should contain everything required to provide the service being requested. In our case, we will choose to return the original request message followed by a comma and then the response. So, our sample message above would provide the response "2,2,4" from our addition calculator. Another example would be: `request(2,5)`, `response(2,5,7)`.

We have not designed in any error handling. In the real world, message definitions include contracts depicting how consumers and providers will communicate when errors occur and so forth.

Message headers

Having defined our payload, we must be aware of one other key item with regard to messaging technologies and how they interact. The most key concept to understand is the concept of *message headers*. Message headers are simply prefaces to a message payload that provide information. Although applications and business objects define their own headers (and almost always do) to contain key information such as error handling, many of the underlying technologies also provide headers.

WebSphere MQ, for instance, has several headers that it uses to provide information about the destination of a message, to whom the response should be sent, the platform on which the message originated and many other items. Although they are accessible by the application, these headers are largely invisible to two applications using WebSphere MQ to communicate.

However, differences arise when a technology used on one side creates a header and that same technology is not used on the other side. In our case, we have decided to use the Java Messaging Service as our interface to MQ from the Java side. As shown in Figure 5-5, the Java Messaging Service, by default, creates its own within the message. The JMS Header provides useful information to the Java Messaging Service runtime about how to handle this message.

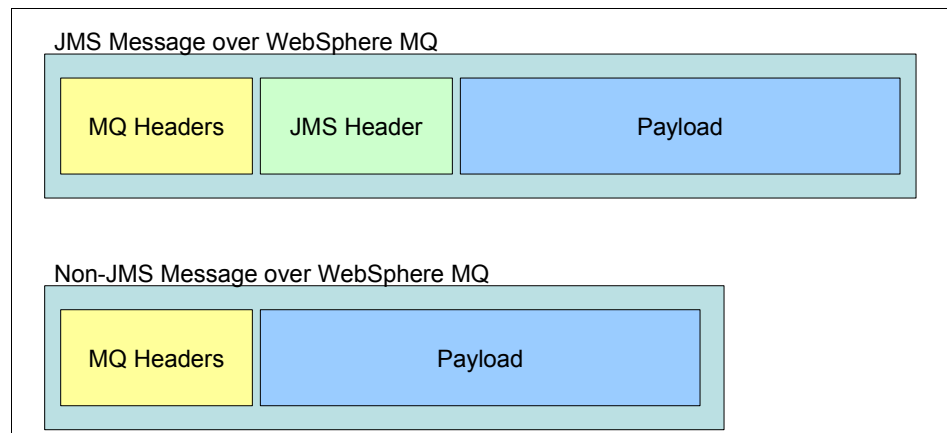


Figure 5-5 The JMS Header and its impact on message layout

Headers are created when messages are put to a queue. When a message is read from the queue each layer of technology strips away its respective headers as it provides information, sending the remaining payload to the layer above.

There are three approaches to consider:

- ▶ **Message transformation**

A message broker such as WebSphere Integrator can be used to transform the message on the fly as it travels from consumer to provider and back. Headers can be added and removed here as appropriate. This has the advantage of keeping the interactions between the application artifacts much more loosely coupled.

- ▶ **Drop the header**

This requires that the technology creating the header allows you to do this and that the header itself provides no useful function in the case of the implementation. In our case, JMS is built as an open messaging standard interface that does not assume that WebSphere MQ or any product is running underneath it. Hence, there is some duplication of functionality between the JMS Header and the MQ Header. This is the option we will use.

In our solution, we decode the MQ message and extract the JMS Header information.

- ▶ **Ignore the header in the payload**

This requires that the header be defined to the other application as part of the payload. While this has the advantage of maintaining the header, such maintenance is rarely useful given that headers are usually meant to be manipulated on both sides. This has significant disadvantages and is seldom recommended.

5.2.2 Simple scenario details

In our simple asynchronous implementation, the required processing of the consumer and provider are easy to delineate.

Service consumer

The consumer's responsibilities include:

- ▶ Formatting a request message properly
- ▶ Sending the request message
- ▶ Obtaining the response message
- ▶ Understanding the response message

Service provider

The service provider's responsibilities include:

- ▶ Getting the message from the queue
- ▶ Obtaining the information required from the request message
- ▶ Business function execution

- ▶ Utilizing any required transactional support
- ▶ Formatting the response message properly
- ▶ Sending the response message

System prerequisites

In order to build and execute the examples provided here, the following software packages must be present on the corresponding systems. If both components will be executed on the same system, then install all these products.

Version information corresponds to the versions we used in our example, not minimum configuration requirements. For minimum configuration requirements for these components, please refer to product documentation.

The .NET System

- ▶ Microsoft .NET Framework v1.1
- ▶ Microsoft Visual Studio.NET v1.1
- ▶ IBM WebSphere MQ v5.3
- ▶ Service Pack MA7P, which can be downloaded free from IBM at the following address:

<http://www-3.ibm.com/software/integration/support/supportpacs/individual/ma7p.html>

The WebSphere system

- ▶ IBM WebSphere Studio Applications Developer V5.1
- ▶ IBM WebSphere MQ v5.3

In addition to the above software, an environment needs to be created within WebSphere MQ, including queue managers and queues as well as MQ channels if multiple queue managers are being used. We do not cover creation of these system objects here. For more information on how to perform these administrative functions, see the WebSphere MQ product documentation.

Notes on installing MQ Classes for .NET

The MQ Service Pack MA7P installation is described by the Web site providing the download. The Service Pack uses the Microsoft Installer to install itself, so you download an MSI file and execute it like an application.

Chances are you have a virus protection package installed. During installation in our vast and highly secure laboratory environment, we encountered a message that indicated that a “malicious script” had been detected during installation. This is caused by a script which registers a required .NET assembly into the *Global Assembly Cache* using a command line utility.

Once the service pack is installed, you will find several code examples for using the MQ Classes for .NET in the following directory:
<WebSphere_MQ_Root>\Tools\dotnet\

Under this folder, there is a samples directory as well as some documentation for the classes. Launch the DotNetClasses.chm file to find this documentation. The documentation provides a simple and concise reference for the classes supported and how they are intended to be utilized. Samples are provided for all basic operations possible using the classes.

5.2.3 .NET consumer to WebSphere service provider

This section describes an implementation of asynchronous interaction from a Microsoft .NET service consumer invoking a WebSphere service. We begin by describing how to implement the WebSphere service, then discuss how the service can be invoked from a .NET environment.

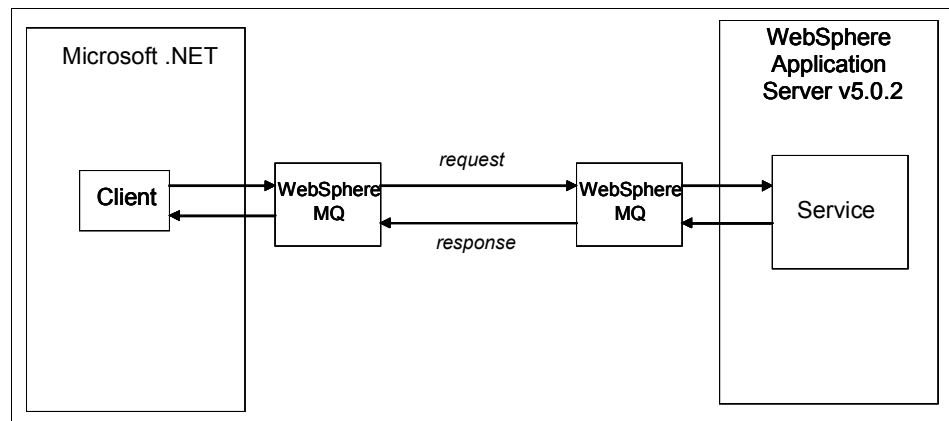


Figure 5-6 .NET client invoking WebSphere service

The service provider and consumer will normally reside on different machines, linked by a network, and connected to different queue managers. Messages can be sent between different queue managers by defining remote queues. This is shown in Figure 5-7 on page 243.

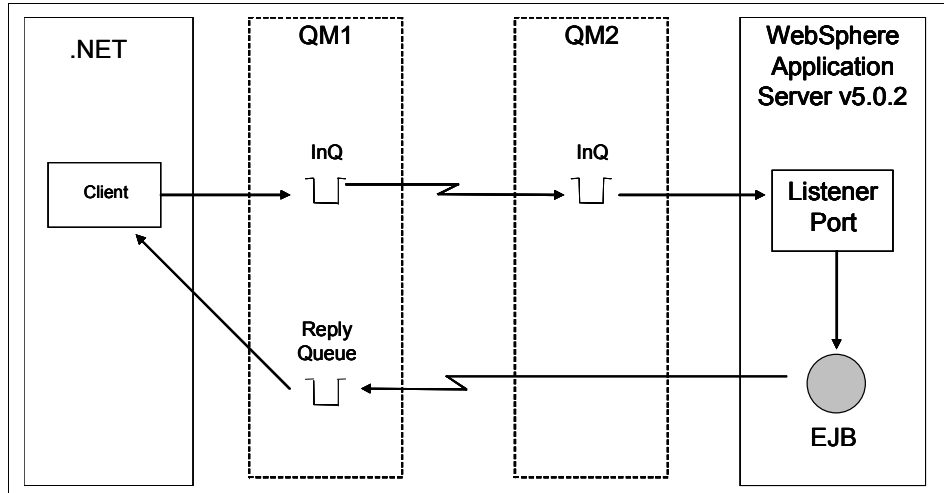


Figure 5-7 Accessing remote queue managers

The .NET client, using the MQ Classes for .NET, puts a message onto an MQ queue. The message is then routed by WebSphere MQ to a remote instance of the queue. This queue is then detected by a listener on the WebSphere service provider, which invokes the back-end service and puts the result back onto a queue.

In the scenario above, the .NET client talks to its local queue manager, QM1. This queue manager has a remote queue definition for a queue managed by another queue manager, QM2. The .NET client puts a message onto the remote queue, and the WebSphere Application Server application retrieves the message by communicating with its queue manager.

We take a look at how to implement the above flow to call the code implemented in the Calculator example described earlier.

We first need to create a J2EE service in the WebSphere environment, and then look at how to use the MQ Classes in order to call it.

Service provider (WebSphere service)

We assume that the business logic for the Calculator application is pre-existing code accessible from the WebSphere service provider. This code is contained in or Calculator.jar file included in the additional material.

We need the Calculator code to be activated upon receipt of a message in a specified queue. This can be achieved using Message Driven Beans (MDBs).

Message driven beans are stateless, server-side, transaction-aware components for processing asynchronous JMS messages.

Message driven beans listen on a queue for receipt of a message. Upon receipt, the `onMessage()` method of the bean is automatically executed.

While the message driven bean is responsible for the actual processing of the message, quality of service items such as transactions, security, resources, concurrency and message acknowledgement are all handled automatically by the bean's container, meaning the developer can focus on implementing the actual business logic of the server side application.

So, in order to enable our existing Calculator code as an asynchronous service, we need to create a message driven bean front end to the application. We shall assume for simplicity of discussion that the only method of the Calculator class we are interested in will be the `add(float, float)` method, and the message format being sent by a consumer will consist simply of a text message "arg1,arg2". So, for example, if the consumer wishes to add 4.0 and 3.7 using the asynchronous service, it would create a text message "4.0,3.7" and place it on the appropriate queue.

Creating a message driven bean can be achieved easily using wizards within WebSphere Studio Application Developer. Message driven beans need to be contained in an EJB module.

1. Create a new EJB project under an existing or a new enterprise application (J2EE V1.3).
2. Right-click the EJB project, and from the context menu select **New -> Enterprise Bean**. Ensure the correct EJB project is selected, and on the next screen you should select **Message-driven bean**. Enter appropriate values for the Bean name and Default package, for example: `redbook.coex.async.stateless.TestMDBBean` and select **Next**. You should enter the name of the ListenerPort which you will later define in WebSphere Application Server, for example: `TestLP`. Click **Finish**.

You will see that a new message driven bean has been created, and within the bean there is a method `onMessage(javax.jms.Message msg)`. It is this method which will be executed automatically when a message is detected on the queue.

3. We want the business logic to call the `add(float, float)` method of the CalculatorService class in the Calculator.jar file. The EJB project needs to reference the Calculator.jar file, so open up the properties page for the project, go to **Java build path -> Add External JARs** and add Calculator.jar to the build path.

4. Back in the `onMessage()` method, enter the following code.

Example 5-1 Implementation of the `onMessage()` method

```
public void onMessage(javax.jms.Message msg)
{
    CalculatorService s = new CalculatorService();
    try {
        TextMessage m = (TextMessage)msg;
        StringTokenizer t = new StringTokenizer(m.getText(),",");
        float a1 = (new Float((String)t.nextElement())).floatValue();
        float a2 = (new Float((String)t.nextElement())).floatValue();

        System.out.println(s.add(a1, a2));
    } catch (JMSException e) {
        e.printStackTrace();
    }
}
```

The `javax.jms.Message` received from the queue is cast into a `javax.jms.TextMessage` as the data format being sent is a basic String consisting of the two float values. The `getText()` method retrieves the actual text of the message, and this is parsed to get the two float values separated by the comma.

Once the float values have been read in from the message, the Calculator back-end code can be invoked by calling the `add(float, float)` method on the `CalculatorService` class.

The code example above simply prints out the result of the addition, but for a full asynchronous solution, the result would need to be placed onto a reply queue. The reply queue expected by the consumer will be specified in the original message, and can be determined with the following line of code:

```
Queue replyTo = (Queue) msg.getJMSReplyTo();
```

This `replyTo` queue can now be used by the service, and the result can be put (as a new `TextMessage`) onto this queue. The consumer can then retrieve the message from the reply queue at a later time.

Code to put the result on a reply queue is shown below; insert the code after the `System.out.println()` line in the `onMessage()` method.

Example 5-2 How to put the result onto the reply queue

```
//put the reply onto the reply queue
final InitialContext ctx = new InitialContext();
QueueConnectionFactory qcf =
    (QueueConnectionFactory)ctx.lookup("jms/redbookQCF"); // use the correct QM
name
    QueueConnection qConn = qcf.createQueueConnection();
```

```

        QueueSession session = qConn.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
        Queue queue = (Queue)msg.getJMSReplyTo();
        TextMessage mReply = session.createTextMessage();
        //call the add method of the calculator service
        mReply.setText((new Float(s.add(a1,a2))).toString());
        QueueSender sender = session.createSender(queue);
        //send the result
        sender.send(mReply);
    } catch (NamingException e) {
        e.printStackTrace();
    }
}

```

The reply message, mReply, will sit on the reply message until it is retrieved by the consumer.

The consumer in this case is running in a .NET environment using the MQ Classes for .NET. We shall now cover the implementation of the consumer using these classes.

Make sure you have added the following import statements to the class:

```

import java.util.*;
import javax.jms.*;
import javax.naming.*;
import redbook.coex.sall.business.*;

```

Service consumer (.NET consumer)

Our .NET consumer will be implemented as a “fat client” .NET assembly created in C# in a similar manner to that used in previous scenarios. This assembly will use the MQ Classes for .NET to create a properly formatted request message and put it to a queue. Code will also be provided to obtain the response from the queue.

1. Create a new .NET solution. To do this, open Microsoft Visual Studio.NET and create a New Project. Use the Windows Application template from the New Project dialog so that we are creating a Windows application based on Windows.Forms. You will need to select a name and location; we chose FatClientConsumerAsynch. Once you click **OK**, your Windows.Forms project will be created with a blank dialog.
2. As depicted in Figure 5-8 on page 247, we added a few text boxes to contain input values (called tbArg1 and tbArg2) and the result (called tbResult) and also added a button which we called OurButton, in which we will implement our code.

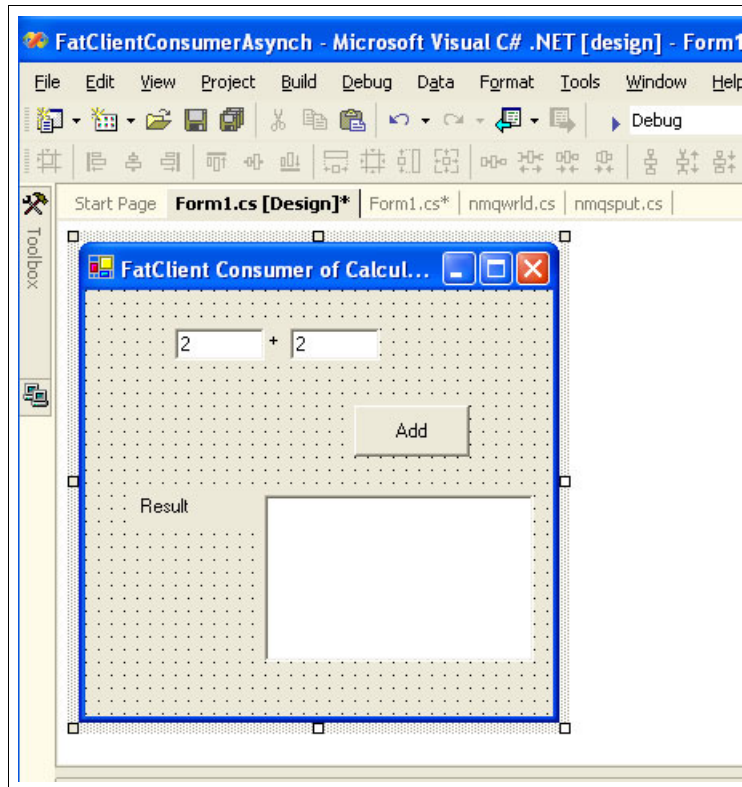


Figure 5-8 The Windows Form for our .NET fat client consumer

3. The first step in implementing the code is to create a reference to the MQ Classes for .NET so we can use them. There are two steps to this process. First, add a project reference within Solution Explorer by right-clicking **References** under the project we just created and selecting **Add Reference....**
4. In the resulting dialog under the .NET tab, choose to browse for a DLL and select the **amqmdnet.dll** under <WebSphere_MQ_Root>\bin.
5. Once the button is placed on the form, double-clicking it will bring up the code that will be executed when the button is pushed. For the sake of keeping our example as simple as possible, we have chosen to implement all our code here.
6. Add a using statement at the top of the C# code, beneath the existing using statements, as shown in Figure 5-9 on page 248.

After executing these steps, your project should look something like Figure 5-9. Note the using statement on the left and the amqmdnet reference in the Solution Explorer.

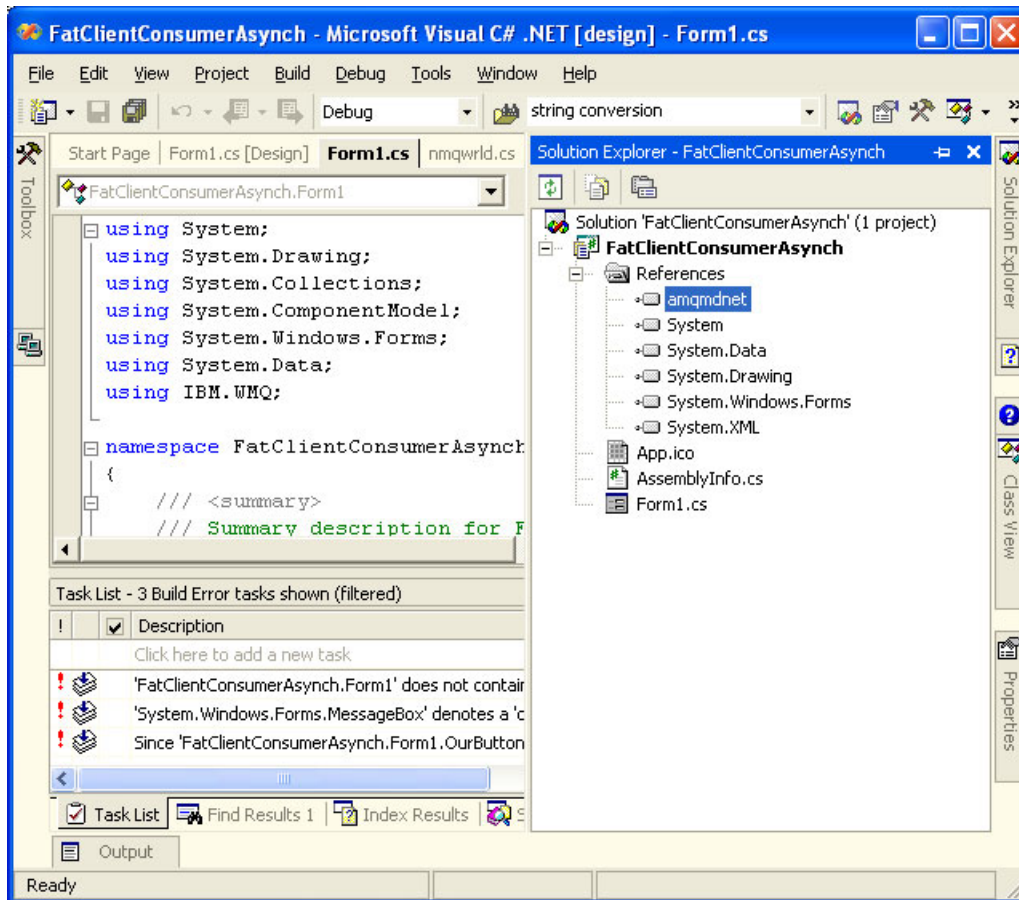


Figure 5-9 References required for the use of MQ Classes for .NET

Putting messages via MQ Classes for .NET

Now we are ready to create code to put the request messages that will drive our calculator service provider. In order to do this, complete the following steps:

1. Create an instance of MQQueueManager. This object abstracts the WebSphere MQ Queue Manager with which we are interacting to obtain access to queues. Creation of this object is slightly different depending on whether you are using the default Queue Manager, or whether the application is running on an MQSeries® Client or an MQSeries Server. See the

documentation for more details. The code to create this object for us was as follows:

```
// MQQueueManager instance
MQQueueManager mqMgr;
// use default queue manager
mqMgr = new MQQueueManager("redbookQM");
```

2. Once a new instance of the MQQueueManager object has been created, we can open the queue of our choice. WebSphere MQ requires you to specify the actions you intend to take upon a queue when opening it and defines the constants used to do so. These constants are exposed via the MQ Classes for .NET through the MQ.MQC public interface. For more information, see the WebSphere MQ documentation and the service pack documentation. The code we used to open our queue, called "inQ" for output (writing messages), is as follows:

```
MQQueue mqQueue;           // MQQueue instance
String queueName;           // Name of queue to use
queueName = "redbookQ";
mqQueue = mqMgr.AccessQueue( queueName, // the name of the queue
    MQC.MQOO_OUTPUT           // open queue for output
    + MQC.MQOO_FAIL_IF_QUIESCING ); // don't if MQM stopping
```

3. Once we have opened the queue, we are ready to create our message. We simply created a string from the text values of tbArg1.text and tbArg2. The MQ Classes for .NET abstract the message into an object as well as the MQ Put Message Options. For more information on the Put Message Options, see the WebSphere MQ documentation. The steps involved here are to create a string message, declare its length, create an instance of the message object, load the string message into that object, create Put Message Options and set them as desired. Once this is complete, we are ready to put our message. The code to prepare our message looks like this:

```
// declare variables and objects required
MQMessage      mqMsg;           // MQMessage instance
MQPutMessageOptions mqPutMsgOpts; // MQPutMessageOptions instance
int            msgLen;          // Message length
String         message;         // Message buffer
// create message
message = tbArg1.Text + "," + tbArg2.Text; // create message
msgLen = message.Length; // set message length
mqMsg = new MQMessage(); // new message instance
mqMsg.WriteString( message ); // load message w/payload
mqMsg.Format = MQC.MQFMT_STRING; // declare format
mqMsg.ReplyToQueueManagerName = "QM1"; // send replies to this QM
mqMsg.ReplyToQueueName = "replyQ"; // send replies to this queue
mqPutMsgOpts = new MQPutMessageOptions(); // declare options
```

4. Once the message is created, you simply call the Put method on our Queue object, passing the Message object and the Put Options object into it.

```
// put message on queue
mqQueue.Put( mqMsg, mqPutMsgOpts );
tbResult.Text = "Message has been sent...";
```

5. Lastly, any of the previous calls could generate an error from WebSphere MQ. The MQ Classes for .NET encapsulate errors and throw them to you as an MQException object. We can utilize a try/catch block to handle these errors. To do this, we wrap all of the above code in a try { ... } and create a catch { ... } block that looks like this:

```
catch (MQException mqe)
{
    // stop if failed
    tbResult.Text = "The following MQ error occurred: " + mqe.Message;
}
```

6. Now build the assembly by clicking **Build -> Build Solution** and you are done with the side of the consumer that puts requests.

The individual bullets above show the lines of code required to implement each logical step in our process of putting a message. Although we have implemented everything under a single method for simplicity, it is obviously possible (not to mention desirable) to separate these functions into places that make more sense for the reuse of objects.

Getting messages via MQ Classes for .NET

We will now cover getting the response messages that will be generated by the requests. The following steps are required:

1. Let's create a new button on our form and name it GetResult. Once that is completed, double-click it to automatically create a GetResult_Click method.
2. Create an instance of MQQueueManager. This is exactly the same as was done under "Putting messages via MQ Classes for .NET" on page 248. Since this object can be shared, a good alternative to doing this under both methods would be to create this object under form construction rather than in each method. In order to do this, simply make the object definition global in scope and create the new instance after the InitializeComponent(); statement in the Form1() constructor.
3. Invoke the AccessQueue() method on our MQQueueManager object to create a new MQQueue object and open the queue for input. Our code looks like this:

```
MQQueue mqQueue;           // MQQueue instance
String queueName;           // name of queue to use
queueName = "replyQ" // name of the queue to open
```

```
mqQueue = mqMgr.AccessQueue( queueName,
    MQC.MQOO_INPUT_AS_Q_DEF // open queue for input
    + MQC.MQOO_FAIL_IF_QUIESCING ); // but not if MQM stopping
```

4. Create a message object to hold the message we are about to get off of the queue. The message object is of type `MQMessage`. We define an object of this type and then create a new instance of it:

```
// create message
MQMessage      mqMsg;           // MQMessage instance
mqMsg = new MQMessage();
```

5. Create a `MQGetMessageOptions` object to define how we are going to get these messages. The `MQGetMessageOptions` object abstracts the MQ Get Message Options. For a full description of the capabilities, see the WebSphere MQ documentation. Here, we use the Get Message Options to set a `WaitInterval` for our Get. This tells WebSphere MQ how long we would like to wait for a new message assuming one is not immediately available.

```
MQGetMessageOptions mqGetMsgOpts; // MQGetMessageOptions instance
mqGetMsgOpts = new MQGetMessageOptions();
mqGetMsgOpts.WaitInterval = 3000; // 3 second limit for waiting
```

6. Now we are ready to get our message. We do so, we invoke the `Get()` method on the queue object we created, passing both the newly created message object and the newly created Get Message Options.

```
mqQueue.Get( mqMsg, mqGetMsgOpts );
```

7. If a string message was returned, it is accessed via the `ReadString()` method on the message object we created. We display it for the user:

```
tbResult.Text = mqMsg.ReadString(mqMsg.MessageLength);
```

8. As before, errors are thrown as an `MQException` and caught via a `try/catch` block. Unlike before, errors can actually be normal. One “normal” error is `No Message Available`, which simply means that no messages were on the queue to be read. So, we surround the above code in a `try { ... }` and implement a `catch { ... }` to evaluate what happened in the event of an error. As before, the MQC:

```
catch (MQException mqe)
{
    // report reason, if any
    if ( mqe.Reason == MQC.MQRC_NO_MSG_AVAILABLE )
    {
        // special report for normal end
        tbResult.Text = "No message to read.";
    }
    else
    {
        // general report for other reasons
```

```

        tbResult.Text = "MQ returned error: " + mqe.Message;
    }
}

```

9. If you test your application at this point, you will see that the returned value shown in the window is wrong; it shows RFH. What happened is that WebSphere put a JMS message in the reply queue and the .NET Classes for MQ do not recognize the JMS message format. The solution is to write your own message formatter using the API and parse the message. Replace and insert the following code after the `// create message` line in step 4 on page 251.

```

// create message
mqMsg = new MQMessage(); // create new message object
mqMsg.Format=MQC.MQFMT_RF_HEADER_2; //
mqMsg.Encoding = 273; //
mqMsg.CharacterSet = 819; //

// great Get Message Options & set them
mqGetMsgOpts = new MQGetMessageOptions();
mqGetMsgOpts.WaitInterval = 3000; // 3 second limit for waiting

// get the message
mqQueue.Get( mqMsg, mqGetMsgOpts );

tbResult.Text+=mqMsg.MessageLength+"\r\n";
String str=mqMsg.ReadString(4); // "RFH "
UInt32 ui32=uint_reverse(mqMsg.ReadUInt32()); // version
ui32=uint_reverse(mqMsg.ReadUInt32()); // length of header
ui32=mqMsg.ReadUInt32(); // encoding
ui32=mqMsg.ReadUInt32(); // coded char set
str=mqMsg.ReadString(8); // format "MQSTR "
ui32=mqMsg.ReadUInt32(); // flags
ui32=mqMsg.ReadUInt32(); // NameValueCCSID
ui32=uint_reverse(mqMsg.ReadUInt32()); // sNameValueCCSID length
str=mqMsg.ReadString((int)ui32);
ui32=uint_reverse(mqMsg.ReadUInt32()); // sJMSfolder length
str=mqMsg.ReadString((int)ui32);
// message content
str=mqMsg.ReadString(mqMsg.MessageLength);
tbResult.Text="Results: "+str+"\r\n";

```

Once you are done with the client, compile it and it is ready to run.

10. The code in the previous step requires an additional function, insert it somewhere at the top of the code.

```
private static uint uint_reverse(uint i) {
    byte[] temp = BitConverter.GetBytes(i);
    Array.Reverse(temp);
    uint returnVal = BitConverter.ToUInt32(temp, 0);
    return(returnVal);
}
```

Sending a JMS message from .NET

If you would like to send a JMS message from the .NET client using the MQ classes for .NET, then you have to generate the message yourself using the MQ classes, just as the sample did when receiving the message. The following sample is a code excerpt for sending a JMS message from .NET.

Example 5-3 Sending JMS message

```
mqMsg = new MQMessage();
// message formatting settings
mqMsg.Format=MQC.MQFMT_RF_HEADER_2; //
mqMsg.Encoding = 273; //
mqMsg.CharacterSet = 819; //
// message header strings
String sNameValueCCSID="<mcd><Msd>jms_text</Msd></mcd> "; // this has to be
nx4 bytes long
String sJMSfolder="<jms><Dst>queue:///default</Dst></jms> "; // this has to be
nx4 bytes long
uint iHeaderLength=(uint)(44+sNameValueCCSID.Length+sJMSfolder.Length);
// assembling the message
UTF8Encoding utf8e=new UTF8Encoding();
mqMsg.WriteBytes(utf8e.GetBytes("RFH ")); // RFH
mqMsg.WriteUInt32(uint_reverse(2)); // version
mqMsg.WriteUInt32(uint_reverse(iHeaderLength)); // length of header
mqMsg.WriteUInt32(uint_reverse(273)); // encoding
mqMsg.WriteUInt32(uint_reverse(1208)); // coded char set ID
mqMsg.WriteBytes(utf8e.GetBytes("MQSTR ")); //format
mqMsg.WriteUInt32(uint_reverse(0)); // flags
mqMsg.WriteUInt32(uint_reverse(1208)); // NameValueCCSID (UTF8)
mqMsg.WriteUInt32(uint_reverse(uint_reverse((uint)sNameValueCCSID.Length)));
mqMsg.WriteBytes(utf8e.GetBytes(sNameValueCCSID));
mqMsg.WriteUInt32(uint_reverse((uint)sJMSfolder.Length));
mqMsg.WriteBytes(utf8e.GetBytes(sJMSfolder));
// your message content
mqMsg.WriteBytes(utf8e.GetBytes("A simple text message from .NET."));
// sending the message
mqPutMsgOpts = new MQPutMessageOptions();
try {
    mqQueue.Put( mqMsg, mqPutMsgOpts );
}
```

```
} catch (MQException mqe) {  
    // report the error  
    System.Console.WriteLine( "MQQueue::Put ended with " + mqe.Message );  
}
```

Configuring the runtime environment

This section describes the step-by-step instructions to run the sample.

Configuring the messaging middleware

1. Create the queue manager in WebSphere MQ: redbookQM.
2. Create two local queues under the queue manager:

```
jms/redbookQ  
jms/replyQ
```

3. Start the queue manager.

Configuring the WebSphere server

In order to run or test the sample, the service provider needs some further configuration.

1. Create a new Queue connection factory for WebSphere MQ provider, with the name redbookQCF, JNDI name jms/redbookQCF, queue manager redbookQM, host localhost (assuming that the queue manager is running on the local machine), and port 1414 (the port number for the redbookQM queue manager).
2. Create a new Queue destination for the message queue, with the name redbookQ, JNDI name jms/redbookQ, and base queue name redbookQ.
3. Create a new Queue destination for the reply queue, with the name replyQ, JNDI name jms/replyQ, and base queue name replyQ.
4. Create a new Listener Port, with the name TestLP, Connection factory JNDI name jms/redbookQCF, and Destination JNDI name jms/redbookQ.

Testing the application

Testing the sample application is quite simple. Run the .NET Windows application you have just developed. Enter a value into the two arguments textbox, then click **Add**. Once you get the message stating the message has been sent, click **Get result**; you should see the results in the textbox.

5.2.4 WebSphere consumer to .NET service provider

This section describes the reverse scenario: that of invoking a .NET service from a WebSphere consumer using WebSphere MQ in asynchronous communication.

We shall first look at the steps involved in creating the service in .NET using the MQ Classes for .NET, and then describe how to call that service.

The code details for getting and putting messages have been covered in the above section. This section does not attempt to repeat much of what was said previously. As such, if WebSphere Consumer to .NET Service Provider interoperability is a scenario of interest to you, it is highly recommended that you read 5.2.3, “.NET consumer to WebSphere service provider” on page 242 in its entirety in addition to this section.

Again, we limit ourselves to the implementation of Calculator we have previously used. Our scenario is a Java consumer, which for our purposes will be a Java program running in a client container, accessing a .NET assembly via asynchronous messaging.

Service provider (.NET service)

Under WebSphere, messaging is implemented on behalf of the service provider in the form of a listener wrapper around the Java code, implementing a Message Driven Bean. As such, the service provider code does not have to implement the code to open a queue and get the request messages to which it will respond, nor consider how the process itself will be started.

Under .NET, however, there is no such paradigm with regard to asynchronous messaging with non-.NET consumers. Therefore, the service provider under .NET must implement the code to connect to the Queue Manager, open queues and read from them.

In addition, decisions about how the service provider will be started must be made. Options include:

- ▶ **Triggering:** WebSphere MQ will automatically start a program when a message arrives on a queue.
- ▶ **Long-Running process:** the program is implemented as a Daemon or Service that is intended to always be running.

Triggering

Triggering is an enormous subject in and of itself and is by far the most common mechanism for starting a service provider. Many options exist for how it can be implemented and each option has some effect on the processing logic. However, there is a common structure we can use for a triggered application. In order to use our calculator class in a triggered mode, we would implement a triggered application that reused the existing .NET class locally.

The following outline shows the basic processing logic of our .NET service provider as a triggered application.

1. Connect to the Queue Manager.
2. Open the input queue.
3. Loop until no message is available or some other error occurs.
 - a. Read a message with a short time-out (wait a few seconds if there is no message there).
 - b. Parse the received message.
 - c. Use the Calculator class to add numbers together.
 - d. Build the response message.
 - e. Put the response message.
 - f. Loop until no message is available or some other error occurs.
4. Close the queue.
5. Close the connection to the Queue Manager.
6. End the execution.

Our service provider would be implemented as a .NET assembly and built as an executable. WebSphere MQ would be administered to start (trigger) this executable whenever a message arrives on the input queue. The executable would be started in its own process space, run until the queue is empty and then end.

More information on triggering is available within the WebSphere MQ product documentation.

Long-running service providers

Long-running processes are usually implemented in situations where triggering is impractical. Examples of this include very high volume processing, where the time and resource overhead required to start up a new process is unacceptable.

A long-running service provider is essentially a listener that you write yourself. This consists of creating a program that is intended to remain running and is always (when it is not doing work) reading from the queue with an infinite time-out. In this situation, the service provider's process space is entirely up and initialized. When a message comes in, it is immediately handed to the thread reading on the given input queue and is processed with an absolute minimum of delay.

Under .NET, a program that runs all the time is usually implemented as a *Windows Service*. A Windows Service runs under the control of the Service Control Manager and can be stopped and started administratively. Services are described in more detail in "Windows Services" on page 67.

Long-running service providers must be able to respond to administrative events. Therefore, they are often more complex from a logic perspective. If we chose to implement our calculator as a long-running service, we would create a Windows Service within Visual Studio.NET that implements the executable under which our Calculator service will be provided and handles administrative activity such as startup and shut-down.

A typical pattern for our logic in this case would be:

- ▶ Start up:
 - a. Connect to the Queue Manager.
 - b. Spawn one or more threads to perform work.
 - c. On notification of failure, spawn new worker threads if possible.
 - d. On administrative shutdown:
 - i. Notify all threads to stop.
 - ii. Close the connection to the Queue Manager.
 - iii. End the execution.
- ▶ Each thread:
 - a. Open the input queue.
 - b. Loop until error or notification to shut down.
 - i. Get a message with an long time-out (possibly minutes).
 - ii. Parse the input message, use the local .NET Calculator class to add the two numbers together.
 - iii. Create a response message and put it to the reply queue.
 - iv. Look for notification to shut down from main thread.
 - c. On error or notification to shut down, close the queue and notify the main thread.

The administrator of the server would have the capability of starting and stopping our service via the Service Control Manager. As long as the service is running, requests would be satisfied immediately by any given thread.

Service consumer (WebSphere consumer)

A WebSphere consumer wants to asynchronously call the Calculator service running in the .NET environment. The .NET Calculator application has now been given the ability to listen on a WebSphere MQ queue for a request message, and will then execute the back-end code and put the result onto a reply queue.

The message format expected by the .NET service provider is a simple string consisting of two float values separated by a comma (for example: 2.45, 45.3).

So, using WebSphere Studio Application Developer, we need to create an application which will create a simple TextMessage, access the WebSphere MQ queue, and put the message on the queue.

As far as the service consumer is concerned, the queue it writes the message to is local to the client. The message will be automatically put onto the remote queue by WebSphere MQ.

In order to create your consumer, create a J2EE Application Client in WebSphere Studio. Within this project, create a new Java class. The following code will create a javax.jms.TextMessage, define a reply to queue, and put the message onto the appropriate queue to call the .NET service.

The numbers being added are hard-coded in this application for simplicity, since the purpose of the application is simply to show asynchronous interoperability.

Example 5-4 Code to put a message on a queue

```
public class Consumer {
    public static void main(String[] args) {
        try {
            final InitialContext ctx = new InitialContext();
            QueueConnectionFactory qcf =
                (QueueConnectionFactory)ctx.lookup("jms/QCF");
            QueueConnection qConn = qcf.createQueueConnection();
            QueueSession session = qConn.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
            //find the queue
            Queue queue = (Queue)ctx.lookup("jms/redbookQ");
            //create the message
            TextMessage m = session.createTextMessage();
            //set the values for the addition in the message
            m.setText("3.0,5.7");
            Queue reply = (Queue)ctx.lookup("jms/replyQ");
            m.setJMSReplyTo(reply);
            //send the message
            QueueSender sender = session.createSender(queue);
            sender.send(m);
            System.out.println("Request Message Sent");
        } catch (JMSException e) {
            e.printStackTrace();
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
}
```

The .NET service will then detect this message on the queue, perform the calculations, and put the result onto the reply queue `jms/replyQ` as defined in the request message.

When the response is returned, the consumer needs to obtain it from the queue. The above code does not include obtaining the response simply to underscore the fact that the response does not need to be obtained right away nor even, necessarily, by the same consumer, assuming that all information necessary to make sense of the response is contained within the message.

Example 5-5 Code to obtain the response message from Calculator

```
...
//code to receive the reply message from the queue
//need to start the queue connection to receive messages
qConn.start();
QueueReceiver receiver = session.createReceiver(reply);
//get the message once it has been placed there by the
```

```
//service provider. the receive(0) command suspends this
//code until a message is received
TextMessage receivedMessage = (TextMessage) receiver.receive(0) ;
System.out.println("Result = " + receivedMessage.getText());
...
```

In order to add the code necessary to receive the message to our client as defined above, simply insert the code in Figure 5-5 on page 259 after the following line:

```
System.out.println("Request Message Sent");
```



Scenario: Synchronous stateful

This chapter provides examples for the synchronous interaction scenarios defined in 4.2.1, “Stateful synchronous interaction” on page 112 from the perspective of a .NET application artifact consuming the services of a WebSphere application artifact.

In this chapter, we will focus on scenario f: Coexistence via business tier logic to business tier logic integration, and scenario c1: Coexistence via client tier logic to business tier logic. Although the examples in this chapter focus on interactions where .NET is the consumer of WebSphere stateful service providers, we also discuss interactions the other way around and solutions for them.

6.1 Problem definition

This section provides a detailed description of a sample problem we are trying to solve within this scenario.

You are reading this book because you are interested in interaction between .NET and J2EE. Interactions can typically be viewed through a consumer/provider relationship. While .NET makes it easy to develop a sophisticated and rich user experience (presentation), it is seldom the case that all enterprise systems run on Windows; remember also that .NET is about exploiting Windows while J2EE is about exploiting language consistency across platforms.

You will recall that we broke interaction types down as follows:

- ▶ Stateless synchronous
- ▶ Stateless asynchronous
- ▶ Stateful synchronous
- ▶ Stateful asynchronous

These categorizations are useful because they describe widely differing interaction scenarios that solutions tend to follow.

Where integration is needed, it can often be accomplished via simple asynchronous mechanisms and techniques, or via stateless mechanisms and interactions such as Web Services. However, there are occasions where maintaining state across invocations is required. Here we define those scenarios, provide an example, then define and pursue solution options.

6.1.1 Description of the problem

In our scenario, we have a stateful service provider implemented in Java. A Business layer .NET component wishes to consume the services of this service provider, so we must provide a mechanism that allows stateful interaction between two Business layer components where a .NET artifact is the consumer and a WebSphere artifact is the provider.

The snippet in Example 6-1 on page 263 shows the code for our service provider's basic interface. Our service provider provides the interface `ICalculator1` which exposes one property and some functionality via three methods. The method `setCurrentTotal` allows the consumer to set the starting value at any time to whatever it wants. The corresponding `getCurrentTotal` method allows the consumer to obtain the current total of the stateful adding machine at any time. The `add` method allows the consumer to pass a new value

as an argument to be added to the current total. The property containing the current total is then updated and returned to the consumer as a return value as well, for the sake of simplicity.

Example 6-1 Java implementation of our stateful calculator

```
public class Calculator implements ICalculator1, ICalculator2, ICalculator3 {
    //args for ICalculator1 setArg methods
    private float calc1CurrentTotal;
    //setCurrentTotal method for ICalculator1
    public void setCurrentTotal(float arg) {
        calc1CurrentTotal = arg;
    }
    //getCurrentTotal method for ICalculator1
    public float getCurrentTotal() {
        return calc1CurrentTotal;
    }
    //add method for ICalculator1, using the pre-set
    //argument calc1CurrentTotal
    public float add( float arg1 ) {
        calcCurrentTotal += arg1
        return calc1CurrentTotal;
    }
    ...
}
```

In order to consume this service, the consumer must first call the set methods for each of these arguments and then call the method in order to obtain the sum. The service provider maintains state between these three invocations. The pseudo code snippet below shows proper use of this stateful service provider.

Example 6-2 Pseudo-code consumer implementation for ICalculator1

```
public class MyConsumer {
    ...
    // create a calculator
    Calculator objMyCalc = new Calculator;
    // set the current total to the value from which we wish to start adding.
    objMyCalc.setCurrentTotal( 2 );
    // add some values to that total
    Answer = objMyCalc.Add( 2 ); // 2 + 2 is 4
    Answer = objMyCalc.Add( 2 ); // 4 + 2 is 6
    Answer = objMyCalc.Add( 2 ); // 6 + 2 is 8
    // local variable Answer is now set to 8.
    // but we can also get the current total from property
    Answer = objMyCalc.getCurrentTotal( );
    ...
}
```

Our consumer runs in the .NET environment and our service provider runs in J2EE to create an interaction which looks something like this:

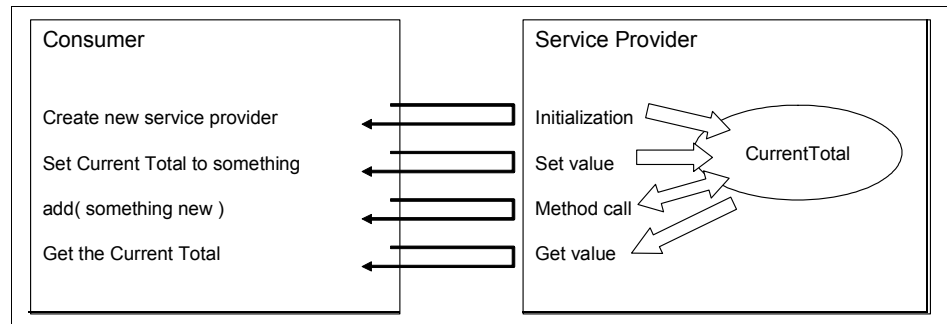


Figure 6-1 Multiple invocations operating on single state object CurrentTotal

From Figure 6-1, it is clear that several invocations are used to perform operations upon and to view a state object. The state of this object must be consistent between invocations. In addition, the order of the invocations is significant. Getting the value of CurrentTotal before all the additions have been performed would not be very useful.

Since the consumer is doing nothing but waiting (blocking) while allowing the service provider to perform any given invocation, this interaction style is synchronous.

Since the service provider is responsible for storing this state information, and since it is not persisting this information anywhere but in a memory area, it is clear that the execution of the service provider must surround all operations. That is, a single instance of the service provider must satisfy all of the requests for a particular consumer for the operations and state to be meaningful.

Stateful, synchronous interaction is quite common; we use it for nearly everything, in fact. In both .NET and WebSphere or J2EE environments, this is most often accomplished by creating an instance of the service provider object within the consumer's process space and performing invocations of methods on that object. When the consumer is finished, the object is destroyed. This is depicted in Figure 6-2 on page 265 and denoted as (A).

In both the .NET and WebSphere environments, it is also possible and common for the service provider to live in another process, as depicted in Figure 6-2 on page 265 (option (B)), or on another physical node from the consumer (as depicted in option (C)). Both .NET and WebSphere have containers within which service providers may run as server-side providers, allowing them to service the requests of many consumers.

Assuming that both our consumer and service provider were written in Java, option A is analogous to the consumer and provider running inside a single Java Virtual Machine (JVM) process. Options B and C are analogous to running our service provider in an Enterprise Java Bean (EJB) container and invoking it via Remote Method Invocation (RMI).

In the .NET world, the analogue for option A would be a .NET consumer application running within the Common Language Runtime creating an instance of the .NET class service provider locally within the same process space. The analogue for option B would be accessing a .NET server class and, for option C, using .NET Remoting to facilitate the stateful interaction across nodes in an RPC fashion.

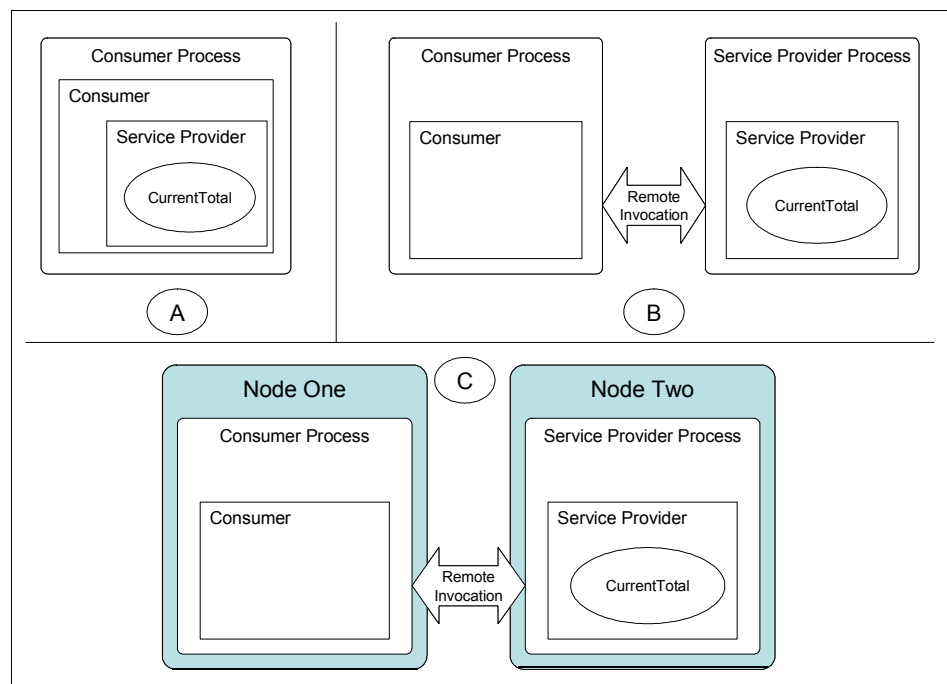


Figure 6-2 Process options for synchronous, stateful interaction

In our basic scenario, we will deal first with option A, a .NET consumer, which in our case will be a simple C# Windows Forms application consuming a simple Java class in process. Our extended scenario will then separate the service provider and consumer onto physically separate nodes and perform the same interaction.

We will discuss more than one way of accomplishing this and make recommendations about how to choose between these methods. Our examples will focus on one method of .NET consumption of WebSphere service providers.

6.1.2 Considerations

Before we can complete the picture of integration between these two environments, we need to understand some basic components that make up the solutions to the stateful, synchronous invocation problem within each of these two environments.

Runtimes and native interfaces

.NET applications run within the Common Language Runtime (CLR). Java applications run within the Java Virtual Machine (JVM). In order to run a Java program within the process space of a .NET application, you must load both runtimes into the same process space.

This might represent quite a bit of overhead. However, there are other options; to understand them, let's first understand the tools provided by each environment for calling into and out of their respective runtime environments on a local basis.

.NET Interop

.NET applications run within the Common Language Runtime (CLR) and obtain services from that runtime, such as language independence and garbage collection. Code which is written to run under the Common Language Runtime is known as *managed code*. The .NET languages C# and Visual Basic.NET are intended to produce managed code which is not binary code executed natively within the operating system, but instead is bytecode compiled at the last minute within the runtime and executed within it in a managed way.

In order for a .NET assembly to call anything that is not managed by the CLR, for instance a C-style function within a Dynamic Link Library (DLL), it is necessary to leave the managed code environment during the call into the DLL. Microsoft .NET provides a mechanism to do this, called Interop.

Interop allows you to call into native, unmanaged code from .NET managed code, and vice-versa. Interop is analogous to the Java Native Interface.

HiJava Native Interface

Java applications run within the Java Virtual Machine (JVM) and obtain services from that runtime, such as language independence and garbage collection.

In order for a Java program to call anything that is not managed by the JVM, for instance a C-style function within a Dynamic Link Library (DLL) on Windows, it is

necessary to leave the managed code environment during the call into the DLL. The J2SE SDK provides a framework for doing this called the *Java Native Interface* (JNI) framework.

The Java Native Interface allows you to call unmanaged, native code from within the Java Virtual Machine and vice-versa. The Java Native Interface is analogous to Interop.

As you can see, there are many similarities between J2EE and .NET.

Stateful remote invocation

The technologies for getting into and out of the respective runtime environments of .NET and J2EE support local invocation. We are also concerned about providing this interaction between processes and physical nodes.

Both J2EE and .NET provide mechanisms for remote consumption of services in a stateful manner. While these mechanisms are intended to provide remote invocation services within their environment (Java to Java and .NET to .NET), the remoting technologies can be combined with the native interface technologies on either side to create the possibility of inter-process and inter-node stateful.

Remote Method Invocation (RMI)

The Java2 SDK provides a facility for implementing stateful synchronous calls to a remote Java object. This is known as *Remote Method Invocation* (RMI). When you implement RMI within your application artifacts, you may choose whether you will use the Java Remote Method Protocol (JRMP) or the Internet Inter-ORB Protocol (IIOP) as the underlying transport.

More information on Remote Method Invocation can be found at the following locations:

<http://java.sun.com/products/jdk/rmi/>
<http://java.sun.com/marketing/collateral/javarmi.html>

More information on the Inter-ORB Protocol (IIOP) may be found at:

<http://www.omg.org>

.NET Remoting

.NET provides a facility for implementing stateful synchronous calls to a remote .NET object. This is known as *Remoting*. In Java, Remote Method Invocation (RMI) separates implementation from the underlying protocol, and .NET Remoting does the same thing. In the case of Remoting, two channels are provided for the developer: `HttpChannel` and `TcpChannel`.

- ▶ HttpChannel uses SOAP formatted messages and transmits information using the HTTP protocol.
- ▶ TcpChannel uses a binary formatter and transmits information using TCP.

It is also possible to create your own channel facility using classes provided within .NET. More information on .NET Remoting may be found at the following location:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetremotingoverview.asp>

Data model

Data representation types differ between .NET and Java. When a consumer requires the services of a service provider that uses incompatible data types, it is necessary to map between them in a consistent way. The same is true of complex and user-defined data types. One of the issues in achieving interoperability between different programming languages is how to provide a language-independent abstraction for common language data types.

Solution options

Our scenario requires that we consume a Java service provider object from a .NET object. The underlying technologies and issues we have just discussed allow us to consider three possible fundamental solutions to our problem.

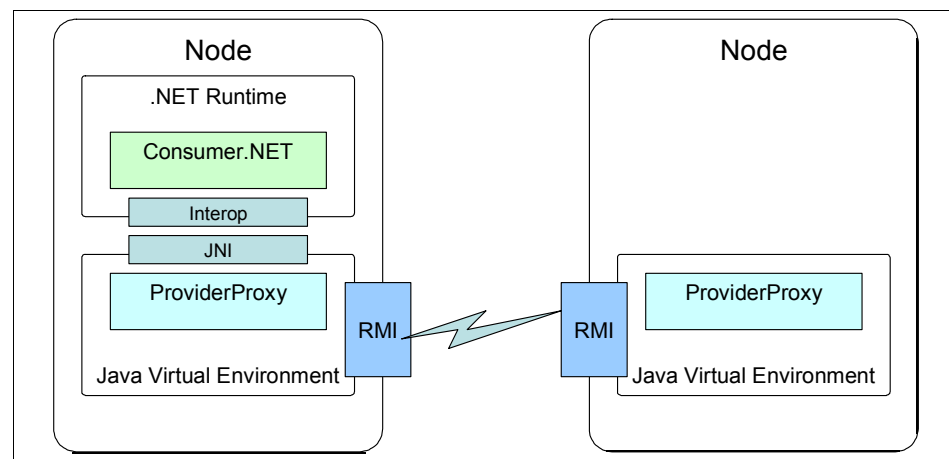


Figure 6-3 Consumer-side integration with Java RMI as the transport

The first option, represented in Figure 6-3, shows consumer-side integration of the environments using Java Remote Method Invocation (RMI) as the transport to cross the network layer. This option has several advantages, including the following:

- Use of the Java Naming and Directory Interface to find remote classes.
- Isolation of .NET code and the Windows OS to nearer the Presentation layer.
- Particularly advantageous if your Service Provider does not run on Windows.
- Useful when CORBA-compliant systems also wish to use the services of this service provider via IIOP.

A second option, represented in Figure 6-4, shows provider-side integration with .NET Remoting as the mechanism, permitting inter-node synchronous, stateful interaction between objects. Provider-side integration is useful when the technology used by the service provider is a technology you wish to contain within the environment.

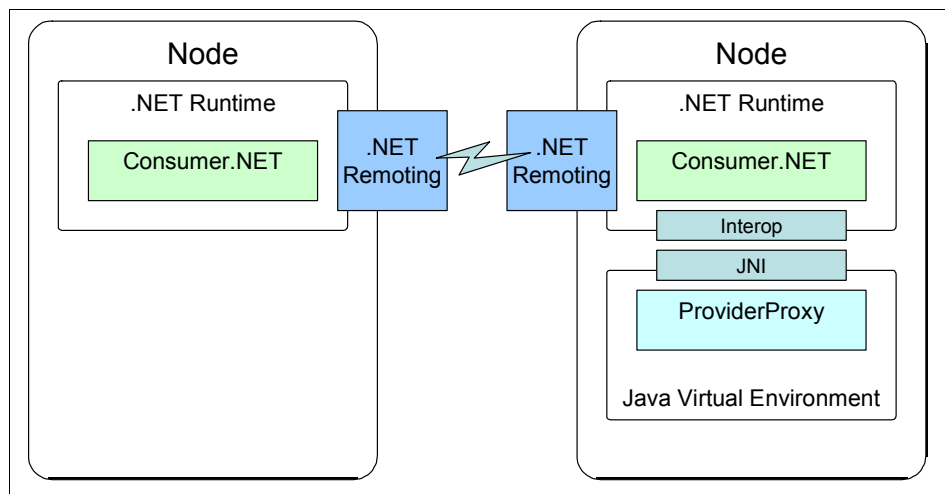


Figure 6-4 Provider-side integration with .NET Remoting as the transport

The third scenario that presents itself from consideration of these underlying technologies is a little bit more complicated and deals with the underlying protocols and access available to .NET Remoting and Remote Method Invocation under Java. This option, depicted in Figure 6-5 on page 270, utilizes the design patterns implemented by the remote invocation technologies for J2EE and .NET. The fact that the protocol used is abstracted from the remote invocation APIs allows us to find and implement a common transport protocol beneath them.

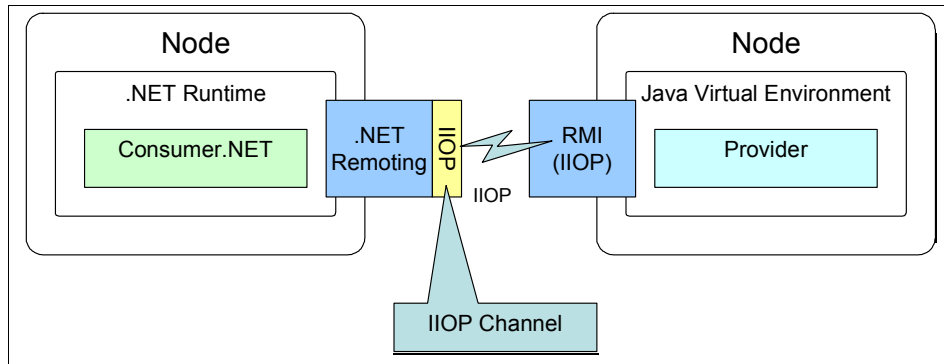


Figure 6-5 Transport Integration via .NET Remoting over IIOP

Under .NET, this is done by implementing a remoting *channel*. A remoting channel is an object that is implemented under the .NET Remoting framework in order to transport messages between remoting targets. As we mentioned previously, Java's Remote Method Invocation includes the implementation of an IIOP transport. Although .NET does not implement such a transport, it does expose the transport hooks for the user to implement.

This is a particularly useful solution for a number of reasons. IIOP not only supports the stateful interaction we require, but it also opens .NET up to interaction with CORBA-based applications. Perhaps as importantly, the fact that the transport is used as the integration point and extends into both runtime environments (the .NET Common Language Runtime and the Java Virtual Environment) means that you have native .NET talking directly to native Java without the need for the overhead associated with Interop and JNI.

In the context of .NET to WebSphere integration, this solution provides some advantages and disadvantages over the process integrations.

Some advantages:

- ▶ Native Runtime Use: avoid the use of (and overhead associated with) Interop and JNI.
- ▶ Infrastructure flexibility: J2EE and .NET application artifacts may exist wherever they are within the infrastructure and it is not necessary to place both frameworks on the same system for integration purposes.

Some disadvantages:

- ▶ No lookup of remote resources via the lookup facilities associated with either platform (unless implemented as part of the solution).
- ▶ Software to write or buy.

Packaged solutions

There are packaged solutions on the market which utilize these underlying technologies to create a synchronous, stateful invocation framework between these environments. Here, we briefly review available options:

- ▶ IBM ActiveX Bridge
- ▶ IBM Interface Tool for Java, also known as Bridge2Java
- ▶ Third-party .NET Remoting to RMI bridging products

IBM ActiveX Bridge

The ActiveX Bridge is a technology developed by IBM for the purpose of providing stateful, synchronous interaction between Microsoft ActiveX component consumers and Java service providers. The ActiveX bridge uses the Java Native Interface framework to expose Java classes and methods to ActiveX applications. It also provides data type mapping facilities between ActiveX and Java.

The ActiveX Bridge uses Reflection to discover and access the methods and properties available within a Java class and then exposes these to ActiveX clients at runtime as IDispatch interfaces.

The ActiveX Bridge supports free and apartment model threading for both client code and ASP. The ActiveX Client may be an Active Server Page running under Internet Information Services (IIS), VBScript or an ActiveX object written in Visual Basic or C++.

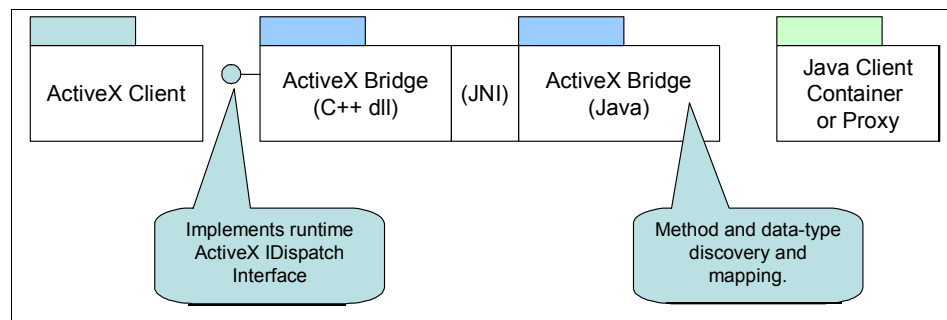


Figure 6-6 IBM ActiveX Bridge

The ActiveX Bridge supports accessing of Enterprise Java Beans via a Client Container object or a Proxy. A Client Container object is simply a Java object created within a local client container that manages the access to an Enterprise Java Bean.

The ActiveX Bridge provides a simple interface with which the user may initialize a Java Virtual Machine, find classes, invoke methods and handle ActiveX to Java data type conversions. A brief overview of these methods is useful for subsequent discussion.

- ▶ The `XJB.JClassFactory` object is the ActiveX Bridge. Your application creates an instance of this object in order to begin to use the Java environment and access Java classes. This object contains the following methods:
- ▶ `XJBInit()` - After creating an instance of `XJB.JClassFactory`, the application calls this method to create an instance of the Java Virtual Machine and initialize it. Parameters passed into this method provide the same control over the JVM initialization as you would have when using **Java** from the command line. After this has successfully been invoked, the JVM is initialized and ready for use.
- ▶ `FindClass()` - After initializing the JVM, an application uses this method to find a Java Class within the classpath. This method returns a `JClassProxy` object which can be used immediately to access static methods and fields of the Java object as well as all the `java.lang.Class` methods.
- ▶ `NewInstance()` - If you need to create an instance of a Java object and access non-static behavior, pass the `JClassProxy` object into this method and it will create a new instance of the Java object, returning a `JObjectProxy` object. This object can be used to access all features of the Java object.
- ▶ `GetArgsContainer()` - Java classes often require constructor arguments. This method creates a container for those arguments that can be used to provide constructure arguments by passing this object to the `NewInstance()` method, or when calling a method on a Java class.

In addition, there are several primitive data-type conversion helpers.

For more information about developing using the the ActiveX Bridge and a reference for the methods above, see the WebSphere documentation at:

<http://publib.boulder.ibm.com/infocenter/wasinfo/index.jsp>

At this site, the reference documentation for developing ActiveX Bridge programs can be found under **WebSphere Application Server -> All Topics By Activity -> Developing -> Applications -> Client Modules -> Developing ActiveX Application Client Code**.

IBM Interface Tool for Java

IBM has a technology call Interface Tool for Java which was originally conceived to deliver stateful integration between Java clients and COM components. Interface Tool for Java can also be used to provide stateful integration between Java clients and .NET assemblies.

Before we describe how Interface Tool for Java can provide Java clients with stateful integration to .NET assemblies, let's briefly consider how Interface Tool for Java provides integration to COM Components (this is important for an understanding of the Java to .NET solution). Most COM Components implement an interface call IDispatch (this is often referred to as an automation interface). Interface Tool for Java works with this IDispatch interface in two ways:

- ▶ The Interface Tool for Java build-time generates Java proxies for COM IDispatch interfaces.
- ▶ The Interface Tool for Java runtime provides the runtime binding, marshalling, and type mapping, and maintains Java to COM object relationships; it also performs object life cycle management between each Java proxy instance and its associated COM class (CoClass) instance. It does this using the IDispatch interface for the given COM component.

These Interface Tool for Java generated Java proxies to COM CoClasses enable any Java-based technology (EJB, Java Bean, Servlet, JSP, etc.) to statefully integrate with the proxied COM components.

This is illustrated in Figure 6-7.

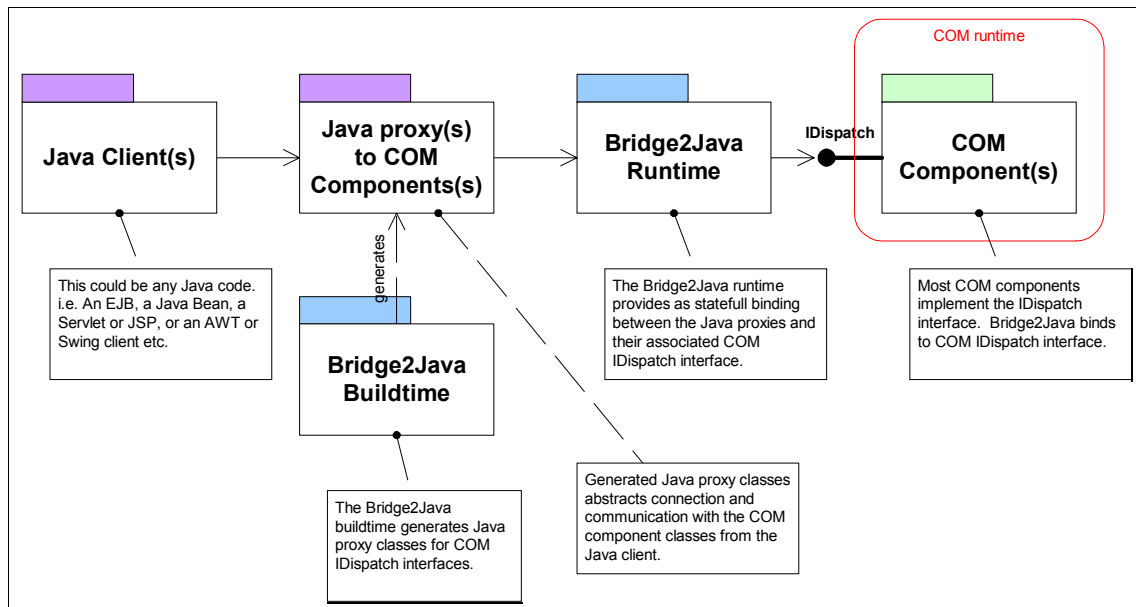


Figure 6-7 the Interface Tool for Java solution model for COM integration

Third-party .NET Remoting to RMI bridging products

It is beyond the scope of this book to cover third-party solutions for .NET Remoting/RMI bridging. However, several products have implemented such

solutions. There is also an open-source .NET Remoting to RMI Bridge being developed under Source Forge, called IIOP.NET, which can be explored at:

<http://sourceforge.net/projects/iiop-net/>

The advantages of such a solution are significant. Although it is not covered in detail within this book, it is something we recommend you look into for such requirements.

6.1.3 Constraints

Consider the following constraints in your solution.

ActiveX Bridge constraints within .NET

Although the ActiveX Bridge is not a .NET implementation, it is an implementation built upon the Component Object Model (COM). As such, it doesn't get you all the way into (or out of) .NET.

However, as we covered in the introduction, .NET is built upon underlying technologies such as COM and provides a facility (Interop) to utilize ActiveX from within the .NET runtime. This provides us with options for implementing ActiveX Bridge from within the .NET environment.

The first option to consider is to use the .NET Interop capability to get directly into the ActiveX Bridge. Since the ActiveX Bridge is implemented as an ActiveX object, it seems as though this should work. Initial testing was performed that revealed that it is indeed possible to access static methods and fields using ActiveX Bridge directly from VB.NET via Interop. However, C# behaved quite differently when making these same invocations and neither environment allowed us to access non-static fields and methods successfully directly from .NET. We therefore decided not to pursue the direct approach.

The second option is to write a native ActiveX DLL that isolates all calls into the ActiveX Bridge. We then use Interop to get to these DLLs from .NET. Although this seems to create one more layer between the applications, our testing found that it is much more reliable and predictable. There are also other limitations (see "JNI limitations" on page 275) which prompted us to consider separating the code that uses Java from the rest of our application processes. Since this book is about solutions that do not rely upon products other than WebSphere and .NET, and in the interest of establishing best practices for use of ActiveX Bridge from .NET, this is the approach we have chosen.

Interface Tool for Java constraints with regard to .NET

As with ActiveX Bridge, Interface Tool for Java was not originally intended to get Java programs into COM objects via IDispatch interfaces.

.NET assemblies do not normally implement IDispatch interfaces, but their interfaces *can* be represented by IDispatch interfaces. Fortunately, the .NET SDK provides a utility called **tlbexp.exe** which, with the correct directives, can be used to generate type information which can include an IDL definition of IDispatch interfaces. When this type of information is correctly registered in the Windows registry, the .NET runtime will automatically provide a façade to the .NET assembly for IDispatch clients (in this case, the Interface Tool for Java runtime is the IDispatch client). This is illustrated in Figure 6-8.

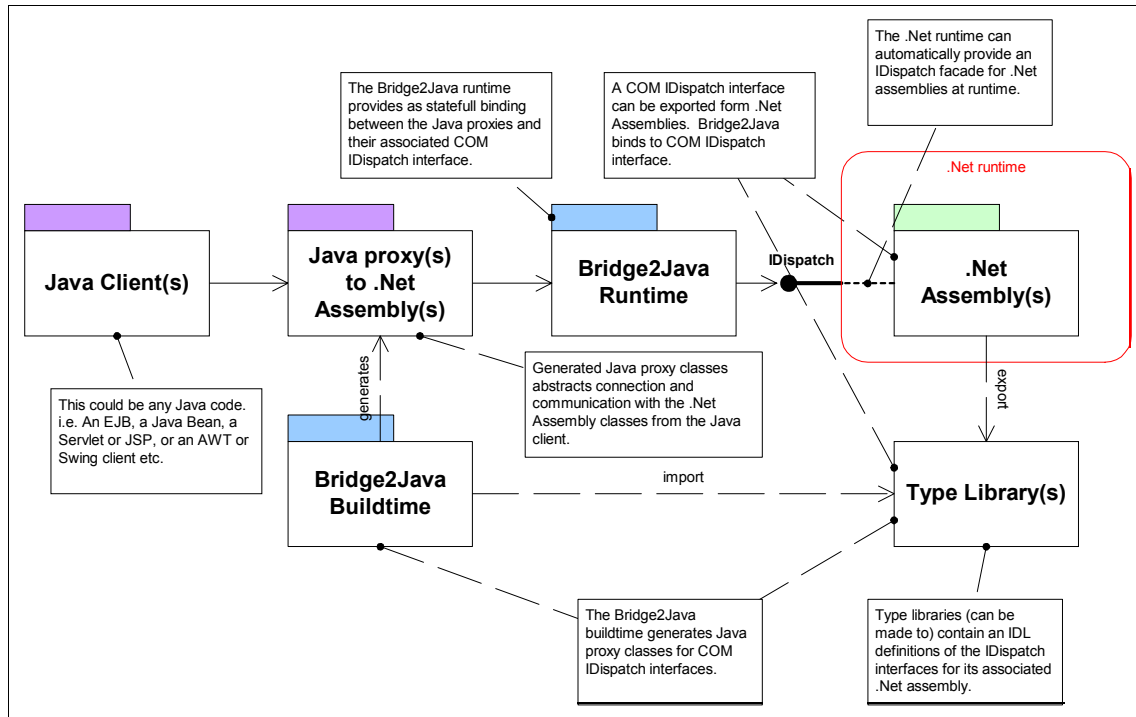


Figure 6-8 the Interface Tool for Java solution model for .NET integration

JNI limitations

A limitation in the Java Native Interface (JNI) prohibits us from loading a Java Virtual Machine more than once in a given process. This means that once you have loaded and initialized the JVM, you cannot unload it or reinitialize it without taking the process in which it is running completely down and back up (stopping and restarting your program). Since ActiveX Bridge uses JNI under the covers, this limitation affects our solution.

This limitation is significant because, for instance, IIS manages the processes under which Active Server Pages run. Once a JVM is loaded for that process, it

cannot be reloaded or reconfigured without bouncing the IIS process under which it is running.

While this limitation is less severe outside of IIS, for instance in a VB application, it is still somewhat inconvenient to ask the user to close and restart the application because we need to reconfigure the JVM. We will make recommendations to address this issue appropriately below.

6.1.4 Recommendations

This section provides recommendations associated with selecting and implementing solutions in this problem space.

Selecting a solution

The selection of a solution from the available options involves consideration of many parameters and the environment. There is no technological silver bullet implementation that will take away the complexity of a stateful, synchronous invocation across platforms such as these. However, there are clear advantages to each approach identified above. Although there is no way to completely cover this topic here, we suggest that, when selecting an approach, you consider the following key issues at a minimum:

- ▶ **Keep it simple.** Complexity is the enemy of success. The more complex solution will always be more expensive and more troublesome than the simpler one.
- ▶ **Isolate environments.** What is your company direction? Does your future hold more of one environment than another or is it unclear? Do your management processes and practices favor one environment or the other? Our options allow you to choose the transport that flows over your network, and therefore choose which environment you will administer remote invocations with. If administrative management processes are mature for one environment and not for the other, it may be wise to consider using that environment at the transport level and relegate the other to discrete islands.
- ▶ **Security.** How are your users authenticated and what is your trust model for computing? This is a complex subject, but each environment has its own security mechanisms and capabilities. Considering how they work can play heavily into the decision of which remote invocation method to use.
- ▶ **Bridging the Virtual Machines versus the Transports.** As discussed above, an alternative to bridging the runtime environments of Java and .NET via JNI and Interop is to bridge the stateful remote invocation mechanisms these environments utilize internally. There are already several third-party products which do just that and it is also possible to implement your own IIOP Channel for .NET Remoting, which effectively implements this bridging mechanism to any CORBA-compliant mechanism such as RMI. We suggest

you consider this option if you have CORBA in your environment since it provides the most flexibility in infrastructure management and placement of artifacts.

ActiveX Bridge best practices under .NET

We have chosen to implement the ActiveX Bridge as our solution. The following recommendations are made with regard to use of ActiveX Bridge under .NET.

Isolate ActiveX Bridge calls into a server-side COM DLL that will execute in its own process space under the control of the Service Control Manager. This will require that you implement the calls to ActiveX Bridge in C++. You will then use Interop from .NET languages to access the DLL.

Although this may require some small extra code, this solution offers several valuable advantages and addresses many of the constraints we have identified. Advantages include the following:

- ▶ Access to Java classes can be isolated into discrete running processes which can be stopped and started at will with a minimum of interruption.
- ▶ Access to Java service providers can be grouped. The designer can decide whether to group by function, performance parameters, resource requirements or any other sensible method.
- ▶ Data type issues between .NET and Java can be handled in one place so that native data types are presented to consumers.
- ▶ Since ActiveX Bridge is thread-capable, multiple clients can be served simultaneously by a given Java Class.
- ▶ Services can be controlled by administrators.
- ▶ Parameters required for the JVM can be externalized into Constructor Strings or via other methods to allow easy fail-over or configuration of the environment by administrators.
- ▶ Security can be implemented on the access to the initial COM objects if desired.

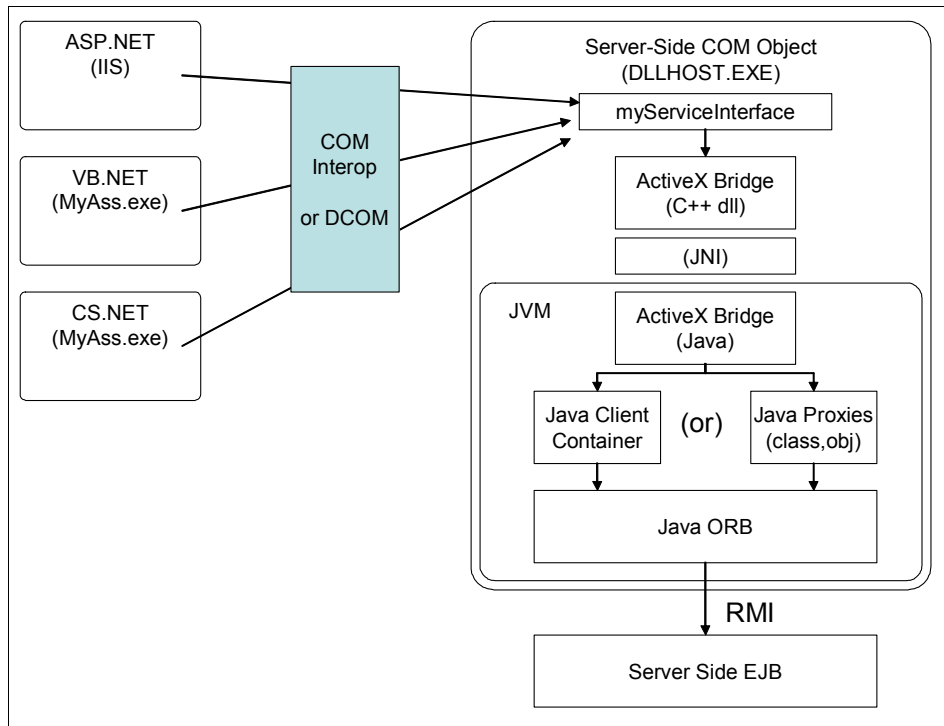


Figure 6-9 Recommended solution with ActiveX Bridge for .NET

Figure 6-9 depicts the recommended implementation of ActiveX Bridge for a .NET environment. A server-side COM object is created that is designed to run under the control of the Service Control Manager. This object has the following responsibilities:

- ▶ Contain all calls to ActiveX Bridge.
- ▶ Initialize the JVM on construction.
- ▶ Expose a thin wrapping interface representing the Java objects you wish to make available to the .NET world. When these methods and properties are used by client applications, the Java methods and fields are manipulated via ActiveX Bridge.
- ▶ Use the ActiveX Bridge functions to handle data representation issues as they come up.

In the interest of completeness, the figure depicts access all the way to an Enterprise Java Bean, which is exactly what the ActiveX Bridge was designed to do. The Java Client Container and Java Proxies depicted merely represent the

options for accessing an EJB. Either or both may be implemented. The location of the EJB in the environment is irrelevant.

6.2 Solution model using the ActiveX Bridge

Our solution model follows the pattern we have established, beginning with a simple scenario to illustrate the processes and then describing the extended scenario.

6.2.1 A solution to the problem

We begin with a stateful calculator written in Java which we have decided to reuse in .NET. The creation of this calculator was covered earlier.

We will implement the ActiveX Bridge inside a separate COM component DLL, written in C++, to handle all calls into the ActiveX Bridge.

In our simple scenario, the client will be a C# Windows Forms fat client, which will reference our COM DLL, instantiate and utilize its methods. The DLL will access the Calculator.jar file that implements our stateful calculator. To simplify the discussion, no cross-process or cross-machine code will be implemented in our simple scenario.

The Calculator.jar code will be entirely reused without change.

6.2.2 Simple scenario details

It is usually helpful to start at the service provider and work our way back to the client when describing an implementation. We will use this convention here. This section will describe in detail what we did to make the Calculator.jar implementation accessible from .NET.

Environment

Construction and execution of the solution discussed here requires the following components on the workstation:

- ▶ IBM WebSphere Studio Application Developer (WebSphere Studio) V5.1. with ActiveX Bridge
- ▶ Microsoft Visual Studio .NET
- ▶ The Microsoft .NET Framework V1.1
- ▶ The Microsoft .NET Framework SDK

Notes on installing WebSphere Studio Application Developer

The default installation of WebSphere Studio Application Developer V5.1 (WebSphere Studio) does not include the ActiveX Bridge. In order to install the bridge and the samples, you must select the **ActiveX Bridge** under “Optional components” in the installation wizard.

In WebSphere Studio, it is not recommended that you take the default installation path for WebSphere Studio because it uses “Program Files” as the default. Spaces in directory names can cause problems when executing batch commands under Windows. We recommend you choose a directory without spaces or other unusual characters in the name. We refer to the directory in which WebSphere Studio is installed as the WebSphere root.

Starting the IDEs and applications

In order to use ActiveX Bridge, the WebSphere environment variables must be set *before* you start any IDE if you are developing or debugging, and before you run any sample code if you are simply executing. These classpath variables include information and many other things needed for the WebSphere Studio environment.

The ActiveX Bridge provides batch files to set these variables under the following path: <WebSphere_root>\AppClient\bin\.

The most critical of these is setupCmdLineXJB.bat, which is used to set environment variables required by WebSphere Studio and WebSphere to run Java programs. This is illustrated in Example 6-3. You have the option of using these scripts to set up the environment and then starting the IDEs or programs from the command line after executing them, or adding these variables to the System Environment Variables.

Example 6-3 setupCmdLineXJB.bat

```
set NAMING_FACTORY=com.ibm.websphere.naming.WsnInitialContextFactory
set PATH=%JAVA_HOME%\bin;%JAVA_HOME%\bin\classic;%PATH%
set XJBRE_CLASSPATH=-Djava.ext.dirs=%JAVA_HOME%\lib\ext;%WAS_HOME%\classes;
%WAS_HOME%\lib\ext;%WAS_HOME%\lib;%WAS_HOME%\properties;%JMS_PATH%
set XJBWAS_CLASSPATH=-Dws.ext.dirs=%JAVA_HOME%\lib\ext;%WAS_HOME%\classes;
%WAS_HOME%\lib\ext;%WAS_HOME%\lib;%WAS_HOME%\properties;%JMS_PATH%
```

The WebSphere documentation describes these files and their use in more detail.

Code development

This section describes in some detail the software we built. It start with a simple description of the responsibilities of each bit of code and then provides details on how the code interacts and how to set up and execute the samples themselves.

In our solution, we develop three modules: the service provider, a COM wrapper and the Consumer. These are depicted in Figure 6-10 and described below.

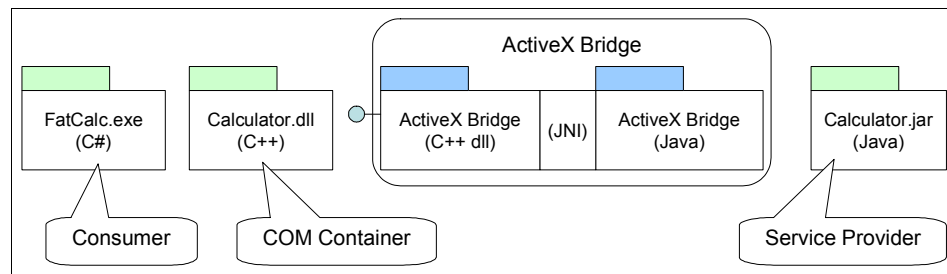


Figure 6-10 Simple scenario code components

Service provider

The service provider we use is the same Calculator.jar file as in Chapter 7, “Scenario: Synchronous stateless (WebSphere producer and .NET consumer)” on page 297 and Chapter 8, “Scenario: Synchronous stateless (WebSphere consumer and .NET producer)” on page 329.

The service provider’s responsibilities include:

- Providing business functionality

The business functionality described here is that of an old-style adding machine. We may set the current total (for example, setting it to 0 is analogous to pressing the Clear key on a calculator). We can then add any value we want to the current total and get the result. At any time, we may simply check the current total as well.

- Providing an interface

The calculator implementation assumes it is being invoked via normal Java implementations by a Java object. In other words, it does not implement JNI or consider non-Java data types.

- Maintaining state between invocations

The calculator maintains state in the form of a private float variable called `CurrentState`. Because state is kept by the service provider, neither the consumer nor the COM DLL need concern themselves with the current value until they need it or wish to reset it.

COM Wrapper DLL

A C++ COM DLL is written which handles the ActiveX Bridge invocation on behalf of the .NET consumers. For simplicity, this COM DLL is written to function as a library application, which means it will not run as a stand-alone server process under the Service Control Manager, but instead will run the consumer's process. For more information on creating a server-side COM object, see the Microsoft Developer Network (MSDN) documentation at:

<http://www.microsoft.com>

The responsibilities of the COM ActiveX Bridge Gateway DLL are as follows:

- ▶ Isolating interaction with the ActiveX Bridge and the service provider into a COM component.

This includes creating an instance of the JClassFactory and invoking all necessary methods to create the JVM, find the class required, create an instance of it and utilize its functionality.

- ▶ Providing an COM approximation of the Java business interface for the consumer to use.

Although it is possible to further abstract the Java interfaces within the COM DLL, making it much more easily reusable for different Java objects, we do not cover these techniques for the sake of simplicity and instead implement an approximation of the business interface (but not the implementation) of the service provider in the COM DLL.

- ▶ Handling any data type mapping required between COM and ActiveX.

This example does not contain data mapping issues.

Service consumer

Our consumer in this case will be a Microsoft .NET Windows Forms application, or fat client, written in C#. It has a simple interface, allowing the user to interact, and uses the COM ActiveX Bridge Gateway DLL to obtain the service from the Service Provider.

The consumer's responsibilities include:

- ▶ Referencing the COM representation of our Java object.

The COM DLL acts as the interface gateway between our .NET object and the ActiveX Bridge.

- ▶ Using the business interface properly (setting and getting the current value property, using the add method correctly).

Writing the code

Here we discuss the code implementation for the samples involved in our simple scenario.

Service provider - COM Wrapper DLL

This object is created in Microsoft Visual Studio.NET. Follow these steps to create the project:

1. Start Microsoft Visual Studio.NET.
2. Create a new solution by clicking **File -> New -> Blank Solution**.
3. Select a project type of Visual C++ Projects.
4. Give it the name CalcWrapper and decide on a location for your solution and project file. Select an **ATL Project** as the template from which you wish to create this project.
5. The ATL Project Wizard will appear. We don't need anything special here, so just click **Finish**.

Note: We use the Active Template Library (ATL) for simplicity of creation.

The resulting solution now contains a set of source files you may edit to provide your implementation.

6. Add a Class to the project:
 - a. Select **Project -> Add Class**.
 - b. From the Add Class wizard, select **ATL Control** as the type of object you wish to create.
 - c. Within the ATL Wizard, give the new class a shortname of Calculator.
 - d. Click **Finish** and the new calculator class will be created. You will be presented with new source files within your project, two of which are called Calculator.cpp and Calculator.h. These files will contain your implementation of the COM Calculator interface.
 - e. Right now, your project should look something like Figure 6-11 on page 284.

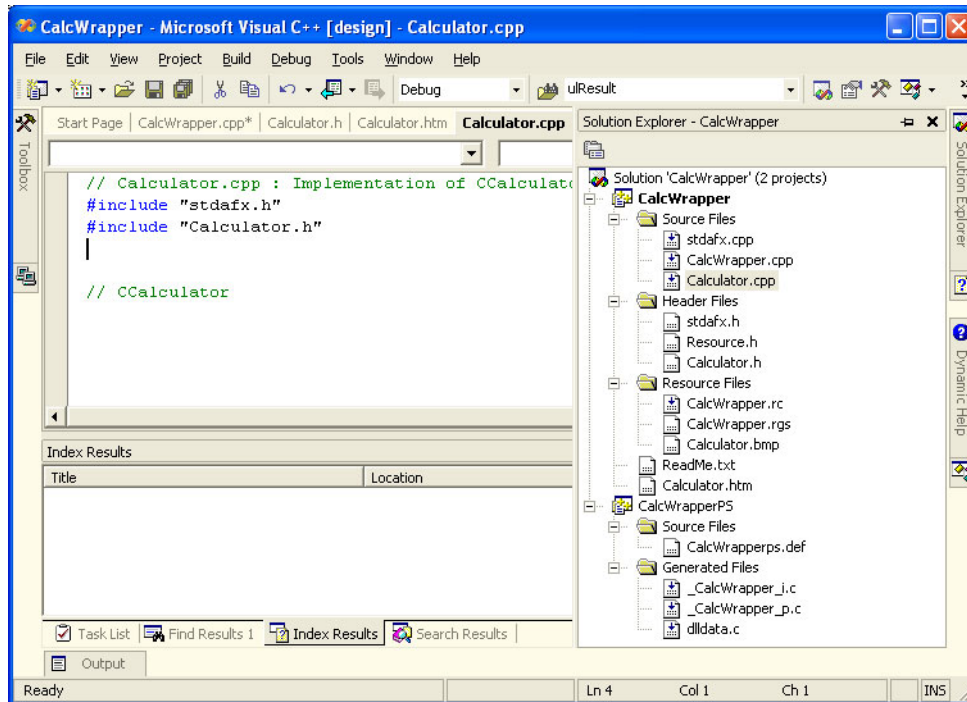


Figure 6-11 Visual Studio after creation of our blank Calculator ActiveX object

- f. Implement the interface by copying in our Calculator.h and Calculator.cpp files, or implement for yourself. Our approach was as follows:
 - i. Create public objects for the JClassFactory, JClassProxy and JObjectProxy; we will need to access the service provider.
 - ii. Within the constructor, create a new JClassFactory object. Use that object to create and initialize the Java Virtual Machine with the XJBIInit() method. Use the FindClass() method to find the Calculator class in Java and then the NewInstance() method to create a new instance of that class. At this point, all methods and fields within our Java implementation of Calculator are available to us.

Tip: The constructor of our object is where you initialize the JVM. That is the most difficult piece of our wrapper because of the settings required. Take a look at our sample code. If you try this and it doesn't work, it is very likely that environment variables were not set prior to execution, so check that first.

- iii. Once the constructor is set up to initialize and make available the Calculator implementation, we need only create methods that wrap the Java method as precisely as possible. It is possible to aggregate services here if you so choose (change the interface to better suit the consumer) or you can implement it exactly as it is in Java. For this, we chose to create the following public methods whose purpose it is to simply wrap calls into the Java calculator using the ActiveX Bridge objects which were created and initialized during the constructor. Our C++ code is shown in Example 6-4.

Example 6-4 Java method wrappers in our COM Wrapper DLL implementation

```
/* *****
 * Method definitions for our Calculator COM Wrapper.
 * ***** */
// wrap the setCurrentTotal method
STDMETHODIMP CCalculator::setCurrentTotal(FLOAT arg) {
    moCalcJObjectProxy.setCurrentTotal( arg );
    return S_OK;
}
// wrap the getCurrentTotal method
STDMETHODIMP CCalculator::getCurrentTotal(FLOAT* fCurrentTotal) {
    fCurrentTotal = moCalcJObjectProxy.getCurrentTotal( );
    return S_OK;
}
// wrap the add method
STDMETHODIMP CCalculator::add(FLOAT arg, FLOAT* fCurrentTotal) {
    fCurrentTotal = moCalcJObjectProxy.getCurrentTotal( arg );
    return S_OK;
}
```

- iv. Now simply build the object and you are finished with the wrapper.

Tip: If you are unfamiliar with the requirements for creating methods under an ActiveX Control beyond simply adding the C++ code shown in Example 6-4, an easy way to do this is to use Visual Studio's Add Method operation. This is done by opening the Class View, right-clicking the **ICalculator** interface and choosing **Add -> Add Method**.

Service consumer

This object is created in Microsoft Visual Studio.NET. Create this project yourself by performing the following steps:

1. Start Microsoft Visual Studio.NET.

2. Create a new Solution by clicking: **File -> New -> Blank Solution**.
3. Select a project type of **Visual C# Projects** and select **Windows Application**.
4. Give it the name `FatCalc` and decide on a location for your solution and project file.
5. At this point, the new project will be created and you will be presented with a blank form. We added the following objects to that form:
 - a. A text Box called `AddMe` to provide the user the ability to enter a number to add.
 - b. A text box called `CurrentTotal` to provide the user a way to see the result and to edit the current total if desired.
 - c. A button called `SetCurrentTotal` to invoke the `setCurrentTotal()` method of our Java calculator.
 - d. A button called `Add` to invoke the `add()` method.
6. There are three simple sections of code to write:
 - a. Create an instance of our COM Wrapper:
 - i. Create a reference to the COM Wrapper Object. This can be accomplished under Solution Explorer by expanding the solution and right-clicking **References**, then selecting **Add Reference**.
 - ii. In the ensuing dialog, click the **COM** tab and find our `CalcWrapper` class. In this same tab, you may also browse to find the `CalcWrapper.dll`.
 - iii. Once it is found, click **OK** to add the reference. Visual Studio will add the reference, automatically create a proxy Interop assembly called `Interop.CALCWRAPPER.dll` which provides access to the unmanaged COM component from .NET. A reference to this assembly is what you use from within .NET.

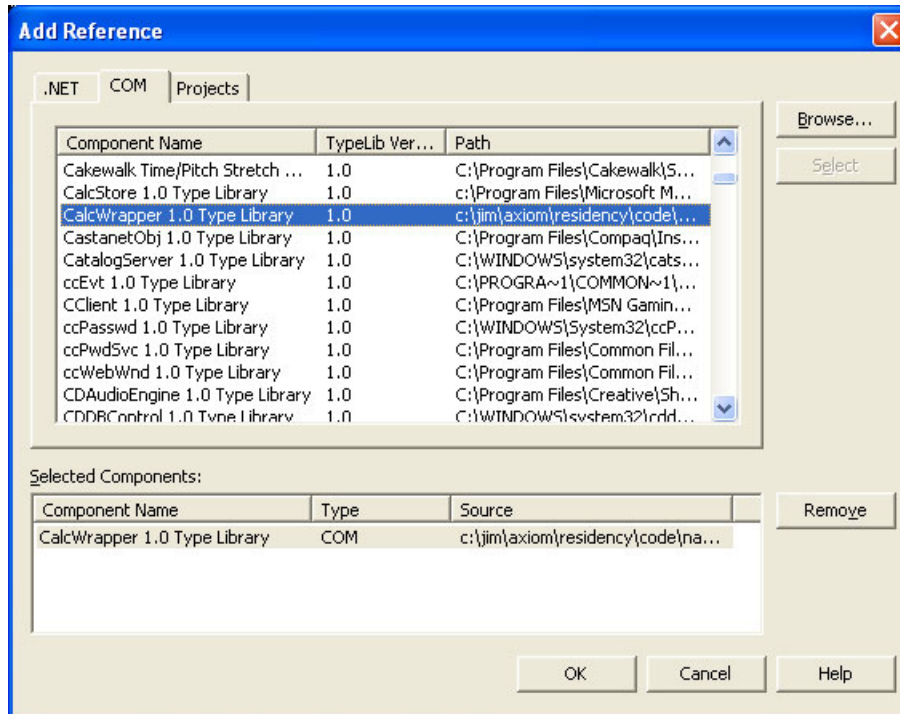


Figure 6-12 Adding the CalcWrapper reference to our C# project

- iv. Once the reference exists, we can add the code to create the instance of our wrapper. We chose to implement that code under the `FatCalc_Load()` method so that the object is created when the form is loaded initially. The following code snippet will give you the idea.

Example 6-5 Creating an instance of our COM Wrapper Object within C#

```
// declare our object variable to hold the calculator wrapper
public CCalculatorClass oCalc;
...
...
private void FatCalc_Load(object sender, System.EventArgs e)
{
    //create instance of our wrapper
    oCalc = new CalcWrapper.CCalculatorClass();
}
```

- v. Now we are ready to use the calculator.
 - b. Implement the `SetCurrentTotal` button.

Example 6-6 Setting the Current Total from the Consumer

```
private void SetTotal_Click(object sender, System.EventArgs e)
{
    oCalc.setCurrentTotal( CurrentTotal.Text );
}
```

c. Implement the Add button.

Example 6-7 Implementation of the Add button within the Consumer

```
private void Add_Click(object sender, System.EventArgs e)
{
    CurrentTotal.Text = oCalc.add( AddMe.Text );
}
```

Executing the programs

We can now execute our application and observe its operation within the IDE or from the command line. The following two procedures show these two choices, respectively.

Executing from within Visual Studio

The following steps illustrate operation of our fat calculator from within the Microsoft Visual Studio Integrated Development Environment.

1. If you have not registered the environment variables contained in the setupCmdLineXJB.bat file, perform the following steps.
 - a. Exit Visual Studio.NET.
 - b. Open a command prompt window by running **cmd.exe**.
 - c. Run the batch file setupCmdLineXJB.bat from
<WebSphere_root>\AppClient\bin.
 - d. Use the **CD** command to change directory to where your Visual Studio Solution file for FatCalc is located. This file should be called FatCalc.sln.
 - e. Type **FatCalc.sln** on the command line. This will open a new instance of Visual Studio.NET with the FatCalc solution already open.
 - f. Set any breakpoints as desired by selecting the line where you wish to break and pressing the **F9** key.
 - g. Press the **F5** key to run the program in debug mode under the IDE.

Executing from the command line

In order to run the fat calculator from the command line, do the following:

1. Open a command prompt under Windows by running **cmd.exe**.

2. Run the batch file `setupCmdLineXJB.bat` from `<WebSphere_root>\AppClient\bin`. This will set up the environment variables for execution. If you chose to add the environment variables contained within that batch file to your system path and have already done so, you can skip this step.
3. Execute `fatcalc.exe` from the command line.

6.3 Solution model using the Interface Tool for Java

Our example does not focus on the Interface Tool for Java solution; this section will briefly explain an implementation in the interest of providing a better understanding of its use.

Note: The Interface Tool for Java is an IBM alphaWorks® technology; you can find more information about the tool at the following URL:

<http://www.alphaworks.ibm.com/tech/bridge2java>

The tool was formerly called Bridge2Java, and in some documents you may find it referred to it by its former name.

Currently, most of the effort surrounding integration between Java clients and .NET assemblies is focused on Web Services. Today, Web Services provide stateless invocation of .NET assemblies from Java clients. But what if you need stateful integration?

IBM has a technology called Interface Tool for Java which was originally conceived to deliver stateful integration between Java clients and COM components. Interface Tool for Java can also be used to provide stateful integration between Java clients and .NET assemblies.

Before we describe how Interface Tool for Java can provide Java clients with stateful integration to .NET assemblies, let's briefly consider how Interface Tool for Java provides integration to COM Components (this is important for an understanding of the Java to .NET solution). Most COM Components implement an interface called `IDispatch` (this is often referred to as an automation interface). Interface Tool for Java works with this `IDispatch` interface in two ways:

- ▶ The Interface Tool for Java build-time generates Java proxies for COM `IDispatch` interfaces.
- ▶ The Interface Tool for Java runtime provides the runtime binding, marshalling, and type mapping, maintains Java to COM object relationships, and performs object life cycle management between each Java proxy instance and its

associated COM class (CoClass) instance. It does this using the IDispatch interface for the given COM component.

These Interface Tool for Java generated Java proxies to COM CoClasses enable any Java-based technology (for example: EJB, Java Bean, Servlet, JSP and so on) to statefully integrate with the proxied COM components. This is illustrated in Figure 6-7 on page 273.

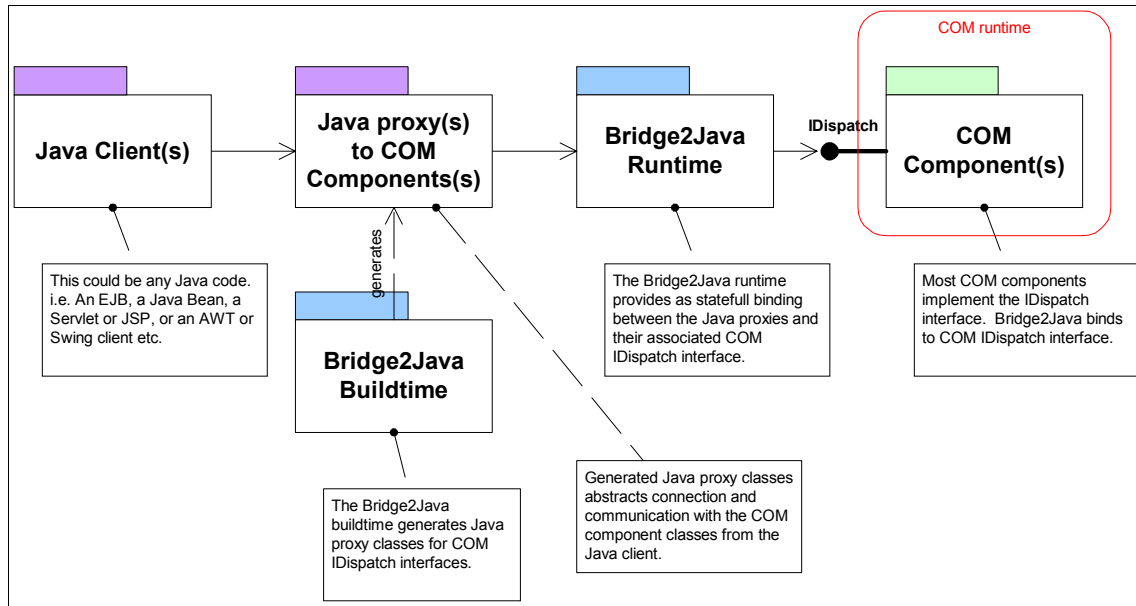


Figure 6-13 The Interface Tool for Java solution model for COM integration

.NET assemblies do not normally implement IDispatch interfaces, but their interfaces can be represented by IDispatch interfaces. The .NET SDK provides a utility called **tlbexp.exe** which, with the correct directives, can be used to generate type information that can include an IDL definition of IDispatch interfaces. When this type information is correctly registered in the Windows registry, the .NET runtime will automatically provide a façade to the .NET assembly for IDispatch clients (in this case the Interface Tool for Java runtime is the IDispatch client). This is illustrated in Figure 6-8 on page 275.

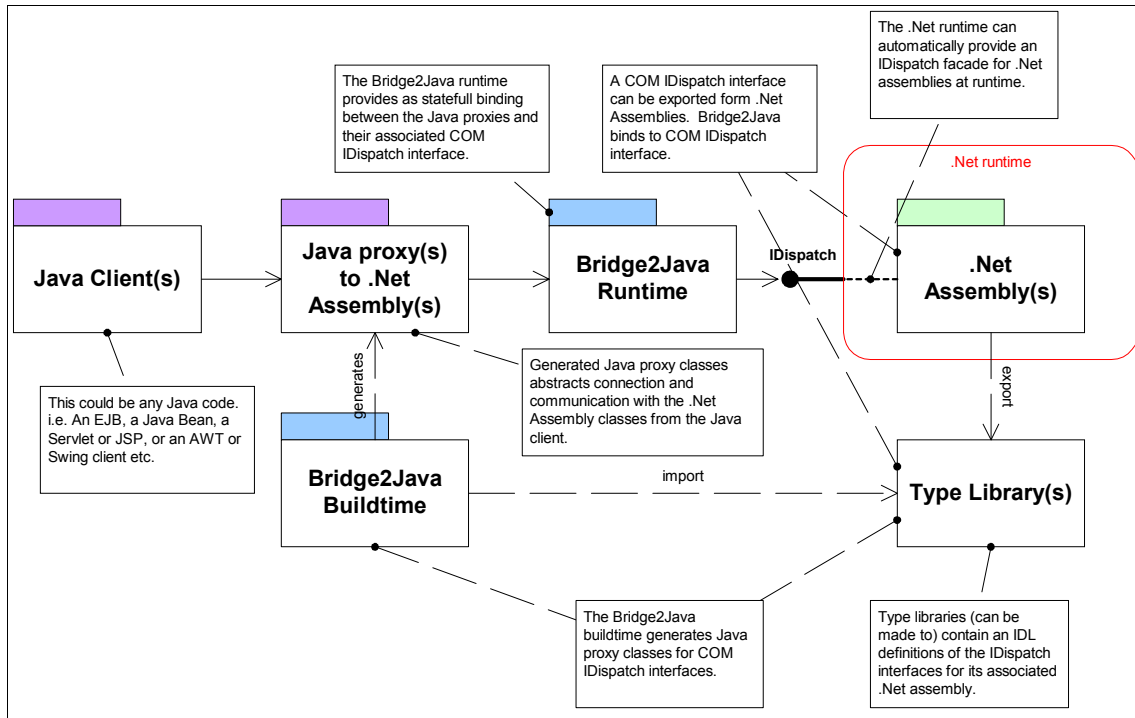


Figure 6-14 The Interface Tool for Java solution model for .NET integration

How to implement an elementary solution

We have contrived a very simple example that demonstrates how to use Interface Tool for Java to provide stateful integration between Java clients and .NET assemblies. We have deliberately used a narrow scope in this example so that we can focus on the integration technologies (see 6.1.2, "Considerations" on page 266 for a broader solution model).

Consider Figure 6-15 on page 292; it illustrates a simple class model with a single .NET assembly Calculator that contains a single class Calculator.Class1, Interface Tool for Java runtime binding via IDispatch, Java proxies com.calculator.Class1 and com.calculator._Class1 generated by Interface Tool for Java build-time, and a Java client com.clients.Client1. The .NET class has a stateful interface. The client must first set arguments using set_arg1() and set_arg2(), then invoke the add() method to add the arguments together. For this to work correctly, a single instance of a .NET calculator object Calculator.Class1 must be associated with the Java client com.clients.Client1 for the life of the Java proxy com.calculator.Class1. As we mentioned earlier, .NET assemblies do not normally implement the IDispatch interface. In order to make the .NET assembly accessible via IDispatch, we need to apply some compile attributes to the .NET

classes or interfaces we want to access via IDispatch (read on for a technique that does not modify the original source code). We will then be able to generate a type library which describes the IDispatch interface for the target .NET classes. We use the type library with the Interface Tool for Java build-time to generate the Java proxies.

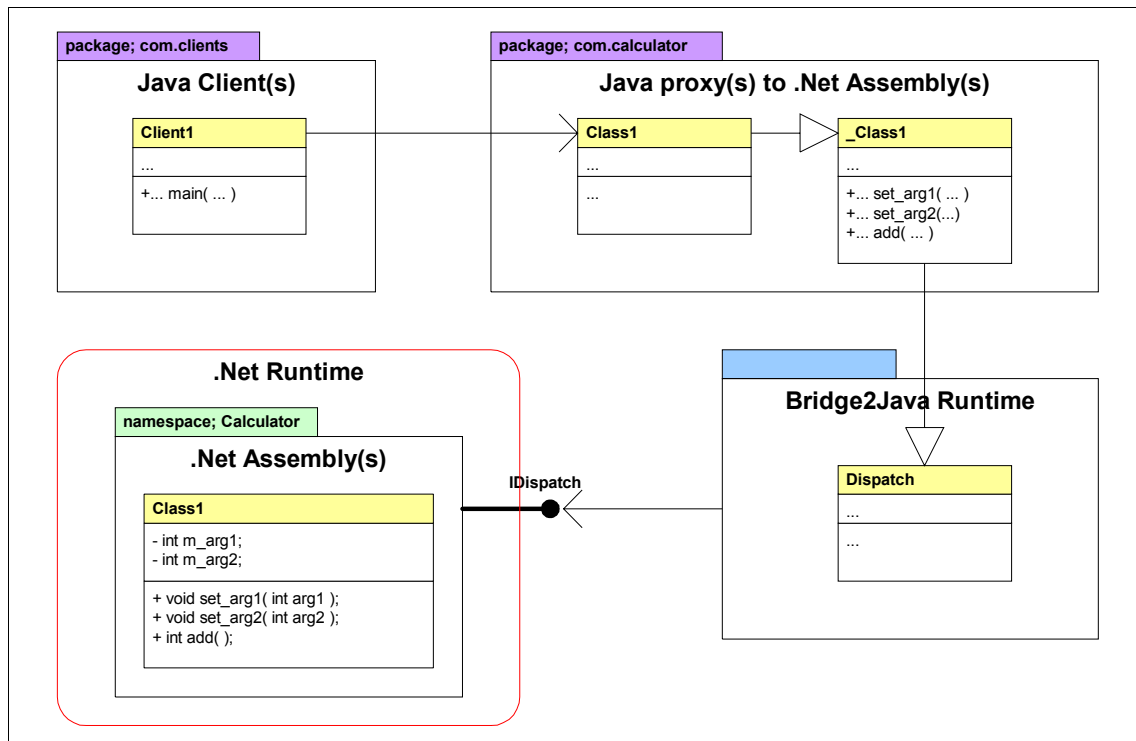


Figure 6-15 Simple class model for .NET integration

What if you do not have the original source code or you simply do not want to, or cannot, modify the original source code? Actually, this is not a problem. Consider the solution model illustrated in Figure 6-16 on page 293. This model is very similar to Figure 6-15, except that we have added a new .NET assembly that has a single class `Calculator2.Class1` which extends our original .NET class `Calculator.Class1`. Using this technique, our thin subclass (it is just a class declaration without any implementation code; see the sample code) has the `IDispatch` interface attribute declaration, but inherits all its methods from the original superclass implementation. This inheritance association is made with the binary implementation of the .NET superclass, not with the source code (for example, you do not need source code for the superclass to use this technique).

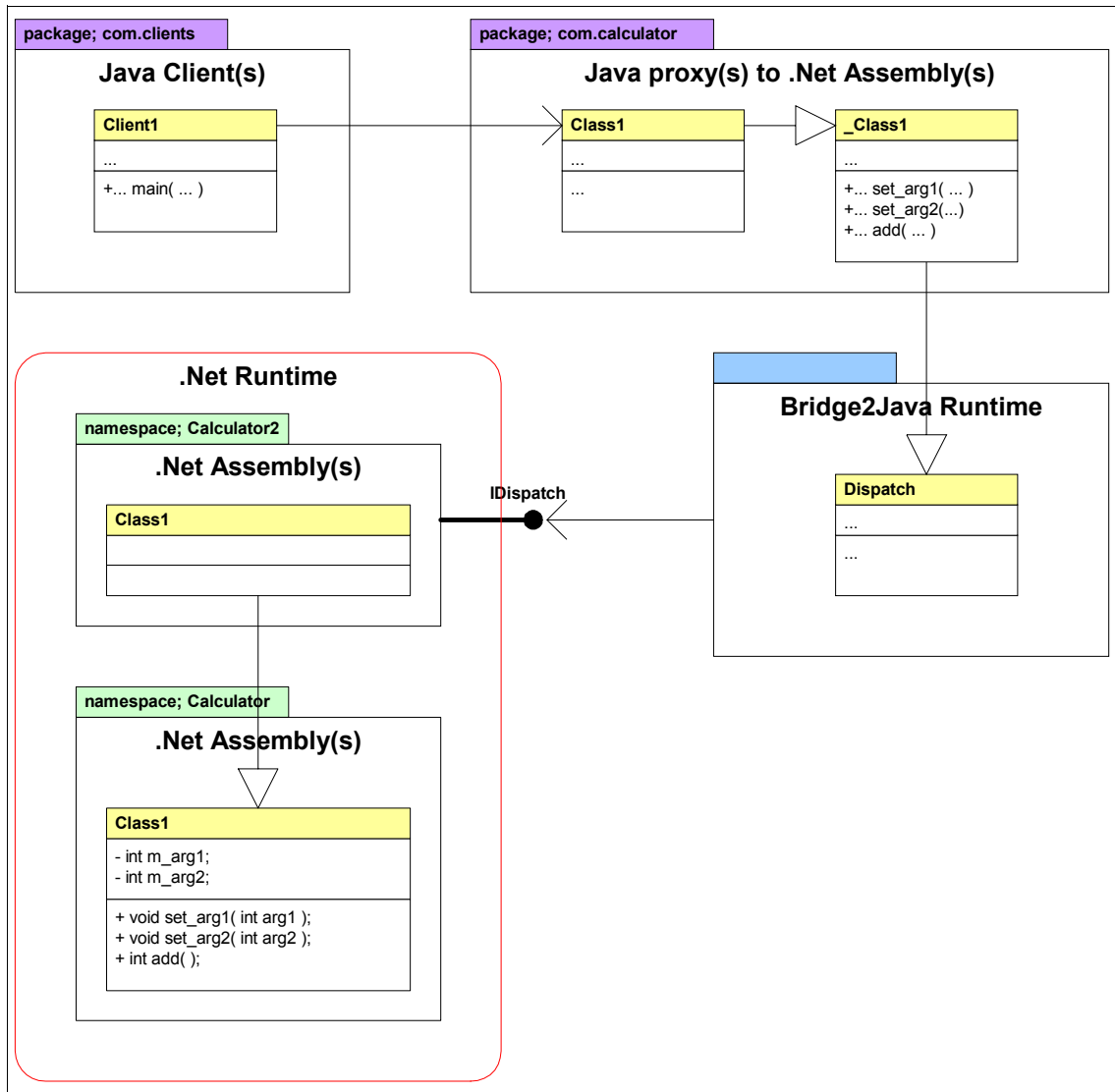


Figure 6-16 Class model for .NET integration using a .NET subclass

Implementation activity

Figure 6-17 on page 294 illustrates the activities involved in building this elementary integration between a Java application and a .NET application.

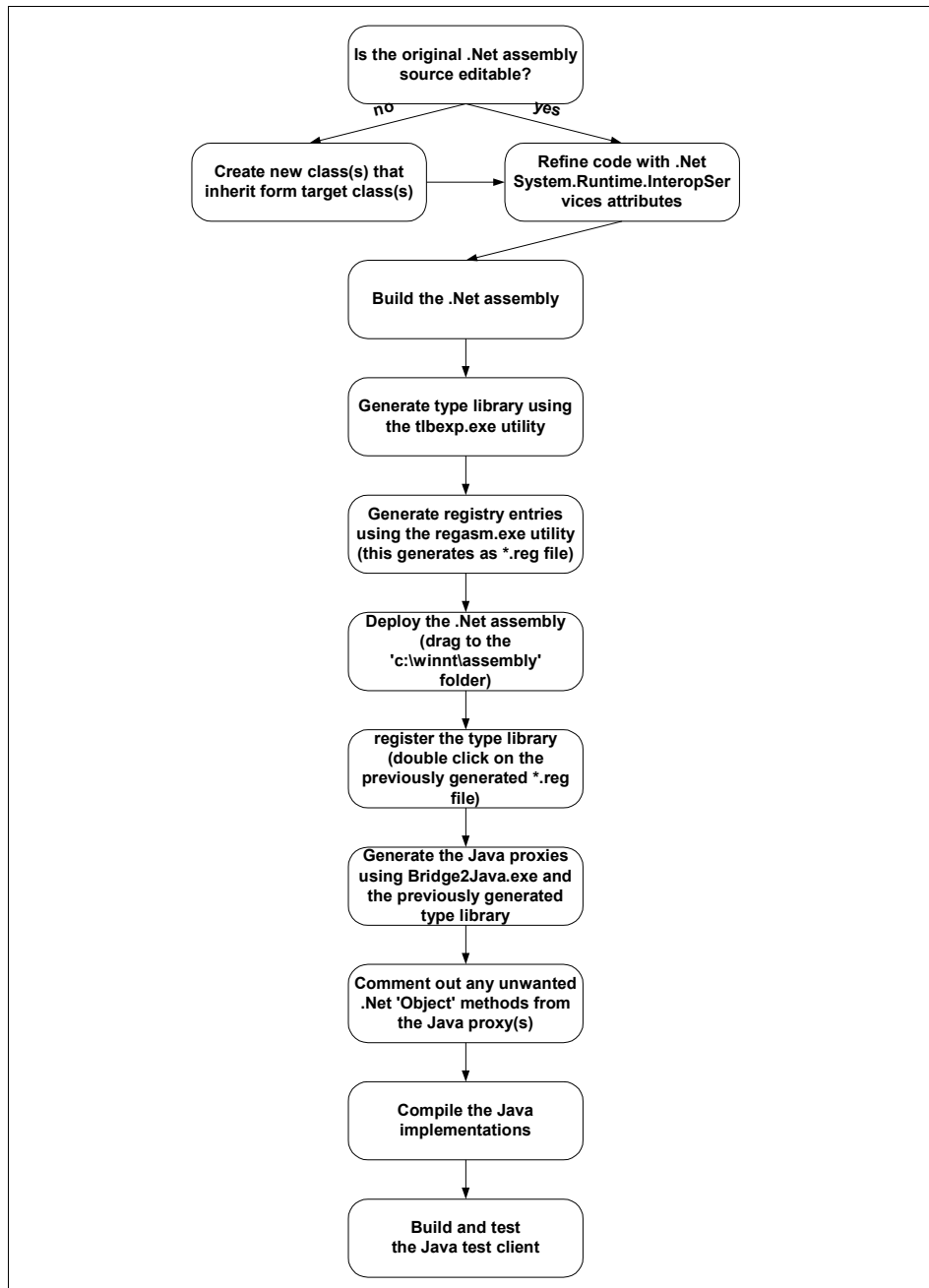


Figure 6-17 Integration activities between Java and .NET using Interface Tool for Java

Generating the type library

Use the .NET SDK command line utility **tlbexp.exe** to generate a type library from a built .NET assembly.

For example: **tlbexp.exe Calculator2.dll** will generate Calculator2.tlb.

Registering the type library

Use the .NET SDK command line utility **regasm.exe** to generate the registry entries for the IDispatch clients.

For example: **regasm.exe Calculator2.dll /regfile:Calculator2.reg** will generate Calculator2.reg.

Applying the registry entries

Double-clicking a .reg file will apply it to the registry. For scripted deployment, you can use **regedit.exe**.

For example: **regedit /s Calculator2.reg** will apply the Calculator2.reg file to the Windows registry.


Other considerations

The example illustrated in this chapter uses the `System.Runtime.InteropServices.ClassInterfaceAttribute` attribute to expose all .NET class methods for interoperability. Finer-grained method exposure can be defined using the `System.Runtime.InteropServices.ComVisibleAttribute` attribute. Consider exposing interfaces, rather than classes, using interoperability attributes on these interfaces to expose only the required methods to the IDispatch interface.

While exploring the use of Interface Tool for Java with .NET, we encountered some problems with registry entries not being correctly set. If your test client fails to bind to the .NET object correctly, check the registry entries. The value of the `InProcServer32` key was sometimes set incorrectly by `regasm.exe`. It should be set to `mscorlib.dll` to enable the .NET runtime to provide an IDispatch callable wrapper to the targeted .NET assembly.

For details, see:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconregisteringassemblieswithcom.asp>



Scenario: Synchronous stateless (WebSphere producer and .NET consumer)

The purpose of this chapter is to provide an implementation of the stateless synchronous interactions using both the remote procedure call (RPC) and document-oriented call paradigms described in Chapter 4, “Technical coexistence scenarios” on page 109.

The first section of this chapter elaborates the problem previously identified in 4.2.2, “Stateless synchronous interaction” on page 115.

In 7.1.2, “Considerations” on page 300 we examine the technical considerations that were taken into account when designing the solution.

The solution itself is presented in two parts: a simplified solution model and an extended solution model. In the simplified solution model, we provide a very basic set of implementations that illustrate how to achieve integration between WebSphere Application Server and Microsoft .NET using both RPC and document-oriented call paradigms. The extended solution model illustrates how the simplified solution model may be extended to incorporate the full interaction

scenarios identified in Chapter 4, “Technical coexistence scenarios” on page 109.

The final section of this chapter presents recommendations on best practises for implementing the integration solutions.

7.1 Problem definition

This section describes some of the issues in integrating stateless synchronous communication between application components residing in WebSphere Application Server and Microsoft .NET.

7.1.1 Description of the problem

The problem addressed in this chapter is how application artifacts deployed in a WebSphere Application Server environment can be made to interact with application artifacts deployed in a Microsoft .NET environment. Figure 7-1 illustrates the nature of the interaction.

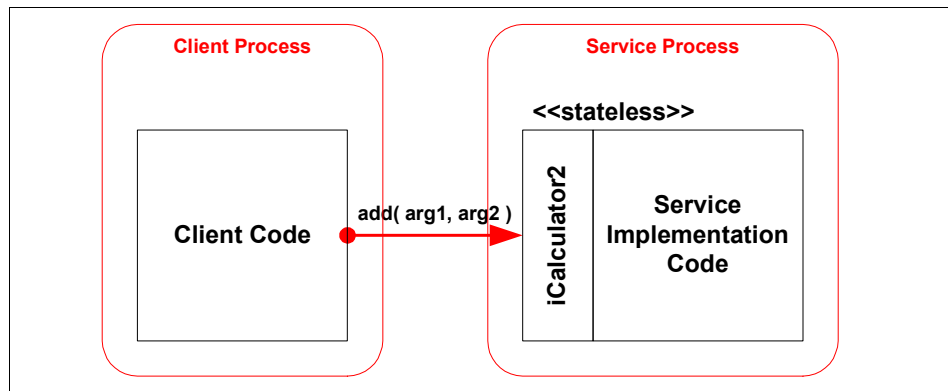


Figure 7-1 Stateless Synchronous Interaction

What this illustration shows is essentially a service-oriented architecture. When we consider such an architecture, we often talk in terms of a service provider which provides some service and a service consumer who calls or consumes the service. In this chapter, the service is provided by a Microsoft .NET application artifact and is consumed by a WebSphere Application Server application artifact.

In this particular scenario, we are only concerned with implementing *synchronous* communications between the two entities. That is, the consumer sends a request to the service provider and then blocks or waits for the response before continuing processing.

The final piece of the puzzle in defining the problem is that we declare that the communication occurs in a *stateless* manner. That is to say, each interaction between service consumer and provider occurs without any knowledge of any previous interactions between the two.

7.1.2 Considerations

In designing and implementing a solution for the scenario, there are several considerations that should be taken into account. This section presents a discussion of these considerations.

Transport mechanism

For communication to occur between WebSphere Application Server application components and Microsoft .NET components, we have to employ some mechanism for making calls and exchanging data between the two. In selecting an appropriate transport mechanism, there are some fundamental questions that should be considered:

- ▶ Is the transport mechanism independent of the underlying platform, programming languages and distributed component technologies?
- ▶ What transport protocols are supported by it, and are they suitable for integrating the two technologies?
- ▶ Does the transport mechanism support the characteristics of the interaction we are implementing? For instance, in this scenario, it must be capable of supporting stateless synchronous interactions.
- ▶ Finally, and perhaps most importantly, is the transport mechanism supported by both WebSphere Application Server and Microsoft .NET?

If no appropriate transport mechanism exists, then we must consider whether we are willing to implement one. In practice, developing a transport mechanism from scratch is seldom a desirable option. Fortunately, it is often more convenient to leverage existing transport technologies by writing adapters that hide the complexities of the underlying transport mechanism from the application.

In this particular scenario, selecting the most appropriate transport mechanism was relatively straightforward. This is because both WebSphere Application Server and Microsoft .NET technologies provide support for Web Services and support a common technology for implementing service-oriented architectures in a distributed stateless synchronous manner, which is SOAP.

SOAP is in fact a specification for the exchange of structured information in a decentralized, distributed environment. It is particularly suitable for our solution in a number of ways:

- ▶ It was designed to enable communication between the actors in a service-oriented architecture. In SOAP, there are three actors: the service consumer and service provider, which we have discussed already, and a service broker. The service broker is responsible for making information about how to access the Web Service available to the service consumer.

- ▶ SOAP is also operating system independent and not tied to any programming language or component technology.
- ▶ It is, in principle, transport protocol independent and can therefore encapsulate a variety of protocols and applications, such as HTTP, the HTTP extension framework, and message-oriented middleware.

For a more detailed discussion of Web Services and the SOAP protocol, refer to the IBM Redbook *WebSphere Version 5 Web Services Handbook*, SG24-6891.

Data model

Applications developed using object-oriented programming languages, such as Java and C#, are written using programming language specific objects and data types. One of the issues in achieving interoperability between different programming languages is how to provide a language-independent abstraction for common language data types.

SOAP addresses this issue by promising interoperability between different programming languages. It does this by providing a data model, which enables a language-independent abstraction for common programming language types. The SOAP data model supports the following data types:

- ▶ Simple XML Schema Definition (XSD) types

These are used to represent the primitive data types found in most programming languages, for instance types such as, int, char, long etc.

- ▶ Compound types

There are two kinds of compound types: structs and arrays.

- Structs

A struct is essentially a grouping of primitive types accessed by a unique name. Structs are conceptually similar to records in languages such as Pascal or methodless classes with public data members in object-based programming languages.

- Arrays

Elements in an array are identified by position, not by name. This is the same concept found in languages such as Java and C#. SOAP also supports partially transmitted and sparse arrays. Array values may be structs or other compound values. Also, nested arrays (which means arrays of arrays) are allowed.

As we can see, while transport mechanisms such as SOAP may address data typing issues between programming languages, the mappings available are generally limited. This is not by design, but really just a consequence of the

difficulties of abstracting from a specific programming language data model to a common model.

Interface definitions

Applications are designed and written to leverage the full feature set of the programming language in which they are developed. Thus, in providing a solution to our interoperability scenario, it is necessary to consider how to abstract Java and .NET objects into some common form such that they can be marshalled for transmission across the wire and unmarshalled at the receiving end. The manner in which this is implemented must, of course, take into consideration any restrictions imposed by the chosen transport mechanism.

In the light of the discussions concerning transport mechanisms and data modelling, we should consider how best to design the interfaces between our application and the underlying integration technologies. These should be designed such that the service provider and consumer implementations are abstracted from the complexities of:

- ▶ Interoperability between different technologies.
- ▶ The underlying transport mechanism.
- ▶ Data modelling differences.

Security

If the security policy states that access to services should be restricted to only authorized groups or users then consideration must be given as to how security credentials can be passed between the technologies.

The solution presented in this chapter does not implement security; however, for a fuller discussion of security-related interoperability issues, refer to 11.4, “Security” on page 484.

The IBM Redbook *IBM WebSphere V5.0 Security*, SG24-6573, presents a full discussion of security within WebSphere Application Server.

You may also refer to the IBM Redbook *WebSphere Version 5 Web Services Handbook*, SG24-6891, which documents how security is implemented in Web Services.

Transactions

Transactional models are implemented within both WebSphere Application Server and Microsoft .NET. Where transactions are a business requirement, consideration has to be given about how to maintain transactional integrity across technologies.

For purposes of clarity, the solution presented here does not address the issue of implementing transactions. For a discussion of the issues in implementing transactions between WebSphere Application Server and Microsoft .NET components, refer to 11.5, “Transactionality” on page 489.

7.2 Solution model

This section describes the solution model for the stateless synchronous interaction between WebSphere and .NET.

7.2.1 A solution to the problem

This scenario concerns the situation where some functional code exists in the business layer of a WebSphere environment which we need to access and interoperate with from a Microsoft .NET environment.

7.2.2 Service provider

Assume there exists some code in a WebSphere environment that we wish to make available to a .NET application. In this scenario, we will achieve interoperability with Web Services technology, so the first step is to Web Service enable the code on the WebSphere system. This means enabling the back-end functional code to be accessed and invoked using SOAP. This can be done most easily using the WebSphere Studio Application Developer tool, and the process for this is described in this chapter.

In our example solution, we will be using the Calculator example. We will use the interfaces described as part of the Calculator (ICalculator1, ICalculator2 and ICalculator3) to show different ways of achieving the interoperability.

The code (and classes) that we need to enable as a Web Service is contained in a JAR file, Calculator.jar.

Calculator.jar contains a class called CalculatorService.java with these methods:

- ▶ public void setCurrentTotal(float arg)
- ▶ public float getCurrentTotal()
- ▶ public float add(float arg1)
- ▶ public float add(float arg1, float arg2)
- ▶ public float add(ICalculator3Args args)

The top three methods are for use with the ICalculator1 interface, which is intended for stateful interaction. This chapter focuses on stateless

communication, and hence we will not be considering ICalculator1 in this chapter. At the time of writing, stateful Web Services are not yet possible.

The add(float arg1, float arg2) method is for use with the ICalculator2 interface, and add(ICalculator3Args args) is for the ICalculator3 interface. Both of these imply stateless interaction, and we will focus on enabling these two methods as the Web Service methods.

The ICalculator2 method will be used to implement a Remote Procedure Call (RPC) style Web Service, and the ICalculator3 method will be used to implement a document-style Web Service. This is intended to show the capabilities of Web Service interoperability using the two communication models, as described in 4.2.5, “RPC interface style” on page 123 and 4.2.6, “Document interface style” on page 124.

The class hierarchy of the service provider side of the Calculator implementation is shown in Figure 7-2.

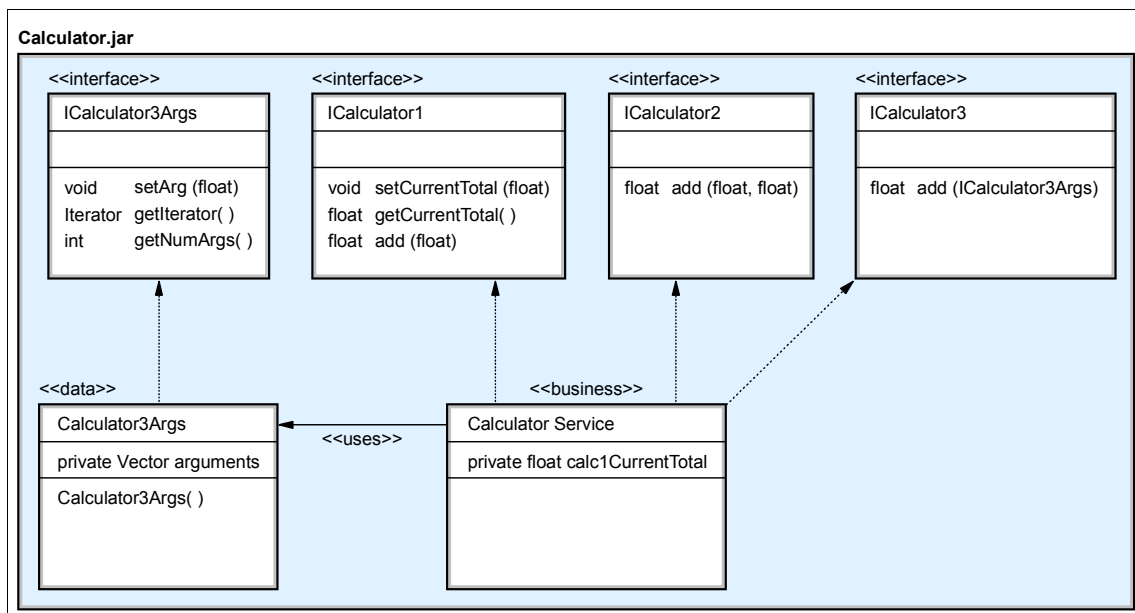


Figure 7-2 Calculator.jar

To view the actual implementation of the code within Calculator.jar, simply create a new Java project in WebSphere Studio Application Developer, for example: **CalculatorCode**, click **File -> Import -> Zip file**, and select **Calculator.jar**.

WebSphere Studio Application Developer has support for three types of Web Services interaction:

- ▶ RPC/Literal
- ▶ RPC/Encoded
- ▶ Document/Literal

The style of interaction of a Web Service is specified in the WSDL belonging to that Web Service.

Scenario implementation: getting started

To enable the existing Calculator code functionality as Web Services, we will be using WebSphere Studio Application Developer. Start WebSphere Studio Application Developer, and open up a new (blank) workspace.

Create an Enterprise Application Project (EAR) which is where we will create our Web Services. Go to **File -> New -> Project -> J2EE -> Enterprise Application Project** and name the project `SimplisticWS2NETCalculator`.

Scenario implementation with the RPC model

We will use the `ICalculator2 add(float arg1, float arg2)` method to demonstrate Web Services interoperability using RPC style communication.

In order to create a Web Service from existing code in WebSphere Studio Application Developer, the code (or the classes) need to be imported into the WebSphere Studio Application Developer tooling so it can be used by the generated Web Service. The tooling will generate Web Service classes which will act as a proxy to the existing code, handling all transport issues and invoking the existing methods. The actual source code (as opposed to compiled classes) is not required by WebSphere Studio Application Developer, since Web Services can be created from compiled Java code.

We assume in this scenario that you have the existing code available (as opposed to the compiled classes), but it makes no difference to the processes involved.

If the existing code is contained within a J2EE Enterprise Application (EAR) project, the user could import the EAR file into WebSphere Studio Application Developer for use in the Web Service creation. However, in our example, the existing code is contained as part of a Java Archive (JAR) file.

We need to enable methods within the `CalculatorService` class to be accessible as Web Services, in this case using WebSphere Studio Application Developer. The `CalculatorService` class is contained in the `Calculator.jar` file. This could either be imported into WebSphere Studio Application Developer as a JAR (or ZIP) file, or added as a reference from the Java build path of a separate J2EE project.

In our example, we will create a new Web project (analogous to a J2EE Web Module WAR) to act as our front-end to the code within the JAR file. We will reference the JAR file from our Web project by adding it to the Java build path and making the JAR file available to the application at runtime. In our example, we have no need to edit the existing back-end code contained within that JAR file, so there is no benefit to importing it into the WebSphere Studio Application Developer tooling; we simply want to enable certain methods within the class to be available as a Web Services. The approach we use here would also be applicable if we did not have the source code available, since we never actually see or use the source code itself.

In order to Web Service enable the `ICalculator2 add(float, float)` method, the code needs to be contained in a Web project. Create a new Dynamic Web Project under the the `SimplisticWS2NETCalculator` enterprise application, called `Calculator2WebService` which will act as a façade to the actual `CalculatorService` class.

Add the existing code to the Java build path of this project. Right-click **Calculator2WebService**, then select **Properties -> Java Build Path, Projects** tab, and check the **CalculatorCode** project. Open the `SimplisticWS2NETCalculator` enterprise application descriptor (`application.xml`); switch to the Module tab and add the `CalculatorCode` project as a Project Utility JAR. Select **Java JAR Dependencies** on the Properties panel and check the **CalculatorCode.jar** file. This will make the compiled classes within the JAR file accessible to any code being written in the `Calculator2WebService` project.

Within this Web project, write a method `add(float, float)`, which in turn calls the actual `add(float, float)` method in the `CalculatorService` class of the `Calculator.jar` file, as shown below. It is this façade which we turn into a Web Service. Create a new Java class under the `Calculator2WebService/Java Source` folder, named `Calculator2WebServiceMethod`, package: `redbook.coex.sall.business`.

Example 7-1 The façade to the CalculatorService add(float, float) method

```
package redbook.coex.sall.business;

import redbook.coex.sall.business.*;

public class Calculator2WebServiceMethod {

    //the webservice method for ICalculator2
    public float add(float arg1, float arg2) {
        float result = 0;

        ICalculator2 calc = new CalculatorService();
        result = calc.add(arg1, arg2);
        return result;
    }
}
```

The Calculator2WebServiceMethod creates an instance of the CalculatorService class, and invokes its add(float, float) method, returning the float result. We are now ready to turn this method into a Web Service.

There is a wizard in WebSphere Studio Application Developer which will guide the user through the Web Service creation. This is started by first selecting the class containing the methods you wish to turn into a Web Service (**Calculator2WebServiceMethod**), and selecting **File -> New -> Other -> Web Services -> Web Service**. Accept all defaults on the Web Service wizard until you reach the window entitled “Web Service Java Bean Identity”. On this page, the user can select which methods within the chosen class they would like to enable as Web Services. In this case, we only have one method - the add(float, float) method which is the façade to the existing back-end code implementing the ICalculator2 interface. Select this method if it is not selected automatically.

Also on this page of the Web Service wizard, there is an option to select the *Style and Use* of the Web Service. Here, in order to create an RPC style Web Service, select **RPC/Encoded**. RPC/Literal Web Service calls are not supported by Microsoft .NET, so for the sake of interoperability, which is the purpose of this scenario, RPC/Encoded was the only suitable option.

Note: The wizard will warn you that the Web Service may not comply with WS-I because the RPC/Encoded style was selected. Ignore the warning.

The wizard, once finished, will automatically generate all necessary files and code for the selected class to be a complete Web Service, including the WSDL file. A WSDL file describes everything a client needs to know about the Web Service in order to call the service, including ports, transport style (RPC in this

case), methods and their required arguments, and the location of the Web Service. For more information about WSDL, see “WSDL” on page 430. The WSDL describing our example Web Service is shown below.

Example 7-2 WSDL for the Calculator2 Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://business.sall.coex.redbook"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://business.sall.coex.redbook"
xmlns:intf="http://business.sall.coex.redbook"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types/>
  <wsdl:message name="addResponse">
    <wsdl:part name="addReturn" type="xsd:float"/>
  </wsdl:message>
  <wsdl:message name="addRequest">
    <wsdl:part name="arg1" type="xsd:float"/>
    <wsdl:part name="arg2" type="xsd:float"/>
  </wsdl:message>
  <wsdl:portType name="Calculator2WebServiceMethod">
    <wsdl:operation name="add" parameterOrder="arg1 arg2">
      <wsdl:input message="intf:addRequest" name="addRequest"/>
      <wsdl:output message="intf:addResponse" name="addResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="Calculator2WebServiceMethodSoapBinding"
type="intf:Calculator2WebServiceMethod">
    <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="add">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="addRequest">
        <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://business.sall.coex.redbook" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="addResponse">
        <wsdlsoap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://business.sall.coex.redbook" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Calculator2WebServiceMethodService">
```

```
<wsdl:port binding="intf:Calculator2WebServiceMethodSoapBinding"
name="Calculator2WebServiceMethod">
  <wsdlsoap:address
location="http://localhost:9080/Calculator2WebService/services/Calculator2WebSe
rvicemethod"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

You can see by the following line that this Web Service does indeed use the RPC style of communication:

```
<wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
```

During Web Service creation, WebSphere Studio Application Developer also creates a new deployment descriptor specifically for the Web Services deployment information, called `webservices.xml`. With this file, as well as the WAR Module deployment descriptor (`web.xml`), this Web project can now be exported as a WAR file, or as part of a J2EE Application Project (EAR file) and be ready for deployment to a runtime server such as WebSphere Application Server.

Complex data type Web Services

Although the example used in this chapter is a very basic Web Service using simple data types, complex data types are also supported in Web Services. Complex data types are represented in both WSDL and SOAP messages as a combination of simple data types, and hence can be handled in the same way as the simple data types. The only real difference is the need for serializers and deserializers to construct the complex object from the SOAP message and vice versa, but this is all generated automatically by WebSphere Studio Application Developer, and handled automatically at runtime by the application server, such as WebSphere Application Server.

As such, the process for creating the server side Web Service for code which involves complex data types is exactly the same as described above, and there would be no benefit gained from implementing a complex data type solution in this book.

Scenario implementation with the Document style model

We will use the `ICalculator3.add(ICalculator3Args args)` method to demonstrate Web Services interoperability using the Document style model of communication.

This is done in an almost identical way to the RPC style Web Service implemented above. Create a new Dynamic Web Project in the

SimplisticWS2NETCalculator EAR file, called Calculator3WebService, which will act as a façade to the actual CalculatorService class.

In order to create the Web Service, we again create a façade class which calls the existing CalculatorService class in the Calculator.jar file. This time, we want to create a Web Service front end to enable the add(ICalculator3Args) method to be accessible from a Web Service. Follow the steps from the RPC model sample in order to add the CalculatorCode.jar to the new Web project class path.

A way to do this would be to implement the front-end method add(ICalculator3Args) to invoke the existing back-end code. However, Web Services cannot pass interfaces as arguments, since this would assume a valid implementation of ICalculator3Args on any service consumer wanting to use the Web Service. This would be bad practice, since it restricts interoperability and relies on consumers having knowledge of the service implementation, which is against the principles of Web Services.

Instead, the Calculator3WebServiceMethod class (which is the façade to the existing business-layer code) will take in an array of float data types as its parameters, and within the method it will convert the array into an instance of Calculator3Args (which implements the ICalculator3Args interface), which can then be used to call the existing back-end method add(ICalculator3Args). This helps to hide the specifics of the service implementation from any clients using the service, but maintains its purpose of enabling the back-end ICalculator3 method as a Web Service. Create the Calculator3WebServiceMethod Java class similar to the previous method implementation (Calculator2WebServiceMethod).

Example 7-3 The façade to the CalculatorService add(ICalculator3Args) method

```
package redbook.coex.sall.business;

import redbook.coex.sall.business.*;

public class Calculator3WebServiceMethod {
    //the webservice method for ICalculator3
    public float add(float[] floatArray) {
        float result = 0;
        ICalculator3Args c3Args = new Calculator3Args();
        for (int i=0; i<floatArray.length; i++) {
            c3Args.setArg(floatArray[i]);
        }
        ICalculator3 calc = new CalculatorService();
        result = calc.add(c3Args);
        return result;
    }
}
```

Example 7-3 on page 310 shows the code for the `Calculator3WebServiceMethod` class. It takes in a float array, converts the array into an instance of `ICalculator3Args` and passes it as an argument to the `add(ICalculator3Args)` method of the existing back-end `ICalculator3` implementation code. The result from the back-end code is then returned by the method.

It is this class which we want to enable as a Web Service. This is done in the same way as described previously for the RPC style communication, but you must select **Document/Literal** as the *Style and Use* of the Web Service method in the wizard (instead of `RPC/Encoded`).

The Web Service wizard will (as with the RPC Web Service) generate all necessary files for the Web Service. The WSDL for the Document/Literal style Web Service is very similar to the one shown above for the RPC Web Service, but the main difference is shown below:

```
<wsdl:soap:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http"/>
```

The Document style Web Service is now complete, and the WAR file `Calculator3WebService` is now ready for deployment to a runtime server such as WebSphere Application Server. Since we will ultimately be generating service consumers to both Web Services (the RPC-style `ICalculator2` and the Document-style `ICalculator3` services), it makes sense to deploy the EAR file containing both Web Service WAR projects.

Deployment

In order to make the Web Services available to any service consumers (clients), they will be deployed to WebSphere Application Server.

In order to do this, you must first export the EAR file from WebSphere Studio Application Developer. Select the **SimplisticWS2NETCalculator** project, then select **File -> Export -> EAR** file from the menu. Follow the wizard and save the EAR file to a suitable location.

Install the EAR file into WebSphere Application Server using the method of your choice, and start the application. If you need help with application deployment, refer to the WebSphere InfoCenter.

You must also make the `Calculator.jar` file available to the server at runtime. One way to do this is to copy the `Calculator.jar` file into the `<WebSphere_root>/AppServer/lib` directory.

If installation is successful, the WSDL files associated with Calculator2WebService and Calculator3WebService should be accessible at:

```
http://<machine_name>:9080/Calculator2WebService/wsdl/redbook/coex/sa11/business/Calculator2WebServiceMethod.wsdl
```

and

```
http://<machine_name>:9080/Calculator3WebService/wsdl/redbook/coex/sa11/business/Calculator3WebServiceMethod.wsdl
```

These WSDL files will be needed by any consumer wishing to develop code to access the services. With access to the WSDL files, and with the services running, we are now ready to develop clients for the service consumer to invoke the Web Services, and ultimately invoke the back-end Calculator operations.

Service provider considerations

There are important considerations when designing the Web Service with regard to best practice application design and increasing qualities of service.

In the WebSphere Application Server environment, there are two types of Web Service endpoints which can be implemented. These are the standard Java Bean Web Service (as implemented for the simple solution) and an EJB Web Service. There are guidelines for deciding which one to use. If the business logic of the service is completely contained in the Presentation layer, it is necessary to use a standard Java Bean Web Service. If the business logic of the service is completely contained in the Business layer, using an EJB Web Service will provide the best practice solution.

If the service application has some processing in both layers of the application, then the following should be considered.

- ▶ If in the existing application (which is being enabled as a Web Service), there is some preprocessing of the request in the Presentation layer prior to calling the business logic in the Business layer, then it is good practice to expose the existing application interface as a Web Service in the Presentation layer. This would require the implementation of the Web Service to be a standard Java Bean Web Service. This ensures that the preprocessing of the request will still take place, with no need to modify the business layer interface.
- ▶ If, however, there is no preprocessing of the request in the existing application Presentation layer then the best practice will be to expose the interface as a Web Service in the Business layer as an EJB Web Service.

Implementing an EJB Web Service is an easy way to add complex qualities of service to a Web Service endpoint. EJB Web Services inherit their qualities from the EJB container in which they reside. For example, an EJB Web Service will have all concurrent client access issues handled automatically by its EJB

container, whereas a standard Java bean Web Services would have to specifically consider how to handle such events.

Transactions are another example. The EJB endpoint runs in the transaction context of its EJB container, and thus the service's business logic can also run under the transaction context as defined by the EJB container. If transactions were required using a standard Java bean Web Service, the transaction handling logic would have to be implemented manually. Therefore, a heavily transactional Web Service would be better designed using EJBs and exposing the Web Service through an EJB Web Service endpoint.

Creating an EJB Web Service using WebSphere Studio Application Developer is very straightforward, and a wizard is provided to help guide the user through the process.

7.2.3 Service consumer

This section explains how to consume the WebSphere Web Service from a Microsoft .NET client. Before creating the service consumer, it is assumed that the WebSphere Web Service has been written and deployed on a server (as described above) and the Web Service is accessible to the .NET application.

Various scenarios for consuming a WebSphere Web Service are discussed later in this chapter. As the process for consuming Web Services in Microsoft .NET is the same in all scenarios, we will demonstrate the code for only one scenario, as illustrated in Figure 7-10.

Creating the service consumer

This section describes how to use a WebSphere-generated Web Service using .NET middleware, and to describe this, a sample application is created which returns the average of float numbers. The Windows client application (msClient) is created to accept data from the user, and the middleware logic (bizAverager) is created to perform the business logic of the application, including consuming the Web Service created in the previous section (which adds numbers together), using this result to calculate and return the average of the numbers.

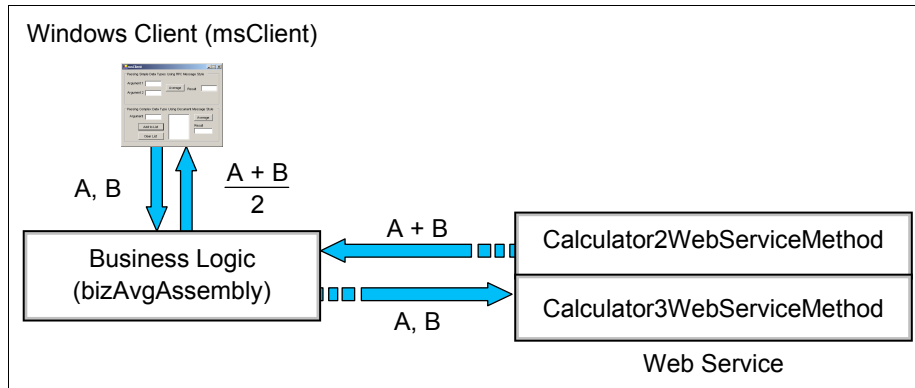


Figure 7-3 Averager application

To develop the client, we will be using Microsoft Visual Studio .NET. Without a development environment, such as Visual Studio, the necessary code can be developed using the .NET Framework V1.1 SDK and a simple text editor.

Start Visual Studio and create the business logic code first, and then the Windows client. To create business or middleware logic, create a new Visual C# Projects/Class Library project `bizAvgAssembly` and rename the `Class1.cs` class as `bizAverager.cs` class, make sure you rename the class and the constructor method also.

Next, add the Web Services reference which the application needs to consume. Right-click the **Web References** tag in the solution explorer then select **Add Web Reference**. The Add Web Reference window appears as shown in Example 7-4 on page 315.

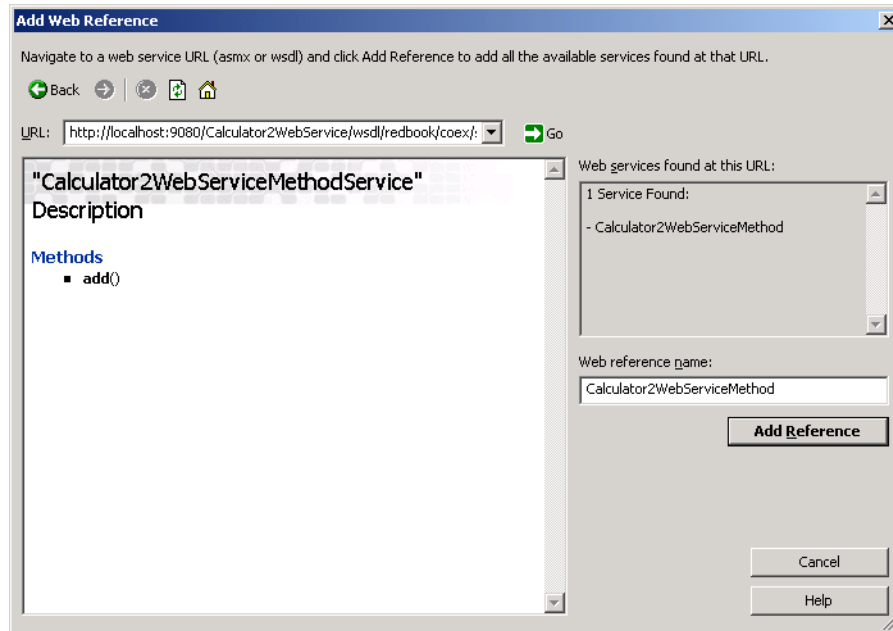


Figure 7-4 Adding a Web reference

In our sample application, we are using two Web Services, the Calculator2WebService for adding two float numbers using RPC style communication and Calculator3WebService for adding an array of float numbers in Document style communication. The WSDL URLs for Calculator2WebService and Calculator3WebService are:

```
http://<machine_name>:9080/Calculator2WebService/wsdl/redbook/coex/sa11/business/Calculator2WebServiceMethod.wsdl
```

```
http://<machine_name>:9080/Calculator3WebService/wsdl/redbook/coex/sa11/business/Calculator3WebServiceMethod.wsdl
```

In .NET, the code for creating RPC style or Document style Web Service *clients* is the same. When you add the Web reference, .NET automatically determines the messaging style and accordingly creates a proxy for accessing the Web Service.

Scenario implementation with the RPC model

To use the WebSphere Web Service from a .NET client consumer, first enter the WSDL path of the Web Service from the Add Web Reference window shown above and click **Go**. This will search and display any methods available for that

Web Service. By clicking the **Add Reference** button, add the service reference to your project.

Add the following method (getAverage) into the bizAverager class, and build the project to create the bizAverager.dll.

Example 7-4 bizAvgAssembly implementation (RPC)

```
public class bizAverager {
//...
    public float getAverage(float arg1, float arg2) {
        //Create Web Service object
        Calculator2WebServiceMethod.Calculator2WebServiceMethodService
rpcSimpleWS= new Calculator2WebServiceMethod.Calculator2WebServiceMethodService
();
        //Request service to add 2 numbers
        float result = (rpcSimpleWS.add (arg1,arg2));
        //Return Average
        return result/2;
    }
//...
```

Scenario implementation with the document model

Next, to implement the scenario using Document style messaging communication, add the following code. The code uses an array of floats for passing the parameters to the Web Service.

Example 7-5 bizAvgAssembly implementation (Document style)

```
...
    //This method uses complex data type-an array of float to invoke Web
    //Service on WebSphere using Document
    //This method takes float array as a parameter and requests Web Service
    //on WebSphere to add it.
    //The result returned is average of all float parameters.
    public float getAverage(float[] args) {
        //Create Web Service object
        Calculator3WebServiceMethod.Calculator3WebServiceMethodService
docComplexWS=new Calculator3WebServiceMethod.Calculator3WebServiceMethodService
();
        //Request service to add float numbers in an array
        float result = (docComplexWS.add (args));
        //Return Average
        return result/args.Length;
    }
...

```

Once you are done with coding the library, click **Build -> Build bizAvgAssembly** from the menu to see if the build was successful. If you encounter any error, check the source again.

Creating a client application

To demonstrate the functionality of the Web Service, let's create a Windows client as shown in Figure 7-5; the Windows client is used to accept the data from the user and to display the average of the numbers.

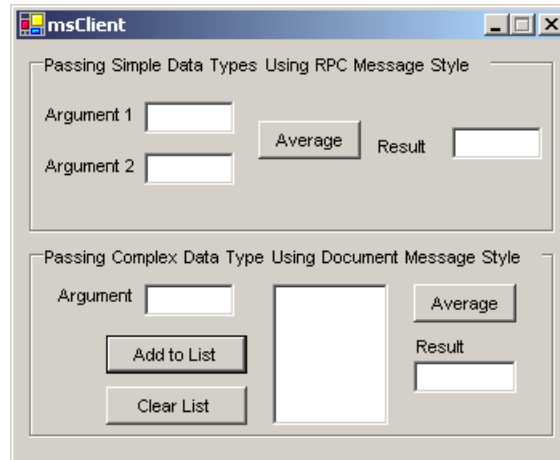


Figure 7-5 .NET client for consuming WebSphere Web Service

First, add the Windows Application project to an existing Class Library project, as shown in Figure 7-6.

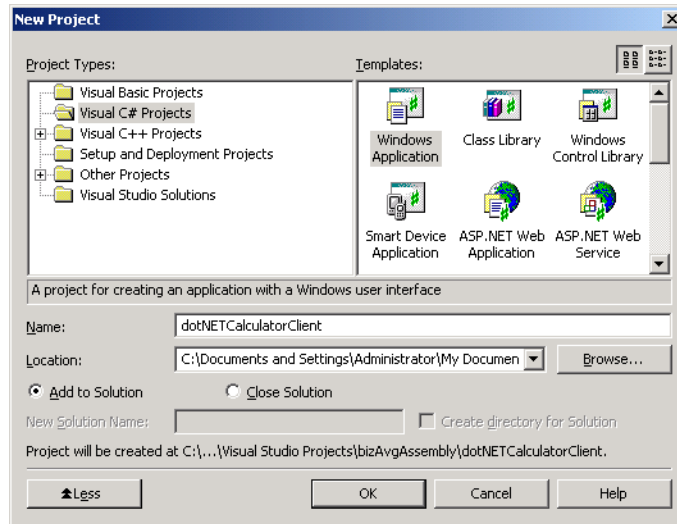


Figure 7-6 Add new Windows Application project

Open Windows Form and create GUI interfaces as shown in Figure 7-5. You can also use the existing client code provided with the book.

The new client application does not know about the bizAvgAssembly yet; you need to add a reference to the library.

The application is divided into two groups; one demonstrates how to call a Web Service using RPC style communication, and the other how to call a Web Service using Document style communication.

For the RPC style Web Service, the client passes two float arguments entered by the user and calls Calculator2WebService to add the two numbers together. For the Document style Web Service, the client passes a float array, a list of arguments entered by the user and calls the Calculator3WebService method.

To add code into the client for consuming the business logic functionality, let's start with the RPC style client. Double-click the **Average** button (the upper button) to open the code window containing the click button event declaration. Add the code as shown below.

Example 7-6 Button click event implementation

```
private void button1_Click(object sender, System.EventArgs e) {
    //Create business object
    bizAvgAssembly.bizAverager bz= new bizAvgAssembly.bizAverager();
    //invoke the method and display average
}
```



```
        textBox3.Text =bz.getAverage (Convert.ToSingle(textBox1.Text),
        Convert.ToSingle(textBox2.Text)).ToString();
    }
}
```

Follow the same steps for creating a Web Service which uses Document style communication. The code is shown below.

Example 7-7 Button click event implementation

```
private void button3_Click(object sender, System.EventArgs e) {
    //create float array
    float[] floatArray = new float[listBox1.Items.Count];

    //fill the array
    for (int i=0; i<listBox1.Items.Count; i++)
        floatArray.SetValue (Convert.ToSingle(listBox1.Items[i]),i);

    //Create business object
    bizAvgAssembly.bizAverager bz= new bizAvgAssembly.bizAverager();
    //invoke the method and display average
    textBox5.Text =bz.getAverage(floatArray).ToString ();

    //cleanup
    listBox1.Items.Clear();
    textBox4.Clear ();
    textBox4.Focus ();
}
```

Note: There is some extra coding for the client, for example an Add to list button, but we will not go into details about it here. You can find more details about the supporting code for the client in the source code.

The coding is now complete. You should now be able to build and run the .NET consumer project to invoke and use WebSphere Web Services in a synchronous, stateless manner as required.

Test

In order to test the sample, make sure that the WebSphere application is running either in the WebSphere runtime environment or in the WebSphere Studio test environment.

Run the .NET client application and provide some random data for both calculators (two numbers, list of numbers). Clicking the **Average** button should bring up the results on the client.

Design considerations for the service consumer

This section discusses various considerations while designing the .NET Web Service consumer application.

► Error handling

Use proper structured exception handling while developing the service consumer. There are some general situations where we have to take special care while writing the code. These are:

- The deployed Web Service server is not available or is down.
- Timeouts caused by heavy traffic on server.
- The Web Service generated errors.

► Use of configuration files in Web Service clients

For better portability, a configuration file can be used to configure information about the Web Service. For example, you can use a configuration file for specifying the Web Service URL for the client proxy so that code will automatically handle situations where the URL is not fixed. This situation could occur when deploying a Web Service from a development to a production environment. Note that while compiling the application, the automatically generated proxy can overwrite the code written by you. You can also generate a custom proxy by using the WSDL.EXE tool provided by .NET. Note that regeneration of the proxy can overwrite the manual code.

When consuming a Web Service, make sure you have the correct URL in the WSDL. For example, when you move the Web Service from a development environment to a production environment, make sure the proper path for the URL location is in the WSDL.

► Type handling

Take care while consuming Web Services on different platforms. The requester and sender data types should match to avoid errors.

Table 7-1 shows Java and .Net type compatibility.

Table 7-1 Java and C# type compatibility

Java type	C# type
boolean	Boolean(boolean)
java.lang.Short	Byte(byte)
java.lang.Integer	Char(char)
java.math.BigDecimal	Decimal(decimal)
double	Double(double)

Java type	C# type
float	Single(float)
int	Int32(int)
long	Int64(long)
byte	SByte(sbyte)
short	Int16(short)
java.lang.Long	UInt32(uint)
java.math.BigInteger	UInt64(ulong)
java.lang.Integer	UInt16(ushort)

7.3 Extended solution

In this section, we will discuss various extended solutions and issues for consuming a WebSphere Web Service in .NET.

Because Web Services are accessible using URLs, HTTP, and XML, they can be consumed on any platform and any layer. The WebSphere Web Service can be consumed from different tiers of the five-tier architecture model in .NET, depending on the client. For example, a client can be an ASP.NET client (running in the Client tier), Windows Application client (running in the Presentation tier), Class Library for business logic in middleware or even a Web Service.

Consuming a Web Service in a Windows .NET client

Consuming a Web Service from a .NET Windows application is fairly straightforward and easy. To consume a Web Service in a .NET Windows client, simply add a Web reference to the client application and access the services by creating an instance of the Web Service as discussed in “Creating the service consumer” on page 313. The proxy is created for the client after adding the Web reference, or you can generate your own proxy to communicate with the Web Service.

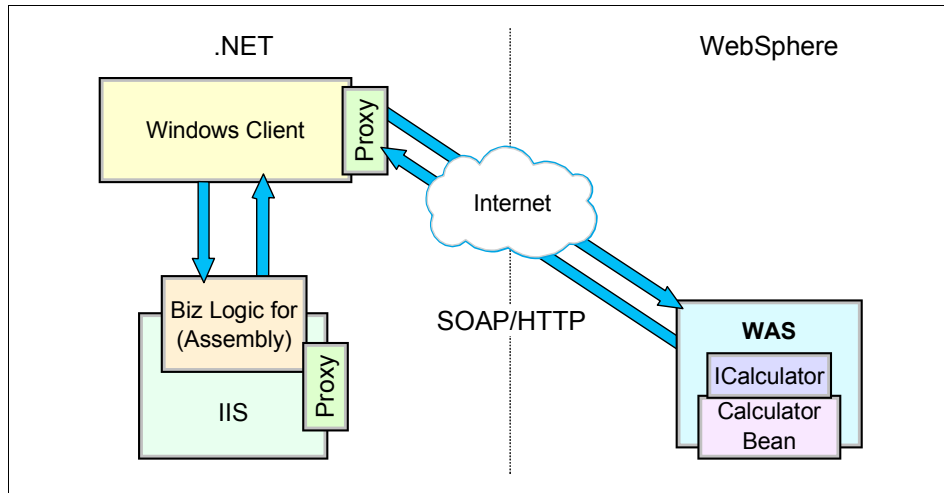


Figure 7-7 Accessing a Web Service in Windows application

The proxy on the client side communicates with the Web Service using the SOAP/HTTP protocol and the WSDL is used for finding the service description. Note that the client-side code and procedure is the same for both RPC and Document style format.

Consuming a Web Service in ASP.NET

This scenario is illustrated in the following diagram. Although the architecture in the diagram is different from the architecture implied in our example solution for our simple average calculator described in “Creating the service consumer” on page 313, the code and the procedure for consuming the Web Service in ASP.NET is same. The only difference is that the code for consuming a Web Service will have a Web Form instead of a Windows Form. The Web Proxy is generated on the client side when you add the Web reference.

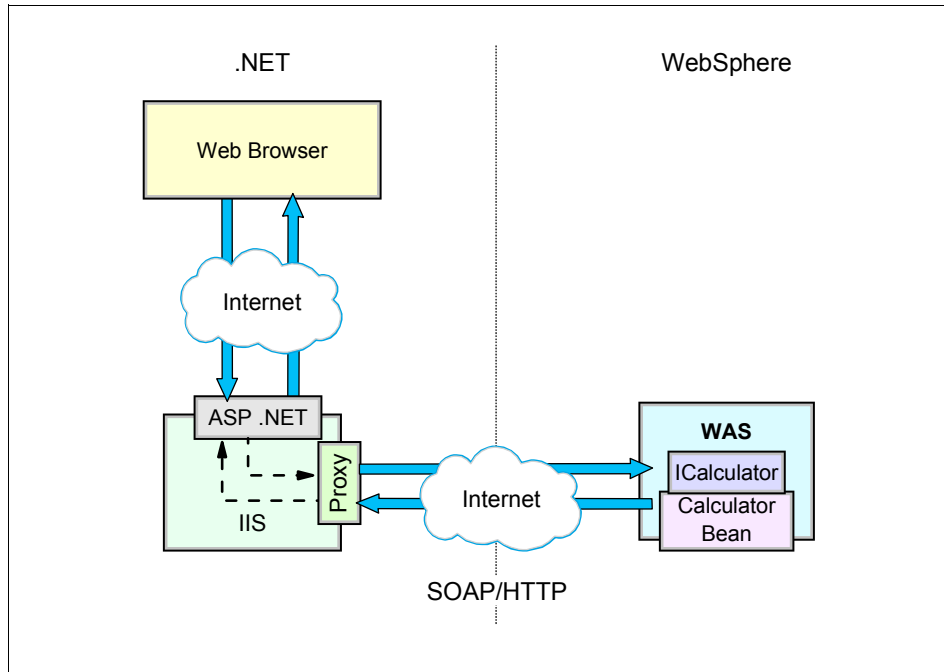


Figure 7-8 Accessing a Web Service in ASP.NET

Assuming the ASP.NET page on IIS consumes a Web Service, when the browser sends a request to the ASP.NET page, the compiled Web page invokes the Web Service through the proxy. The compiled Web page processes the result, and the IIS then returns the HTML response to the browser.

Consuming a Web Service in the business logic tier

The code and procedure for consuming a Web Service in a middleware component is the same as seen in previous samples. Figure 7-9 shows this scenario with a Windows client and a Web client.

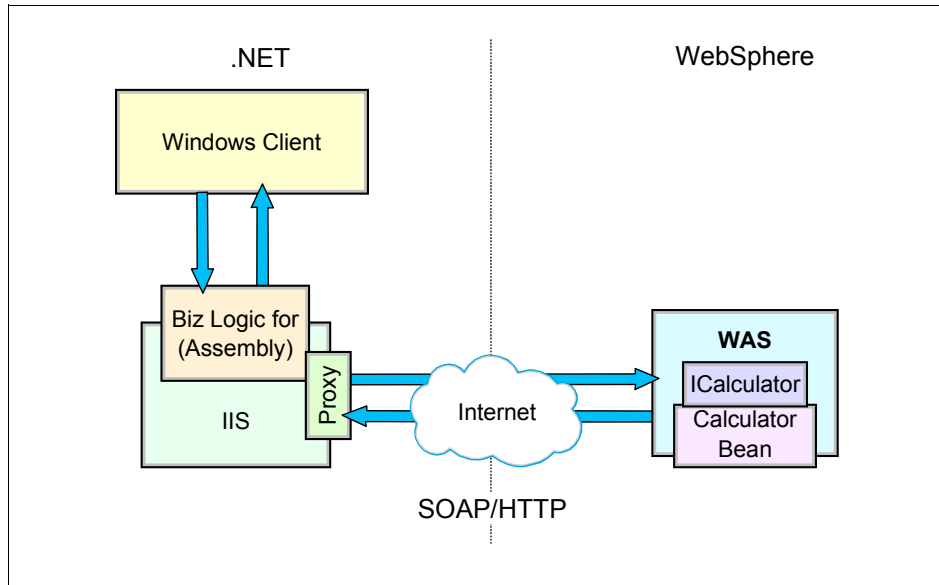


Figure 7-9 Consuming Web Service in the business logic tier using a .NET client

The Windows client uses methods of the business logic, where the business logic invokes Web Services. This hides the actual implementation of the Web Service from the client. This scenario can be used with COM+ applications which reside on the server side of the .NET application. In such cases, the assembly in the COM+ application will invoke the Web Service.

Assuming the business logic and client are distributed in different locations, the business logic can be accessed using .NET remoting from the client side.

The same scenario with a Web application using a browser client is shown in Figure 7-10.

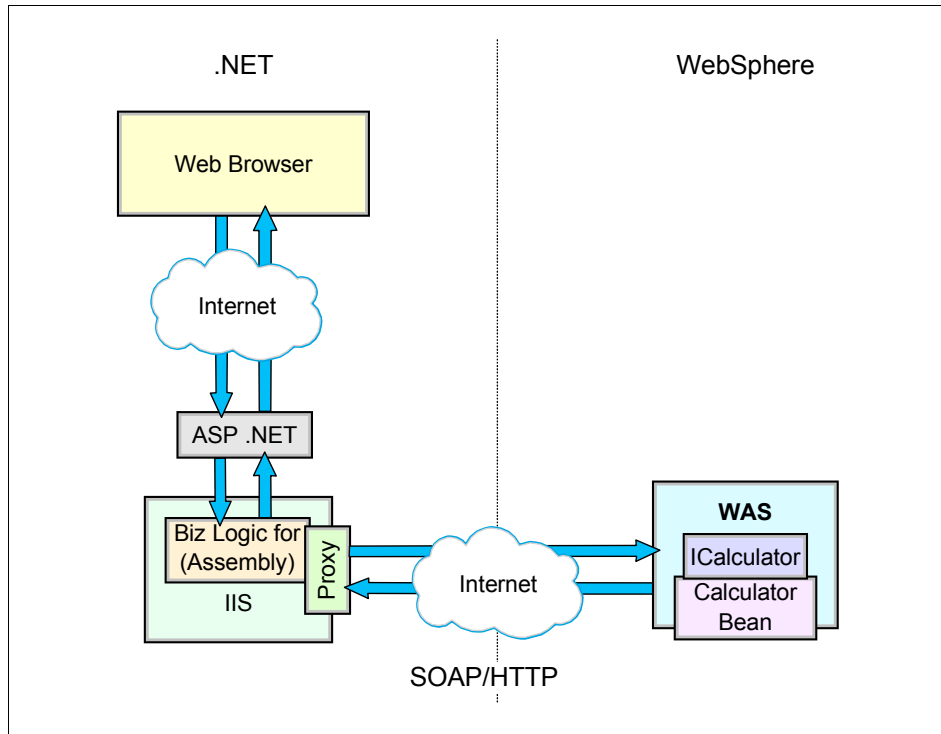


Figure 7-10 Consuming a Web Service in middleware using ASP.NET application

In this scenario, the compiled ASP.NET Web page uses a service from the business logic, and the business logic invokes a Web Service. Here, we assume that both the ASP.NET application and the business logic are the same location, but for scalability this can be separated in different tiers, one for the ASP.NET/IIS and another for the business logic. In such cases, the business logic can be mounted onto a COM+ application. The COM+ provides necessary adaptors as a component service. Note that the proxy is required at the business tier to access the Web Service.

Users could also design a .NET console application, a Windows service or a mobile application to access a Web Service.

Generating your own proxy on the client side

When you add a Web Service to your project, the Visual Studio IDE takes care of generating a proxy for the client side, but you can also write and compile your own proxy code. .NET provides a tool, WSDL.EXE, to generate the proxy from the provided WSDL. You need to compile the proxy code when you write or generate your own proxy. You can insert your own code into an existing proxy

code to make it more flexible. For example, the WSDL URL in the proxy class can be accessed from a configuration file. This will avoid recompilation of the Web Service client when the Web Service is moved from one server to another.

7.4 Recommendations

This section covers recommendations based on issues discovered in the implementation of our interoperability solution.

Note: This section is about recommendations regarding interoperability of the Web Services. It does not cover recommendations about J2EE best practices or application design strategies.

Data types

When designing your Web Services, you should be careful not to pass language-specific data types as arguments to your Web Service methods. When writing the Web Services, one cannot be certain what consumers will use the Web Service in future, and on what platform they will be implemented. For example, if a Java Web Service took in a `java.util.Vector` as a parameter, a Microsoft .NET service consumer written in C# would have no concept of what a `java.util.Vector` is and would not be able to interoperate with the service.

It is also not possible to pass a Java interface as a parameter of a Web Service method since the service consumer will not have an implementation of the interface, and the interface definition will be meaningless to the consumer.

A way to get around this is to implement the Web Service with simple parameters which can be mapped using an xsd schema (such as an array of simple data types), and implement a wrapper class which converts the language-specific type (or interface) into the simple data type and vice versa (see “Scenario implementation with the Document style model” on page 351 for an implementation of this approach).

Data types which can be used (on the WebSphere side) and still be interoperable include:


- ▶ Java primitive types: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double` (along with their corresponding wrapper Java classes)
- ▶ Standard Java classes: `String`, `Date`, `Calendar`, `BigInteger`, `BigDecimal`, `QName` and `URI`
- ▶ Java arrays (of supported types)
- ▶ User-defined Java classes (made up of supported types)

Note: When using such user-defined Java classes, use the get and set methods to help ensure portability and interoperability of the Web Service.

RPC style versus Document style

There are occasions when RPC style Web Services are a better option than Document style Web Services, and vice versa. Both RPC/Encoded and Document/Literal Web are supported by WebSphere and .NET, so interoperability between them is not a problem.

However, the Web Services Interoperability Organization (WS-I) specification 1.0, to which J2EE 1.4 conforms, recommends that encoded bindings not be used, which suggests that for better interoperability, Literal data formatting should be used in the SOAP messages. This then leads to the issue that Microsoft .NET does not support RPC/Literal SOAP messages, which would ultimately suggest that for ideal interoperability (ignoring performance issues), Document/Literal SOAP formatting is the best choice in all occasions.



Scenario: Synchronous stateless (WebSphere consumer and .NET producer)

This chapter is the second part of the synchronous stateless discussion. As opposed to Chapter 7, “Scenario: Synchronous stateless (WebSphere producer and .NET consumer)” on page 297, here the roles are changed and WebSphere is the service consumer and .NET is the service provider.

This chapter does not repeat the problem definition; for that, please refer to 7.1, “Problem definition” on page 299.

8.1 Solution model

This section discusses the design and implementation of a solution for the stateless synchronous interaction from application artifacts in a WebSphere Application Server application to application artifacts in a Microsoft .NET application.

The solution provided in this section restricts itself to providing a simplified implementation that illustrates how interoperability can be achieved between WebSphere Application Server and Microsoft .NET business layer components.

This section contains the following subsections:

- ▶ A solution to the problem

Here we illustrate our solution design, including the rationale for some of the design decisions taken. We then continue by presenting the details of the implementation.

- ▶ Scenario implementation for the service provider

In this section, we describe how to implement the service provider part of our scenario, using both Procedure Call (RPC) and Document style paradigms.

- ▶ Scenario implementation for the service consumer

Here we describe an implementation for the service consumer side of our example scenario.

Our solution to the problem is based upon a simple calculator application. The application provides a very limited set of functionality. The Calculator service provides only the ability to perform a simple addition of two numbers or addition of a list of numbers, and return the results. The reason for this is to keep the example as simple as possible, so that the objective of how to integrate WebSphere Application Server and Microsoft .NET application artifacts is not obscured by the complexities of our chosen scenario.

8.1.1 A solution to the problem

This scenario concerns the situation where there exists some code in the Business layer of a Microsoft .NET environment which we need to access and interoperate from a WebSphere Application Server environment.

In this section, we describe a basic implementation of a solution, which is intended to highlight the process of achieving the interoperability. We shall consider other implementation options in the extended solution model section later on. In describing our implemented solution to the problem, we shall detail necessary steps for both the service provider side and the service consumer.

8.1.2 Service provider

A Web Service provides the infrastructure to connect to other applications via various Web protocols and messaging formats such as HTTP, XML, and SOAP. In this section, we are going to discuss how to create a Web Service in .NET with various messaging formats. We will demonstrate how to use Web Services for both RPC-style and document-style communication. To demonstrate this, we will create a sample Web Service application.

Creating the Web Service in .NET

To illustrate the steps involved in creating a Web Service in .NET, we created two different Web Services. The first one shows:

- ▶ How to pass a simple data type (a float)
- ▶ How to use RPC style communication in Web Services

The second one demonstrates:

- ▶ Passing an array of floats as an argument to the Web Service
- ▶ How to create document-style Web Services

To create these Web Services, we used the following technology:

- ▶ Windows 2000 Server with IIS
- ▶ Visual Studio .NET
- ▶ C# language

The Web Services in our example provide a simple Calculator function for adding the numbers. In designing our Web Services, we followed standard object-oriented programming best practices in that the sample Web Services are divided into two functional blocks, one of which performs the actual addition of numbers and the other is responsible for invoking the business logic.

The code for performing the addition is implemented in a separate assembly, `Calculator.dll`, which encapsulates and hides the actual implementation details of a class. This also helps to simplify the code, and the assembly can be reused in other scenarios.

The structure of `Calculator.dll` is illustrated in following figure.

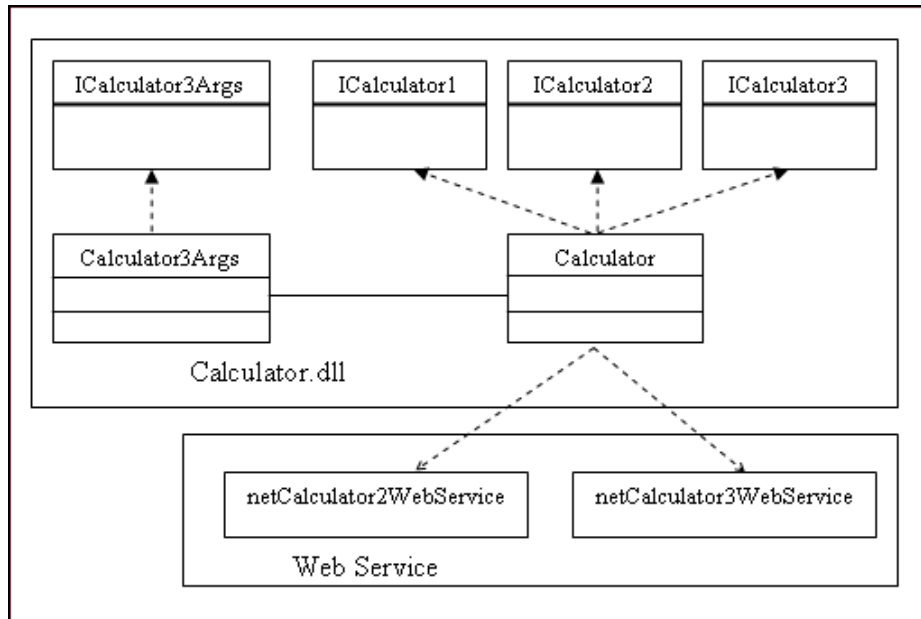


Figure 8-1 The Calculator.dll class

We will first describe how to create the Calculator.dll, and then how to enable this functionality to be accessed via Web Services.

Creating the Calculator.dll assembly

As shown in the above figure, Calculator.dll contains a Calculator class which is the main class and contains the actual implementation of the business functionality. This class implements the methods declared in three interfaces. These interfaces are:

- ▶ ICalculator1 - the methods in this interface imply stateful interaction, and as this chapter is intended to describe stateless, synchronous interoperability, ICalculator1 is not discussed further in this chapter.
- ▶ ICalculator2 - declares the method `add(float arg1, float arg2)` which implies stateless interaction, and is used in this chapter to demonstrate stateless, synchronous Web Services using RPC-style communication.
- ▶ ICalculator3 - declares the method `add(ICalculator3Args args)`, which again implies stateless interaction and is used in this chapter to demonstrate stateless, synchronous Web Services using document-style communication.

Interfaces ICalculator2 and ICalculator3 pass float arguments and a complex type, ICalculator3Args, respectively.

1. To create the assembly, open a new Class Library project in Visual Studio. Rename `Class1.cs` as `Calculator.cs` and add the classes for the interfaces. The following figure shows a snippet of the Solution Explorer with all interfaces and a class used for Calculator assembly.

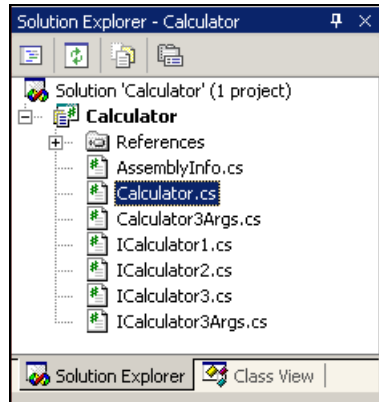


Figure 8-2 Files in Calculator Assembly

2. Open and write the code for each interface and main Calculator class as described below.
3. The `ICalculator2` interface is used for demonstrating RPC message style Web Services which take two float arguments to be added together. The implementation for `ICalculator2` consists of the following code:


```
public interface ICalculator2 {
    //declare method to add arguments
    float add(float arg1, float arg2);
}
```
4. The `ICalculator3` interface is used for demonstrating a document message style Web Service which takes a complex data type (`ICalculator3Args`). The code for `ICalculator3Args` is:


```
using System.Collections ;
public interface ICalculator3Args {
    //declare method to set argument and to get iterator
    void setArg(float arg);
    IEnumerator getIterator();
}
```
5. The `Calculator3Args` class is used to represent a complex type to hold a list of floats. This complex type is used in the `add(ICalculator3Args)` method of the `Calculator` class where all floats in the object are added together. Methods are available in the `Calculator3Args` class for adding new arguments to the object, getting the object and its values, and for getting the number of floats

currently stored in the object. The Calculator3Args class is implemented using an ArrayList data structure. The code for Calculator3Args is as follows.

Example 8-1 Calculator3Arg code

```
using System;
using System.Collections ;
public class Calculator3Args : ICalculator3Args {
    //Code to create and populate array
    ArrayList argArray;
    public Calculator3Args() {
        argArray = new ArrayList();
    }
    public void setArg(float arg) {
        argArray.Add (arg);
    }
    public IEnumerator getIterator() {
        return argArray.GetEnumerator();
    }
}
```

6. The Calculator class implements the add() methods defined in the three ICalculator interfaces, and performs the business code of adding the numbers. The code for the Calculator class is as shown below.

Example 8-2 Calculator class

```
using System;
using System.Collections ;

public class Calculator : ICalculator1, ICalculator2, ICalculator3 {
    float a1;
    float a2;
    public Calculator() {
        //super();
    }
    //Stateful Synchronous
    /* Code for Stateful Synchronous */
    //Stateless Synchronous simple data type
    public float add(float a1, float a2) {
        return a1+a2;//Return some of parameters
    }
    //Stateless Synchronous complex data type
    public float add(ICalculator3Args args) {
        float total =0;
        IEnumerator iterator = args.getIterator ();
        while (iterator.MoveNext())
        {
            //add each array element
        }
    }
}
```



```

        total += ((float)iterator.Current);
    }
    return total;//Return sum of array elements
}
}

```

7. Build the project once the coding part is over to create Calculator.dll. The following diagram shows the relation between Calculator.dll and the Web Services we will now create.

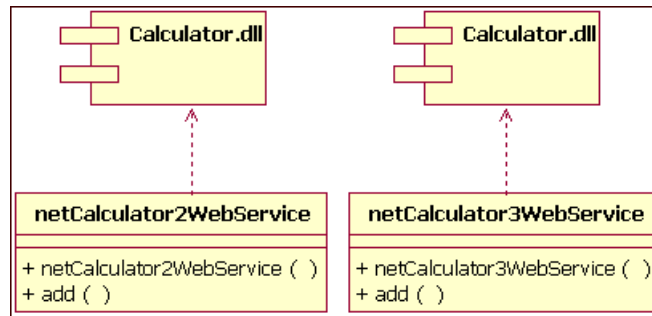


Figure 8-3 Calculator.dll Assembly and Web Services

Developing the Web Services in Microsoft .NET

Microsoft .NET supports Web Services either using RPC/encoded message style or Document/literal message style.

The Document style format indicates that the SOAP body simply contains an XML document. The literal XML schema definitions are used between provider and consumer and therefore this combination is referred to as Document/Literal.

The RPC (Remote Procedure Call) style format indicates that the SOAP body contains an XML representation of a method call such as DCOM and CORBA. The RPC style uses the names of the method and its parameters to generate structures that represent a method.

In Microsoft .NET, you can create a Web Service either using a normal text editor or using Visual Studio Integrated Development Environment (IDE). The Visual Studio IDE is recommended for increased productivity and better debugging facilities of Web Service projects.

To begin the Web Service development, open Visual Studio and select **File -> New Project** to open the New Project window as shown. Select the Project Type **Visual C# Projects** and then select **ASP.NET Web Service**.

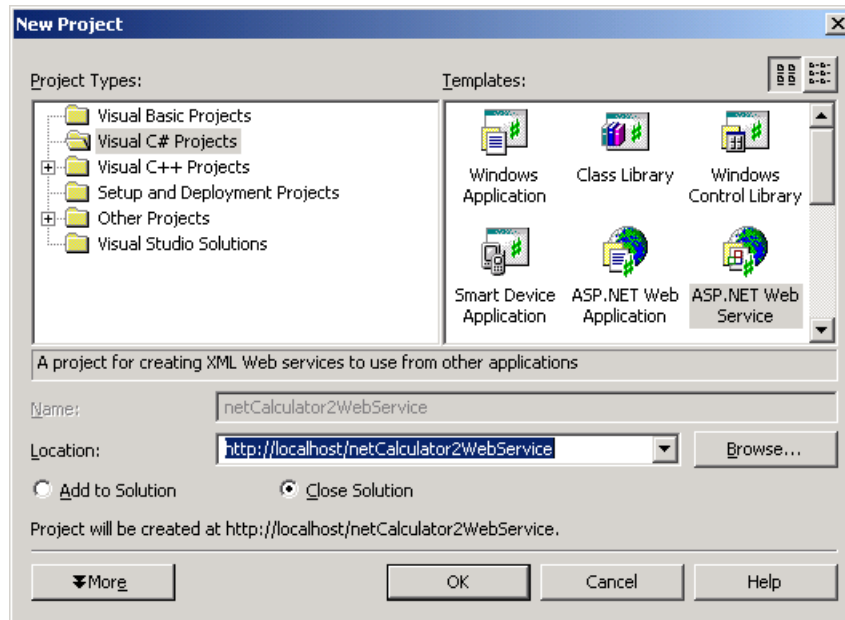


Figure 8-4 Creating New Web Service

Note that the Web Service project is created in a Web directory as well as in the Web Project Cache. The Web Project Cache is a location to store Web pages while working offline. The Web Project Cache setting can be modified by selecting **Tools -> Option** from the menu, and by selecting custom options.

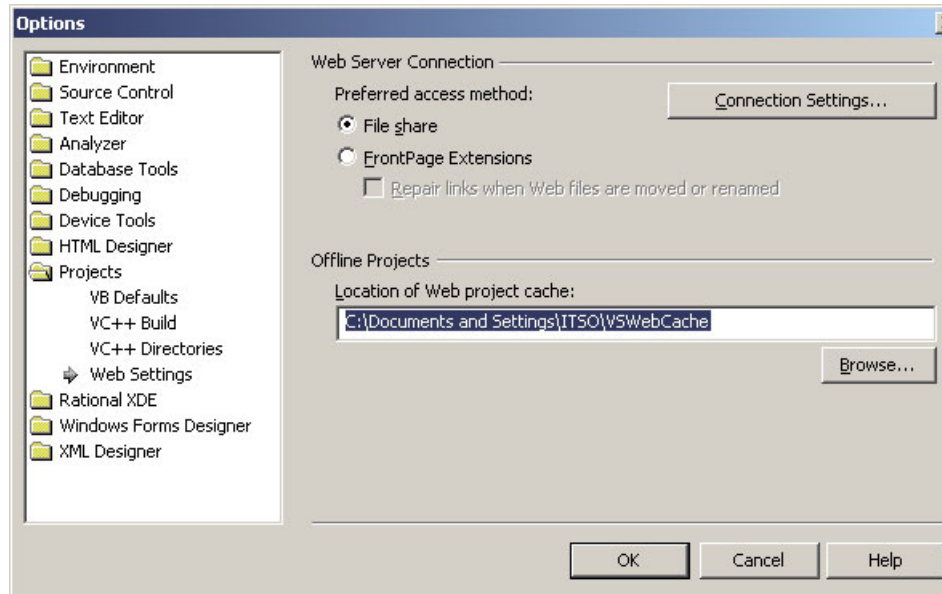


Figure 8-5 The Web Project Cache to work offline

When you create a new ASP.NET Web Service project, default files are generated into your project such as Service1.asmx (or similar), Web.config and Global.asax. The Service1.asmx file is used to define various services whereas the Web.config file is used to define application specific settings. Global.asax is an optional file that contains code for responding to application-level events.

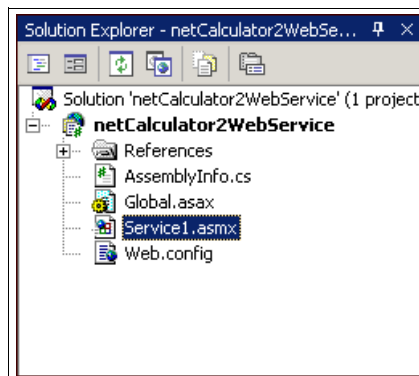


Figure 8-6 Web Service default files in Solution Explorer

The namespace for the created Web Service is the name of the project by default. Before starting the coding of the service, it is recommended that you give

a more meaningful name to Service1.asmx, such as Calculator2WebServiceMethod.asmx.

As discussed earlier, the existing back-end Calculator code is implemented in a separate assembly (or DLL). We wish to enable this back-end functionality as Web Services. In order to use the Calculator assembly in your project, add a reference to the Calculator.dll by right-clicking **References** in the Solution Explorer and selecting **Add Reference....** Browse to the path where Calculator.dll is stored and then click **OK** to add the Calculator assembly to the project.

Note: After the Calculator.dll is built, it sits under the Visual Studio project directory. You can use the project directory, or you can copy the file to a production directory.

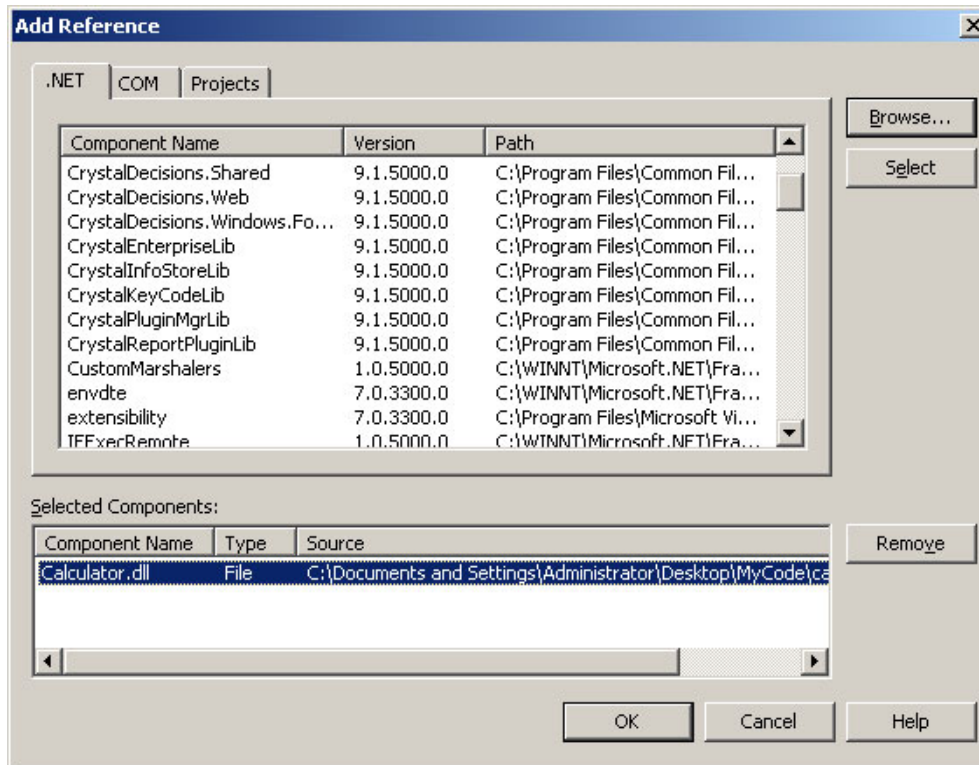


Figure 8-7 Adding a reference to the assembly

We will now describe how to implement the Web Services which will use the existing back-end functionality stored in the Calculator.dll assembly. We will first cover implementation of a Web Service using RPC message style (netCalculator2WebService) and then the document message style (netCalculator3WebService). Note that the steps are the same for creating both Web Services except for the RPC message style where we have to add an extra attribute.

Scenario implementation with the RPC model

Here we will describe how to write a Web Service which takes simple data types, using RPC style communication. Our Web Service method will then in turn call the back-end functionality contained in the Calculator.dll assembly and return the result of the addition of the two float parameters. We will demonstrate this by passing a simple float data type to the Web Service method.

We will describe how to specify that the Web Service is to use the RPC style messaging format, by using the [SoapRpcService] attribute.

The RPC style specifies that all parameters are encapsulated within a single element named after the Web Service method, and that each element within the element represents a parameter named after its respective parameter name.

Microsoft .NET provides an attribute-based mechanism for controlling the format of the XML in the SOAP message. The document message style is the default messaging format in .NET and in order to change it, you need to use either the [SoapRpcService] or the [SoapRpcMethod] attribute.

The [SoapRpcMethod] attribute is useful when you want to declare a specific method to support RPC style messaging format within a class.

For example, in the following class, the method getName() (without the [SoapRpcMethod] attribute) uses the default Document style messaging format, whereas the method getID() (with the [SoapRpcMethod] attribute) supports the RPC style messaging format.

Example 8-3 RPC style method declaration

```
class rpcDemo {
    [WebMethod]
    public string getName(int id) {
        // code comes here...
    }

    [WebMethod]
    [SoapRpcMethod]
    public int getID(string name) {
        // code comes here...
    }
}
```

The `[SoapRpcService]` attribute is useful when you wish to declare all Web methods within a class to support the RPC style messaging format as opposed to Document style messaging. In our example, we use the `[SoapRpcService]` attribute before the declaration of class.

In Visual Studio, open the `Calculator2WebServiceMethod.asmx` file; this opens the Design view. Select **Click here to switch to code view....** You will find the empty Web Service skeleton; remove the sample code from the namespace body, then insert the following method.

```
namespace netCalculator2WebService {
    //...
    [SoapRpcService]//Set message type as RPC
    public class Calculator2WebServiceMethod :
        System.Web.Services.WebService {
        //...
    }
    //...
}
```

`[SoapRpcService]` requires an additional namespace to exist, so insert the following declaration to the beginning of the file after the last using line:

```
using System.Web.Services.Protocols;
```

Declare the `add()` method which takes the float parameters. To declare `add()` as a Web Service method, the `[WebMethod]` attribute is used. Attaching the `[WebMethod]` attribute to a public method indicates that the method should be exposed as part of the Web Service. The `[WebMethod]` attribute has several properties to alter the Web Service method behavior. Some of the attribute properties are:

- **BufferResponse** - Enables the buffering of responses for a Web Service method.

- ▶ **CacheDuration** - Enables the caching of the results for a Web Service method.
- ▶ **Description** - Supplies a description for a Web Service method that will appear on the Service help page.
- ▶ **TransactionOption** - Enables the Web Service method to participate as the root object of a transaction.
- ▶ **EnableSession** - Is used to set or get session state.

The following example shows how to set properties for a Web method and implement the method body. Insert the following method under the `Calculator2WebServiceMethod` class.

```
[WebMethod( Description = "Calculator for adding numbers", BufferResponse =
true )]
public float add(float arg1, float arg2)
{
    //Create Calculator object
    ICalculator2 calc2 = new Calculator ();
    //Add parameters
    float result = calc2.add(arg1, arg2);
    //return result
    return result;
}
```

Once the coding of this method is complete, build the Web Service solution by selecting **Build -> Build Solution** from the menu.

Testing the Web Service

In .NET, testing of the Web Service can be done in two ways:

1. By using the built-in test application
2. By developing or adding a project to test it

When you run a Web Service project in Visual Studio .NET, a browser is opened to allow the developer to test the Web Service functionality. The test Web application lists all services available for that Web Service. By selecting the relevant service and entering values in the Web application, the Web Service can be tested.

For the Web Service `netCalculator2WebService`, you will see the following page with the `add()` method. You can review the description for the service by clicking the **Service Description** link.

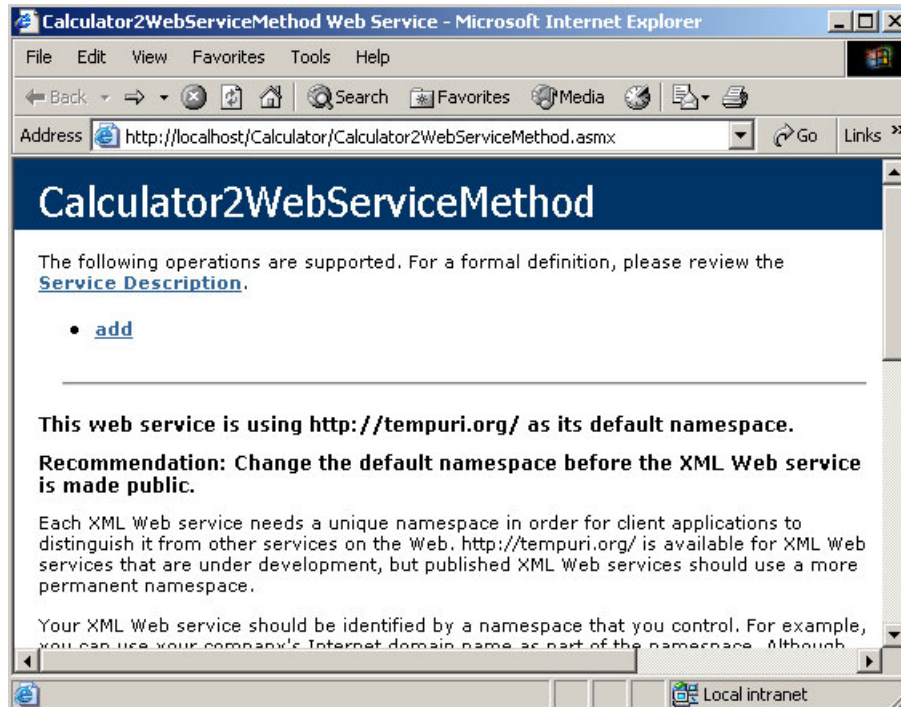


Figure 8-8 Testing Web Service -1

When you click the **add** service, the following page appears and you can test the service by inserting values and then clicking the **Invoke** button.

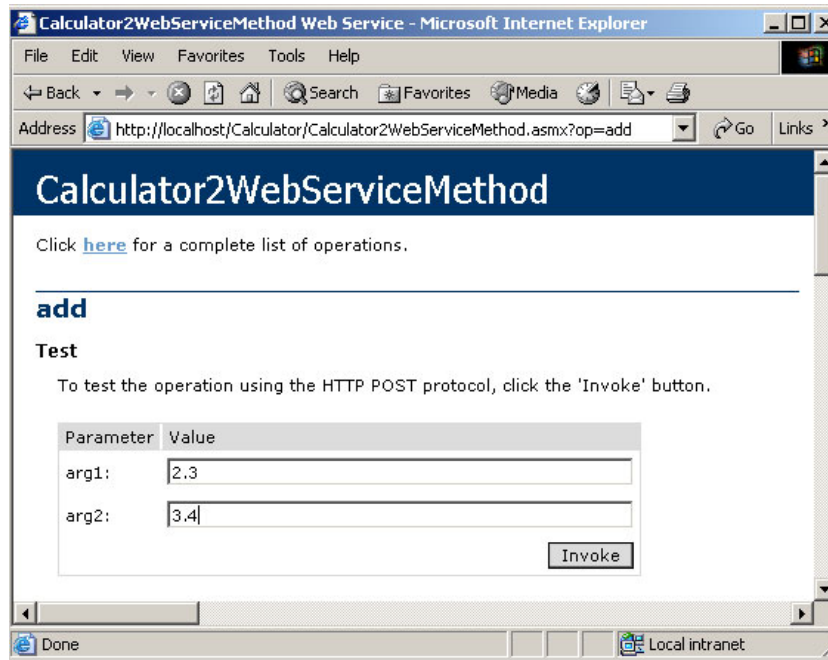


Figure 8-9 Testing Web Service methods - 2

When you invoke the method, the internal mechanism invokes the `add()` method of the Web Service. The Web Service in turn calls the existing back-end business logic stored in the `Calculator.dll` assembly which performs the actual add functionality. The result is returned in XML format to the test application and displayed in the browser as shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
<float xmlns="http://tempuri.org/literalTypes">5.7</float>
```

Figure 8-10 XML result of Calculator2webServiceMethod

A Web Service can also be tested by creating another project to act as the client to the Web Service. This can help when debugging the code. To add another project, click **File -> Add Project -> New Project** from the main menu bar. Select the type of project you want to add and then add a reference to the Web Service in the project. This is the same process as consuming a Web Service in a .NET application, described in Chapter 7, "Scenario: Synchronous stateless (WebSphere producer and .NET consumer)" on page 297.

Scenario implementation with the Document style model

Here we will describe how to write a Web Service which takes an array of simple data types, using Document style communication. Our Web Service method will then in turn call the back-end functionality contained in the Calculator.dll assembly and return the result of the addition of the array (or list) of float parameters. We will then demonstrate this using the test application provided by Visual Studio .NET.

To create a Document message style Web Service, no extra attributes are required since Document style is the default messaging style in .NET. Steps for creating the Web Service are very similar to those described in the RPC section above. Create a new C# Web Service project with the name netCalculator3WebService, add a reference to the Calculator.dll, implement the code using the directions below and finally build the binary code.

```
namespace netCalculator3WebService {  
    //...  
    public class Calculator3WebServiceMethod :  
        System.Web.Services.WebService {  
        // ...  
    }  
    //...  
}
```

The back-end code we are enabling as a Web Service is the add(ICalculator3Args) method. However, Web Services cannot pass interfaces as arguments, as this would assume a valid implementation of ICalculator3Args on any service consumer which wants to use the Web Service. This would be bad practice, since it restricts interoperability and relies on consumers having knowledge of the service implementation, which is against the principles of Web Services.

Instead, we have chosen to take in an array of floats as the argument for the Web Service method. Within the code of the Web Service method, this array is transformed into an object of type Calculator3Args, which can then be passed to call the back-end business code for the list of numbers to be added together. The result from the operation is returned as a float from the Web Service.

The Web Service method code is shown below:

```
[WebMethod]//Declares web method  
public float add (float[] args) {  
    //Create object to pass parameter array  
    ICalculator3Args argsArray = new Calculator3Args();  
    for (int i=0;i<args.Length;i++)  
        argsArray.setArg(args[i]);  
    //Create Calculator object  
    ICalculator3 calc3 = new Calculator();
```

```
//Send float array as parameter to get sum of array
float result = calc3.add(argsArray);
//return result
return result;
}
```

Note that for testing and developing the projects, we used *localhost* as our host name, but when you publish your Web Service to the outside world, we recommend that you use a more meaningful host name.

- The netCalculator2WebService URL:

```
http://<machine_name>/netCalculator2WebService/Calculator2WebServiceMethod.
asmx
```

- The netCalculator3WebService URL:

```
http://<machine_name>/netCalculator3WebService/Calculator3WebServiceMethod.
asmx
```

Deployment

To provide the Web Service to the consumer, it is necessary to deploy it on the IIS server. The deployment of the Web Service involves copying the .asmx file and assemblies used by the Web Service to a virtual Web directory. The deployment can be done manually by creating folders copying the files into the virtual directory, or by using a setup program.

The directory structure in IIS for the Web Service netCalculator3WebService is as below:

```
\Inetpub
  \wwwroot
    \netCalculator3WebService
      Calculator3WebServiceMethod.asmx
    \Bin
      Calculator.dll
      netCalculator3WebService.dll
```

Additionally, you can add a discovery (or .disco) file to make the Web Service available using the discovery mechanism. The disco mechanism can be added by enabling discovery for the Web Service.

Design considerations for the service provider

This section discusses some design consideration for designing a .NET Web Service.

- Parameter validation and type handling

A Web Service available on the Internet is available to everyone who has proper access, so make sure that the parameters passed to your Web

Service are valid. By passing invalid parameters, hackers can generate errors. To avoid this, perform proper type validation for Web Service parameters.

It is possible that your Web Service may be utilized in a language other than the language used for the Web Service. In such cases, make sure you received proper data with the correct type.

► Security

.NET has several security models for Web Services. In .NET, the security can be applied at different levels. The basic security model for Web Service consists of:

- Authentication and authorization
- Securing the connection
 - Authentication
 - Authorization
 - Securing the connection

► Use of cache

The Web Service performance can be improved by enabling caching. When caching is enabled, the output of the Web Service is cached on the server. The cached output then can be shared with other requests without re-executing the service.

8.1.3 Service consumer

It is assumed at this stage that the Microsoft .NET Web Services have been written, and are currently running on a server accessible from the machine that will be used as the service consumer (or at least used for code development for the service consumer).

The scenario we use to demonstrate the interoperability between the technologies is a simple Average Calculator, which includes two operations. One operation is for calculating the average of two numbers, and the other is an operation for calculating the average of a list of x numbers.

The Average Calculator, instead of implementing its own Add code, will instead make use of the Web Service methods for adding, as developed in the Microsoft .NET service provider section above.

The Average Calculator functionality for calculating the average of two numbers will invoke the `add(float, float)` Web Service method associated with the `ICalculator2` interface. The average of a list of numbers will be performed with the invocation of the `add(float[])` Web Service method, which in turn calls the

`add(ICalculator3Args)` method associated with `ICalculator3` on the Web Service side. This is, however, unknown to the service consumer, and is irrelevant since the only information they are interested in is the method signature of the actual Web Service methods and how to call it, all of which is contained within the WSDL files.

Important: The Web Service *consumer* has been implemented using its own implementation of an `ICalculator` interface. The methods and structure are, in this implementation, the same as were used on the service *provider* side. This is unlikely to be the case in a real-world scenario, but we have used the same implementation pattern to simplify understanding of the code for the reader and increase the speed of development.

This section will describe how to implement our solution to this scenario. The implementation is shown in Figure 8-11 on page 348.

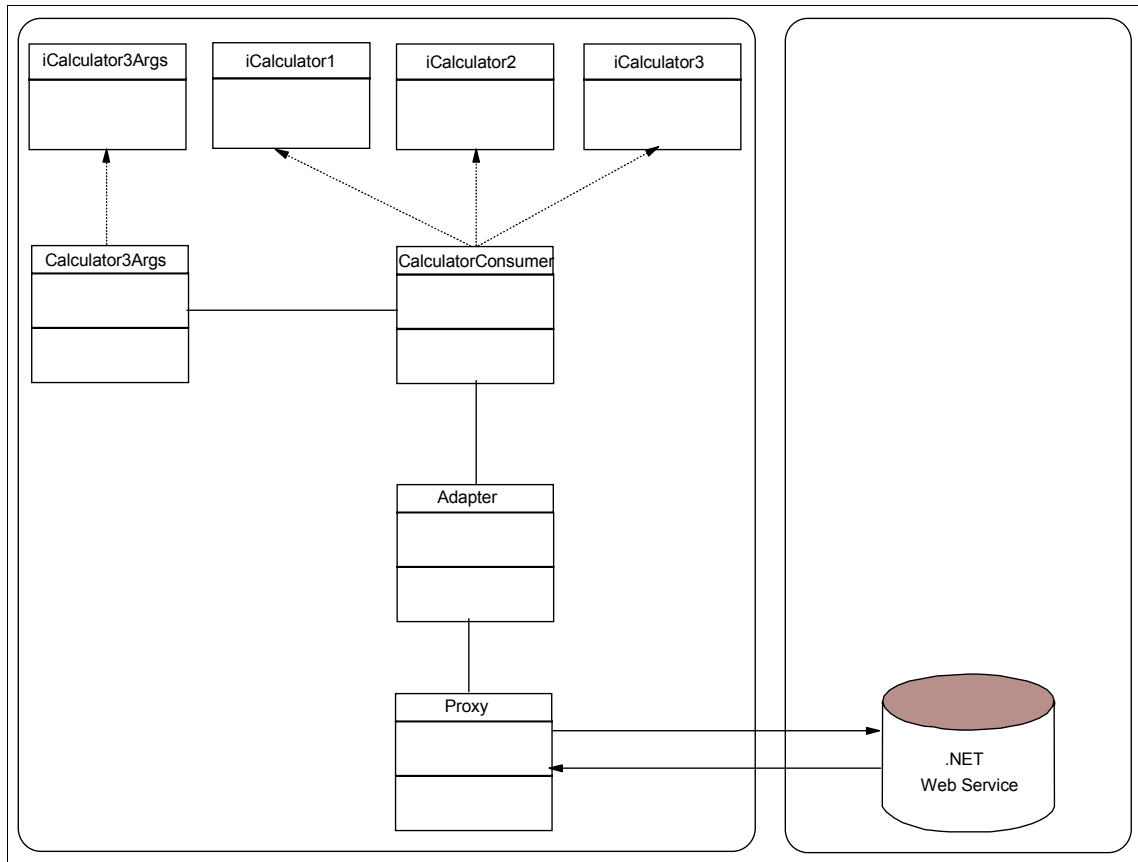


Figure 8-11 Class diagram for simple solution model

The methods of the `CalculatorConsumer` and the classes of the `Calculator.jar` file are shown next.

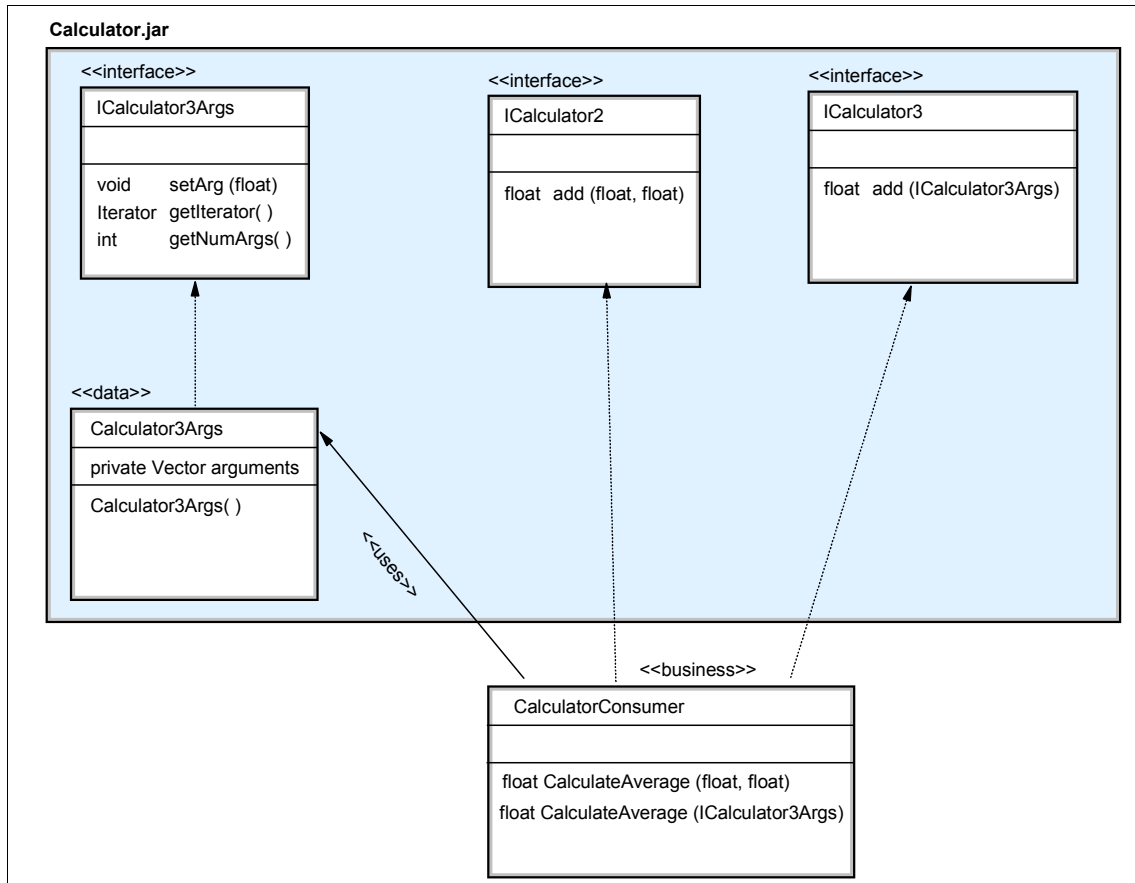


Figure 8-12 Classes and methods in the solution

The CalculatorConsumer class is responsible for the business logic of the application. It contains methods for calculating the average of two numbers and for calculating the average of a list of numbers. It implements the add methods defined in the interfaces ICalculator2 and ICalculator3.

When calculating the averages, the Average Calculator application calls the add methods in the CalculatorConsumer class. These add methods in turn route the request, via an adapter, to a Web Service proxy. The adapter is responsible for converting any data types used on the service consumer side into the necessary format (and back) for using the Web Service methods. For example, the add(ICalculator3Args) method in CalculatorConsumer makes use of the add(float[]) Web Service method, so the Adapter is needed to convert the ICalculator3Args object into a float array.

The proxy then makes the Web Service call to the service in the Microsoft .NET environment. The specifics of how the service is implemented is not known by the proxy. The proxy class simply knows where the Web Service is located and how to call it. The Web Service returns the result of the `add()` operation, which is returned back up through the adapter into the `CalculatorConsumer` class. The `CalculatorConsumer` uses the result, dividing it by the number of arguments received, and outputs the average of the numbers.

The following sections describe how to implement this solution, first looking at invoking the RPC style `add(float, float)` Web Service, and secondly invoking the `add(float[])` Web Service using Document style communication. Once the Web Service proxies have been implemented, we will move on to discuss the implementation of the complete scenario including the `CalculatorConsumer` class and the Average Calculator application.

We will be using the WebSphere Studio Application Developer Integrated Development Environment (IDE) for implementing our solution. Start WebSphere Studio Application Developer in a new (empty) workspace ready for development.

We are going to be implementing a completely deployable application, so first you should create a new Enterprise Application Project which will contain our other modules (to be implemented later). Call your new EAR project `SimplisticWS2NETCalculator`, as this will be a simple implementation of the Calculator code to demonstrate interoperability between WebSphere and Microsoft .NET.

Scenario implementation with the RPC model

In order to create a proxy to a Web Service written in any language, on any platform, you should only need access to the WSDL file for that Web Service. It should contain all information needed to invoke the service.

WebSphere Studio Application Developer will automatically generate a Web Service client from a WSDL file.

You will need to create a Web project for the client to go into. Call this `Calculator2ClientProxy`. Now you can go to **File -> New -> Other -> Web Service -> Web Service Client**. Choose to create a **Java proxy**, and on the next window select the Web project you would like the client code to go into, in our case: **Calculator2ClientProxy**.

The Web Service Selection Page is where you enter the URI to the WSDL document previously generated in Microsoft Visual Studio .NET. You could either save the WSDL to your workspace and browse to it there, or simply enter the URL of the WSDL on the remote machine, in our case:


```
http://localhost/netCalculator2WebService/Calculator2WebServiceMethod.asmx?  
WSDL
```

You can accept all wizard defaults from there. Click **Finish**.

The wizard will automatically create several files, including four Java files under the `JavaSource` directory. If it isn't already, name the package of the files to be `redbook.coex.stateless.synchronous.ws2net.business`. It is likely that the tooling will automatically give the package a name based on information contained in the WSDL file, but it is good practice to choose your own package name based on your own naming conventions.

The four new Java files created by the wizard handle the generation of the SOAP messages and invocation of the Web Service. All the complexities involved in using the Web Service are handled for you automatically. All the user has to do is call the relevant methods of these new Java classes. This will be described in more detail later on.

Scenario implementation with the Document style model

Creation of the Document style Web Service proxy is exactly the same as for the RPC style, since the communication method used by the Web Service is defined by the service provider and specified in the WSDL file. WebSphere Studio Application Developer determines the transport method from the WSDL file automatically, and generates the necessary classes automatically, again hiding all complexities from the user.

So in this case, create a new Dynamic Web project called `Calculator3ClientProxy`. Go to **File -> New -> Other -> Web Service -> Web Service Client**, create a Java proxy, and on the next window select the **Calculator3ClientProxy** Web project.

On the Web Service Selection Page, enter the URI to the Calculator3 WSDL file, in our case:

```
http://localhost/netCalculator3WebService/Calculator3WebServiceMethod.asmx?  
WSDL
```

You can again accept all defaults from there. Again, rename the Java package where the new Java files are placed to `redbook.coex.stateless.synchronous.ws2net.business`.

On this occasion, more than four core Java files will be created because the array of floats is handled as a complex data type. There are extra classes added for the handling of the serialization and deserialization of the complex type, as well as a Java bean to represent the object itself, in this case `ArrayOfFloat.java`.

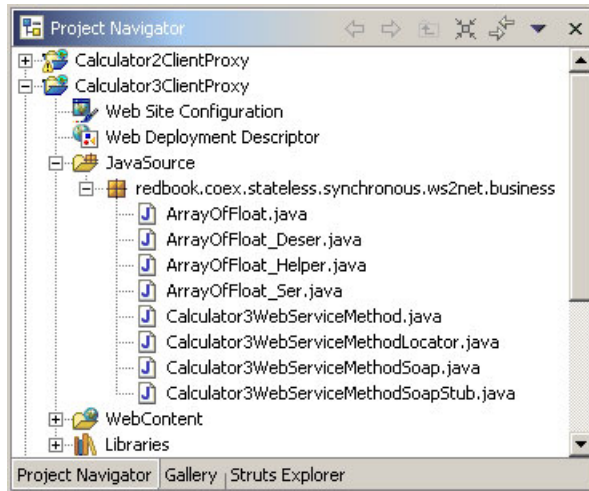


Figure 8-13 Files created by Client wizard

These classes are of little concern to the user, since all serialization and deserialization is handled behind the scenes, and is performed automatically by the new generated classes.

Now that the code has been generated to call the Web Services, the rest of the coding of the solution is standard Java development.

Implementing the Average Calculator

As shown in Figure 8-11 on page 348, the proxy classes are invoked from an Adapter, the Adapter being responsible for conversion between the data format used by the Web Service method and data format used by the CalculatorConsumer class (which is developed later).

Since different functionality is required for the Calculator2 and Calculator3 Web Services, two separate Adapter classes were implemented. In the Calculator2ClientProxy, under the redbook.coex.stateless.synchronous.ws2net.business package (the same location as the four generated proxy classes), create a new Java class called Calculator2Adapter.

This class *implements* the `add(float, float)` method associated with the `ICalculator2` interface. The implementation of the Calculator2Adapter class is shown below.

Example 8-4 Calculator2Adapter code

```
package redbook.coex.stateless.synchronous.ws2net.business;

import java.rmi.*;
import javax.xml.rpc.*;
import redbook.coex.sall.business.*;

public class Calculator2Adapter implements ICalculator2 {

    public float add(float arg1, float arg2) {
        float result = 0;
        try {
            //create an instance of the web service proxy
            Calculator2WebServiceMethodLocator locator = new
Calculator2WebServiceMethodLocator();
            Calculator2WebServiceMethodSoap proxy =
locator.getCalculator2WebServiceMethodSoap();
            //call the web service method
            result = proxy.add(arg1, arg2);
        } catch (ServiceException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

The above code shows just how easy it is to actually call the Web Service. The only code required by the user is the creation of the `Calculator2WebServiceMethodSoap` object, then the calling of the method `add(float, float)`.

Important: The Calculator2Adapter (and Calculator3Adapter later) implements the ICalculator2 interface. The interface is packaged in the Calculator.jar file for convenience.

In order to make the application work, create a new Java project, for example: CalculatorCode, and import the Calculator.jar file. Open the SimplisticWS2NETCalculator application descriptor (application.xml) and add the CalculatorCode Java project. Add the CalculatorCode Java project to the Java Build Path of the Web project and to the Java JAR Dependencies list.

It is very important to understand that the interfaces packaged with the Calculator.jar are only for your convenience; in real life, the client and the implementation interfaces are not going to be the same and the client has to be implemented purely based on the WSDL input.

The Calculator3Adapter class in the Calculator3ClientProxy project is very similar. However, the purpose of the Adapter is to perform any necessary data format changes between the Average Calculator code and the Web Service call. As the Calculator3Adapter class implements ICalculator3, it implements the add(ICalculator3Args) method, but the Web Service method requires a float array, so this is where the conversion takes place.

Example 8-5 Calculator3Adapter code

```
package redbook.coex.stateless.synchronous.ws2net.business;

import java.rmi.*;
import java.util.*;
import javax.xml.rpc.*;
import redbook.coex.sall.business.*;

public class Calculator3Adapter implements ICalculator3 {
    public float add(ICalculator3Args args) {
        float result = 0;
        Iterator iterator = args.getIterator();
        float[] argsFloatArray = new float[args.getNumArgs()];
        int i = 0;
        while (iterator.hasNext()) {
            argsFloatArray[i] = ((Float)iterator.next()).floatValue();
            i++;
        }
        ArrayOfFloat arrayOfFloat = new ArrayOfFloat();
        arrayOfFloat.set_float(argsFloatArray);
        try {
            //create an instance of the web service proxy
```

```

        Calculator3WebServiceMethodLocator locator = new
Calculator3WebServiceMethodLocator();
        Calculator3WebServiceMethodSoap proxy =
locator.getCalculator3WebServiceMethodSoap();
        result = proxy.add(arrayOfFloat);
    } catch (ServiceException e) {
        e.printStackTrace();
    } catch (RemoteException e) {
        e.printStackTrace();
    }
    return result;
}
}

```

The Calculator3Adapter class takes an ICalculator3Args object into its add() method, converts it into an instance of the pre-generated ArrayOfFloat object, and calls the Web Service in the same way as the Calculator2Adapter does.

The business logic code of the Average Calculator is implemented in the class CalculatorConsumer.java. Create a new Java Project called CalculatorConsumer, and create a new Java Class called CalculatorConsumer in the package redbook.coex.stateless.synchronous.ws2net.business.

This class implements the ICalculator interfaces, so add either CalculatorCode project to the Java build path. The Average Calculator has no need for the ICalculator1 interface, so just implement ICalculator2 and ICalculator3. Furthermore, the class uses the Adapter classes from the Web projects, so add the Calculator3ClientProxy and the Calculator2ClientProxy projects to the Java build path also.

Example 8-6 CalculatorConsumer code

```

package redbook.coex.stateless.synchronous.ws2net.business;

import redbook.coex.sall.business.*;

public class CalculatorConsumer implements ICalculator2, ICalculator3 {
    public float add(float arg0, float arg1) {
        Calculator2Adapter adapter = new Calculator2Adapter();
        float result = 0;
        result = adapter.add(arg0, arg1);
        return result;
    }
    public float add(ICalculator3Args args) {
        float result = 0;
        Calculator3Adapter adapter = new Calculator3Adapter();
        result = adapter.add(args);
        return result;
    }
}

```

```

    }
    public float calculateAverage(float arg0, float arg1) {
        float total = add(arg0, arg1);
        return total/2;
    }

    public float calculateAverage(ICalculator3Args args) {
        float total = add(args);
        return total/args.getNumArgs();
    }
}

```

CalculatorConsumer.java has implementations of the two add methods as defined in ICalculator2 and ICalculator3. It also has the business logic for the actual calculateAverage() methods, one method taking in two floats and the other taking in the list of arguments ICalculator3Args. These methods make use of the add methods, which in turn call the add methods of the Adapter classes, which use the proxy classes to make the actual Web Service calls.

Add the CalculatorConsumer Java project to the SimplisticWS2NETCalculator enterprise application as a utility project.

Now we will write the actual front end of the application. In this case, we have chosen to implement a thin client, where JSPs call the CalculatorConsumer class in order to get the results.

Create a new Dynamic Web Project called SimplisticCalculatorClient. Add the CalculatorCode and the CalculatorConsumer projects to the Java build path and to the Java JAR dependencies for this new project.

Create a new HTML file and name it AverageCalculator.html. This file should be the front page for the Average Calculator, with the ability to enter two numbers and calculate the average, or enter a list of numbers and calculate the average.

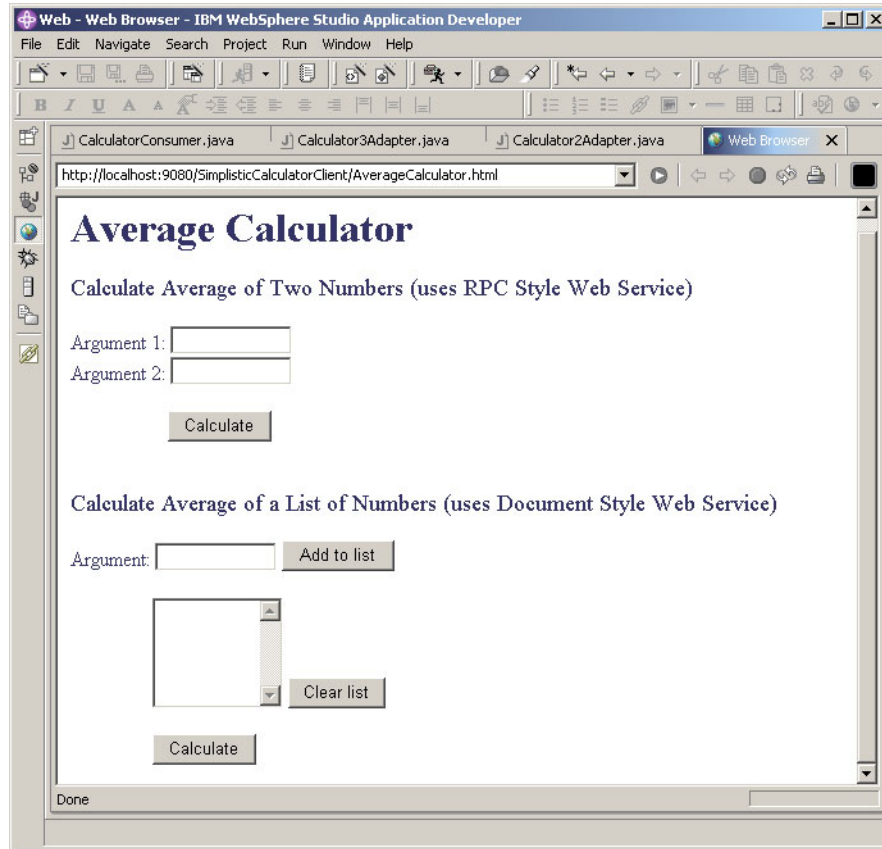


Figure 8-14 The thin client HTML page

When the Calculate buttons are clicked, the form is redirected to a JSP file. The average of two numbers button is redirected to the `CalculateAverageRPCStyle.jsp`, and the average of a list is directed to `CalculateAverageDocStyle.jsp`. Create the two JSP files also.

The code in the JSP looks as shown in Example 8-7 on page 358.

Example 8-7 A section of CalculateAverageRPCStyle.jsp

```
<jsp:useBean id="CalcConsumer"
class="redbook.coex.stateless.synchronous.ws2net.business.CalculatorConsumer"
scope="request"></jsp:useBean>
<%
    float arg1 = new Float(request.getParameter("Arg1")).floatValue();
    float arg2 = new Float(request.getParameter("Arg2")).floatValue();
    float result = CalcConsumer.calculateAverage(arg1, arg2);
%>
<%=result%>
```

The JSP uses the CalculatorConsumer bean, reads in the arguments, converts them into floats, and calls the calculateAverage method. The calculateAverage method calls the appropriate add method in the CalculatorConsumer class (as defined in the ICalculator interfaces). The add method in the CalculatorConsumer class uses the appropriate add method in the Adapter class, which converts the data into the necessary format, then uses the proxy to invoke the .NET Web Service. The result from the .NET Web Service is returned to the CalculatorConsumer class via the Adapter where it is used to calculate the average of the list of numbers. The result is output to the JSP.

Once this is complete, the application is ready for deployment to WebSphere Application Server. Select the **SimplisticWS2NETCalculator** EAR project and export the EAR file. When this is deployed in WebSphere Application Server and is running, the front page URL will be:

http://<machine_name>:9080/SimplisticCalculatorClient/AverageCalculator.html

The application can also be tested from within WebSphere Studio Application Developer, using the WebSphere Test Environment. To do this, simply select the **AverageCalculator.html** file and select **Run on Server...**

Important: In order for the application to function, a small change needs to be made to the Server configuration. The WAR classloader policy for the SimplisticWS2NETCalculator application needs to be set to Application (as opposed to Module). See the WebSphere InfoCenter for instructions on how to do this.

8.1.4 Test

In order to test, you will need to deploy the new enterprise application to a WebSphere server or run it in the WebSphere Studio Test Environment.

To run the test environment, you can either create a new WebSphere V5.02 Test server or just right-click the enterprise application and select **Run on server**; this

creates a new test server automatically after selecting the server type and the HTTP port.

The Test server will automatically start, but the settings are not entirely correct for the application. Stop the server and open the server configuration. Select the **Applications** tab, then select the **simplisticWS2NETCalculator** application. Since the Web application not only uses utility JARs, but also classes from other Web modules, we need to set the WAR classloader policy to APPLICATION. Save the configuration, close the file then start the server.

Once the application server is running, right-click **AverageCalculator.html** under the SimplisticCalculatorClient project and select **Run on server...** Select the Existing server and wait until the browser comes up with the HTML content.

Once the HTML content is available, provide some random data and click the **Calculate** button. Perform the operation with both calculators (two numbers, list of numbers).

8.2 Extended solution model

The solution described above is a simple solution to show how to quickly and simply get WebSphere talking to Microsoft .NET business logic in a stateless, synchronous way using Web Services. This section will look at different ways to implement a full solution in order to gain quality of service over the simple model.

An extended solution to the problem

This section will discuss more advanced topics with regard to designing an interoperable scenario for accessing a Microsoft .NET Web Service from a WebSphere client. The advanced topics will be discussed based on an extended implementation of the Average Calculator scenario.

The simple solution described earlier will be extended to provide better separation between the tiers of the five-tier architecture model, and to provide better quality of service. To understand what is meant by the term Quality of Service in this context, please refer to Chapter 11, “Quality of service considerations” on page 471.

Extended scenario considerations will be considered from the point of view of both the service provider and the service consumer.

Service consumer

There are many possible scenarios for stateless, synchronous interoperability between WebSphere and .NET, as discussed in Chapter 4, “Technical

coexistence scenarios” on page 109. The diagram below shows the different layer-to-layer communication scenarios.

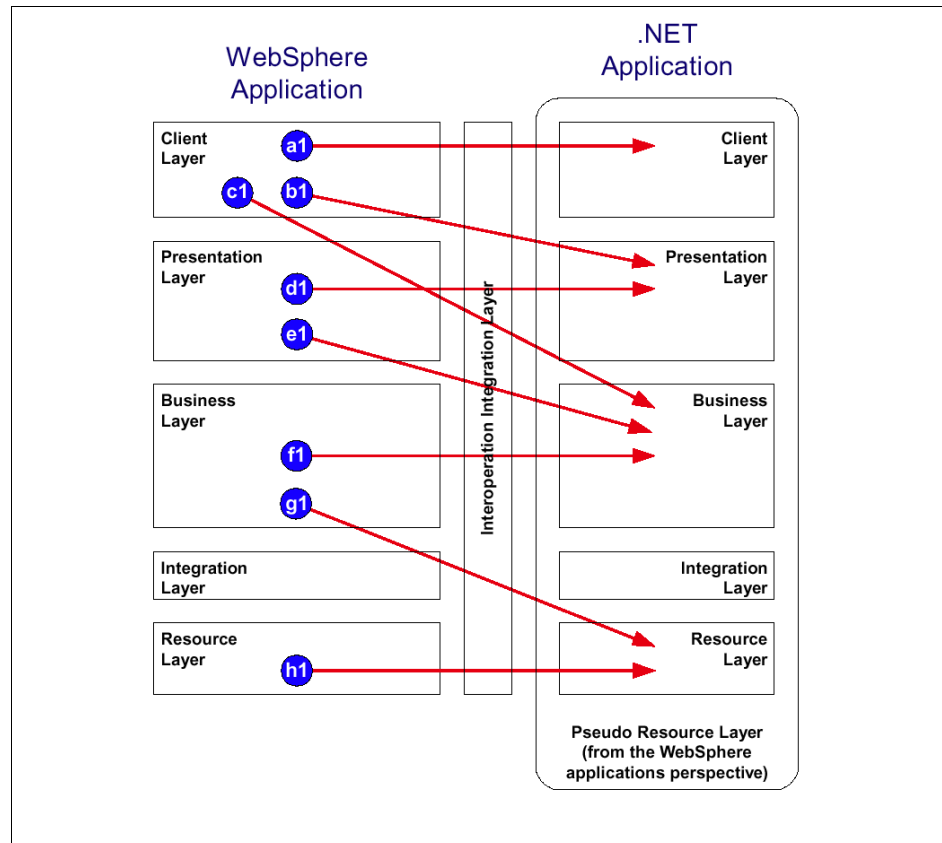


Figure 8-15 Possible layer-to-layer Interop options

We will assume that there already exists some back-end code on the .NET side which will be made available as a Web Service. How the service logic is implemented on the .NET side is unknown and irrelevant to the service consumer. As it is normally the case that Web Services would reside on the Business layer of an application, we shall assume this is the case on the .NET side, although it would make little difference if it were actually in the Presentation layer.

As shown in Figure 8-15, the cases for a WebSphere solution to interoperate with the Business layer of the .NET side are c1, e1 and f1.

We will therefore look at examples of how to implement those scenarios using stateless, synchronous communication.

Scenario c1

Scenario c1 concerns the Client layer on the WebSphere side interoperating with the Business layer of the .NET side.

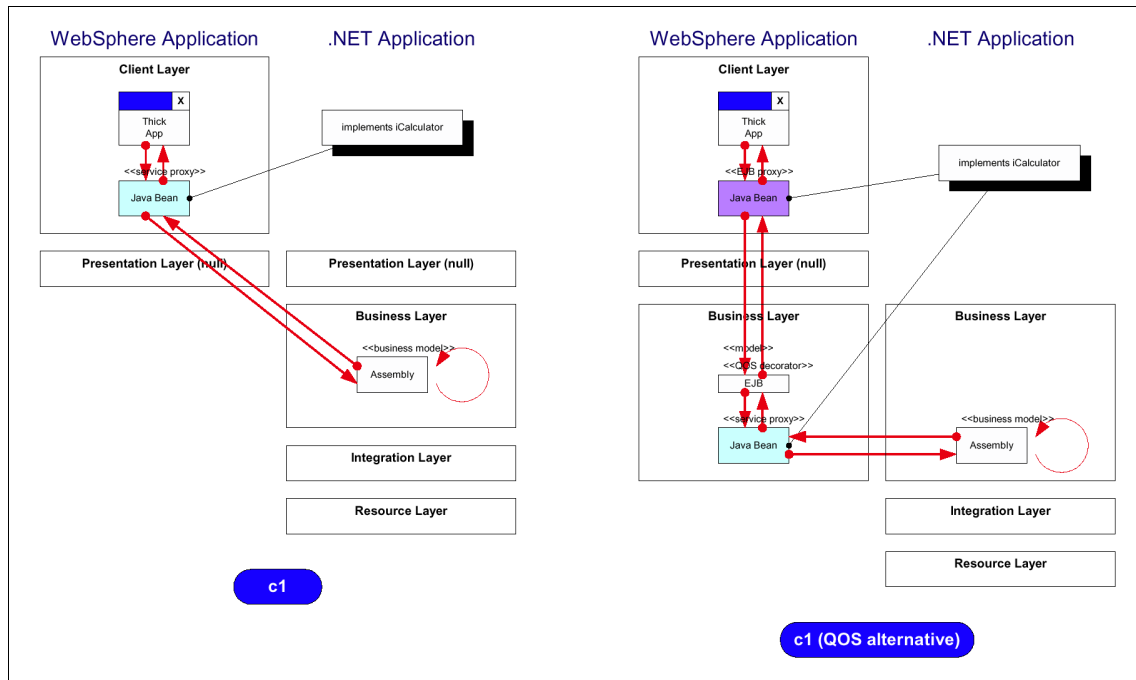


Figure 8-16 Scenario c1

So, with our assumption that there already exists some back-end code on the .NET side, this would be made available as a Web Service in a similar way as was described in the simple scenario chapter.

This will involve a WSDL file being generated to describe the Web Service, and this would then be used in WebSphere Studio Application Developer to create a proxy to the Web Service. As a simple case, this proxy could simply reside as a Java Bean on the Client layer, being invoked directly from a thick client application, such as a Swing GUI. This would demonstrate Client layer WebSphere interoperating in a stateless, synchronous manner with Business layer .NET code.

This is a very basic approach to this implementation, and offers very little in the way of quality of service. A better approach to this method would be to implement the solution as shown in the c1 (QoS alternative) diagram. This is still, in effect, a way to interoperate between the Client layer and Business layer, but the actual interaction phase is between Business layer and Business layer.

So ultimately, a better way to implement a Client layer to Business layer solution is in fact to go via the Business layer on the WebSphere side, which boils down to an implementation of Scenario f1, described later.

Scenario e1

Scenario e1 concerns the Presentation layer on the WebSphere side interoperating with the Business layer of the .NET side.

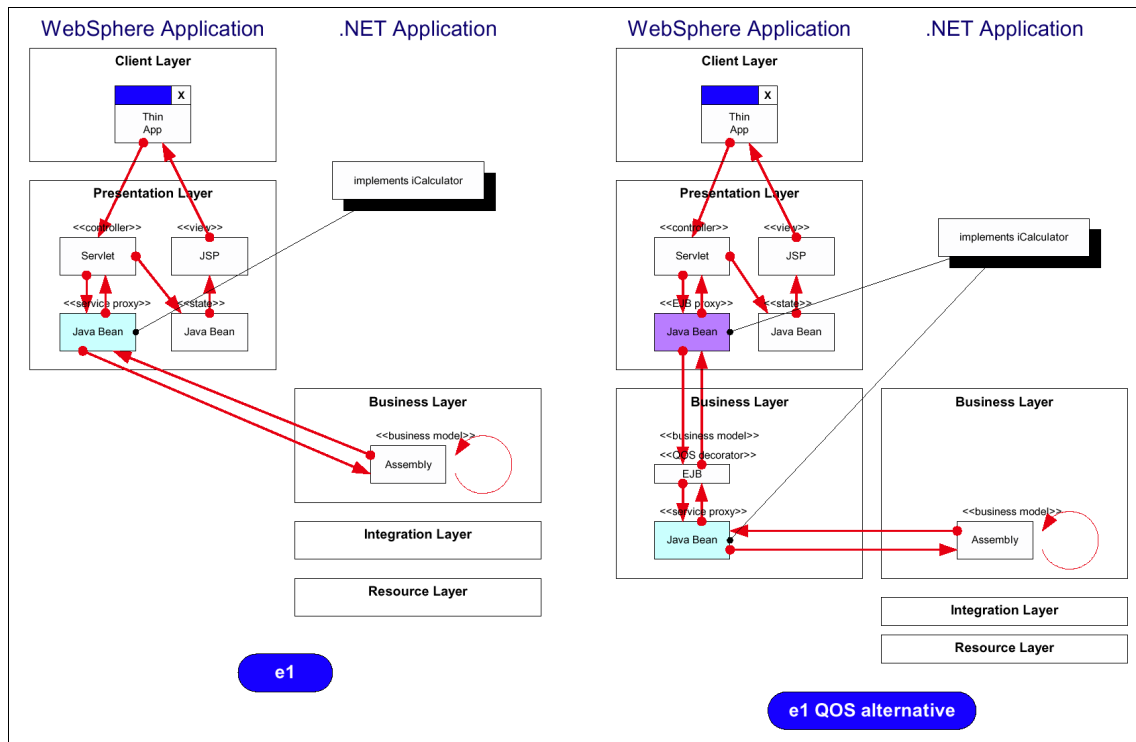


Figure 8-17 Scenario e1

Once again, assume that the Business layer of the .NET application is a Web Service access point to some back-end code. Again, the WSDL is used on the WebSphere side to create a Java Bean service proxy.

This scenario differs from c1 in that instead of the Java proxy being directly used by a thick client, it is instead being used from a controller servlet in the Presentation layer. The user interacts with a thin application (such as an HTML page), which calls the controller servlet to make the Web Service call. If the application requires it, any state Java Beans would also be part of the Presentation layer and used by the servlet. Finally, the result is passed back from

the Web Service, via the controller servlet and any state Java beans, into a JSP which is responsible for displaying the result to the user.

An example of such a state Java bean would be the `Calculator3Args` class implemented as part of the simple solution to our Average Calculator. Although the actual interoperability phase of the solution is stateless, the `Calculator3Args` class maintains the state of the list of arguments that is ultimately used in the `add(ICalculator3Args)` method of the `CalculatorConsumer` class.

However, the solution described again provides little quality of service, which can be improved by instead implementing the *e1 QoS alternative* solution. This solution moves the Java bean service proxy into the Business layer of the WebSphere application and is accessed via an EJB “decorator” which adds quality of service associated with EJBs.

This, like the *c1 QoS alternative* solution, ultimately boils down to an implementation of Business layer to Business layer interoperability, which is described in scenario f1.

Scenario f1

Scenario f1 concerns interoperability between the Business layer on the WebSphere side interoperating with the Business layer of the .NET side.

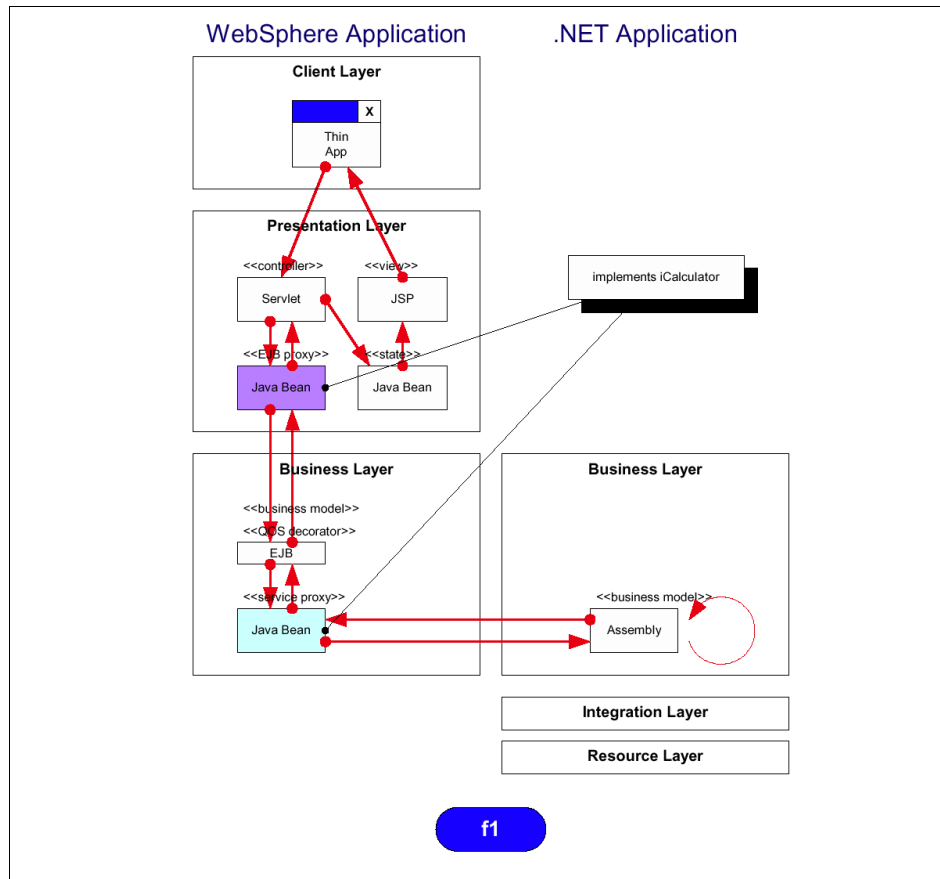


Figure 8-18 Scenario f1

This solution is analogous to the quality of service implementations of scenarios c1 and e1, and is considered to be a best practice approach to any implementation of stateless, synchronous interoperability between WebSphere and the .NET Business layer.

This type of approach encourages distinct separation between layers of the solution, which follows best practice guidelines and enhances maintainability of the code.

The actual interoperability phase of the solution is identical to that of the simple solution. The .NET assembly, which contains the implementation of the Web Service on the .NET service provider, has a WSDL file describing the service. This WSDL file would be consumed on the WebSphere side (using a tool such as WebSphere Studio Application Developer) and used to generate a Java Bean

service proxy capable of invoking the Web Service, as was done in the simple solution.

This Java proxy is accessed from the Presentation layer via an EJB, which contains no actual implementation of logic for the solution, but merely acts as a quality of service decorator to the proxy.

There is a Java bean in the Presentation layer which basically acts as a proxy to the EJB from the controller servlet which drives the Web Service. This ultimately means that all presentation logic is separated from the Business layer implementation, and the business logic is only accessible via the EJB. The EJB will automatically handle issues such as scalability, performance, security and availability, therefore giving such qualities to the overall consumer-side solution.

There is not much that can be done to add quality of service to the actual stateless synchronous interoperability of the Web Service. Additional qualities such as guaranteed delivery can be achieved by using alternative transport protocols, such as Web Services over JMS. However, JMS messaging is asynchronous in nature and is not an appropriate solution for this scenario.

Ultimately, any extended solution implementation on the service consumer side is basically just J2EE best practice, and has little to do with the actual interoperability of the solution. Since this book is intended to demonstrate ways of interoperability between .NET and WebSphere, the extended functionality has not been implemented in this book.



Scenario: Web interoperability

The purpose of this chapter is to discuss issues relating to interoperability between Web applications residing in WebSphere Application Server and Microsoft .NET.

The following scenarios are identified for Web interoperability:

- ▶ Shared presentation components
- ▶ Session state interoperability
- ▶ Data propagation
- ▶ Security

9.1 Introduction

You can find the related scenario in Chapter 4, “Technical coexistence scenarios” on page 109. In this chapter, we discuss sub-scenarios that apply to Web application interoperability between the two platforms, WebSphere and .NET. The chapter includes the following sections:

- ▶ 9.2, “Shared presentation components” on page 368

In this section, we examine options for servicing Web content in a heterogeneous environment using the Microsoft Internet Information Services Web server. In addition to discussing various interoperability scenarios, this section also documents how to configure Microsoft IIS for interoperability with WebSphere Application Server.

- ▶ 9.3, “Session state interoperability” on page 376

Web applications nearly always require information about the user and their interaction with the application to be maintained during the life cycle of that user’s session. Therefore, one of the key considerations for making Web applications in a mixed environment interoperable is how to implement statefulness between the applications. This section describes various strategies for implementing statefulness.

- ▶ 9.4, “Data propagation” on page 395

This section discusses scenarios and solutions for sharing data between Web applications operating in a heterogeneous environment.

- ▶ 9.5, “Integrated security” on page 418

In this section, we examine a mechanism for sharing security credentials across WebSphere Application Server and Microsoft IIS. The section provides an implementation of a shared security model which leverages existing technologies to provide single sign-on in an heterogeneous environment.

9.2 Shared presentation components

This section documents how to configure the Microsoft Internet Information Services (IIS) for operation in a heterogeneous environment. Figure 9-1 on page 369 illustrates how Microsoft IIS can be used in such an environment.

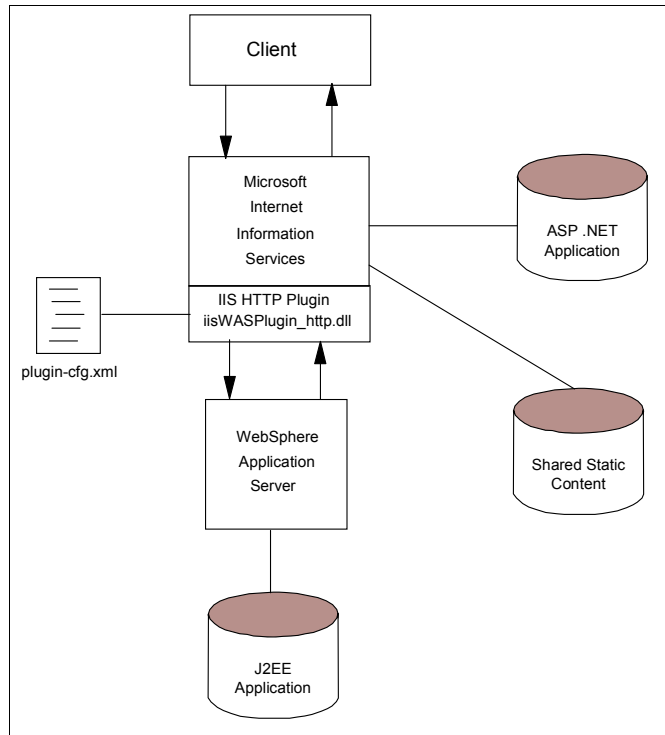


Figure 9-1 Microsoft IIS server in a heterogeneous environment

In the shared presentation components scenario, the IIS Web server handles requests for both static and dynamic content in the following manner:

- J2EE application resources

In this scenario, the client makes a request for a dynamic Web component hosted by the WebSphere Application Server. When the request is received by IIS, the WebSphere Application Server HTTP plugin examines the URL of the request and attempts to match it with the virtual host and URI mappings contained in its configuration file. In this instance, since the resource that is being requested is hosted in the WebSphere Application Server Web application, the plugin matches the URL and forwards the request to WebSphere. WebSphere Application Server processes the request and returns the response via Microsoft IIS to the client.

- ASP.NET application resources

In this scenario, the client makes a request for a resource hosted in an ASP.NET Web application. As before, the request URL is compared to its configuration file mappings by the HTTP plugin. This time, however, since the resource is not being hosted by WebSphere Application Server, no match

occurs. The plugin now forwards the request to Microsoft IIS, which processes it and sends the response back to the client.

► Static content

In this scenario, the client makes a request for some static content; this could be an HTML page or an image. Whether the content relates to a WebSphere Application Server or Microsoft .NET Web application is not relevant; it could be either. The HTTP plugin again attempts to match the URL and fails. The request is forwarded to Microsoft IIS, which services the requested resource back to the client.

9.2.1 Configuring Microsoft IIS for shared presentation

This next section illustrates how to configure Microsoft Internet Information Services for the scenario illustrated in Figure 9-1 on page 369. It is assumed that Microsoft Internet Information Services has already been installed. However, if it has not then an installation guide can be found at:

<http://www.microsoft.com/windows2000/en/server/iis/default.asp>

Installing the Microsoft IIS HTTP plugin

The Microsoft IIS HTTP plugin can be installed as part of the WebSphere Application Server installation. The installation procedure for WebSphere Application Server is documented in the IBM Redbook *IBM Application Server V5.0 System Management and Configuration*, SG24-6195.

To install the plugin, perform a custom install as described in the aforementioned redbook. When prompted for the features that you wish to install under the Web Server Plugins category, select the **Microsoft IIS plugin**, as shown in Figure 9-2 on page 371.

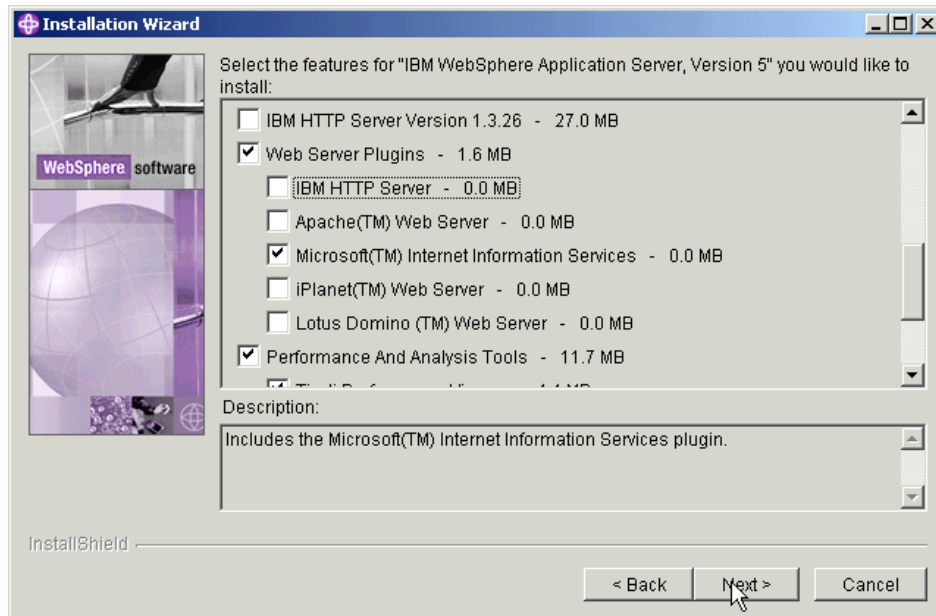


Figure 9-2 Installing the Microsoft IIS HTTP plugin

The WebSphere Application Server installation will install the HTTP plugin `iisWASPlugin_http.dll` and, if the instance of Microsoft IIS is on the same machine, will also configure Microsoft IIS for the plugin. In order for the changes to take effect in Microsoft IIS, you will need to restart the IIS services as follows:

1. Log on to the Windows 2000 server as a user with Administrator privileges.
2. Open the Windows control panel by clicking **Start -> Settings -> Control Panel**.
3. Select **Administrative Tools**.
4. Open the Internet Services Manager.
5. In the left-hand pane, right-click the machine icon and select **Restart IIS ...** from the context menu.
6. In the *Stop/Start/Reboot ...* dialog box, ensure that **Restart Internet Services** is selected from the pull-down menu and click **OK**.

Microsoft IIS will be restarted and the plugin will be loaded automatically.

Manually configuring Microsoft IIS for the HTTP plugin

If the instance of Microsoft IIS is on a remote machine then you will need to copy the HTTP plugin library and HTTP plugin configuration file, `plugin-cfg.xml`, to the

remote server; perform a manual configuration of Microsoft IIS using the following procedure:

1. Open the Internet Services Manager and, if not already started, start the IIS application, using the procedure described above.
2. Create a new virtual directory for the Web site instance that you intend to have work with WebSphere Application Server. To create this directory with a default installation, expand the tree on the left until you see Default Web Site. Right-click **Default Web Site** and select **New -> Virtual Directory**. In the wizard for adding a virtual directory, do the following:
 - a. Type `sePlugins` in the Alias to be used to Access Virtual Directory field.
 - b. Browse to the directory in the *Enter the physical path of the directory containing the content you want to publish* field.
 - c. Select the **Allow Execute Access** check box in the *What access permissions do you want to set for this directory* field.
 - d. Click **Finish** to add the `sePlugins` virtual directory to your default Web site.
3. Add the Internet Services Application Programming Interface (ISAPI) filter into the IIS configuration. Right-click the host name in the tree on the left pane and click **Properties**. In the Properties dialog, do the following:
 - a. Go to the Internet Information Services tab.
 - b. Click **WWW Service** in the Master Properties window.
 - c. Click **Edit** to open the WWW Service Master Properties window, as shown in Figure 9-3 on page 373.

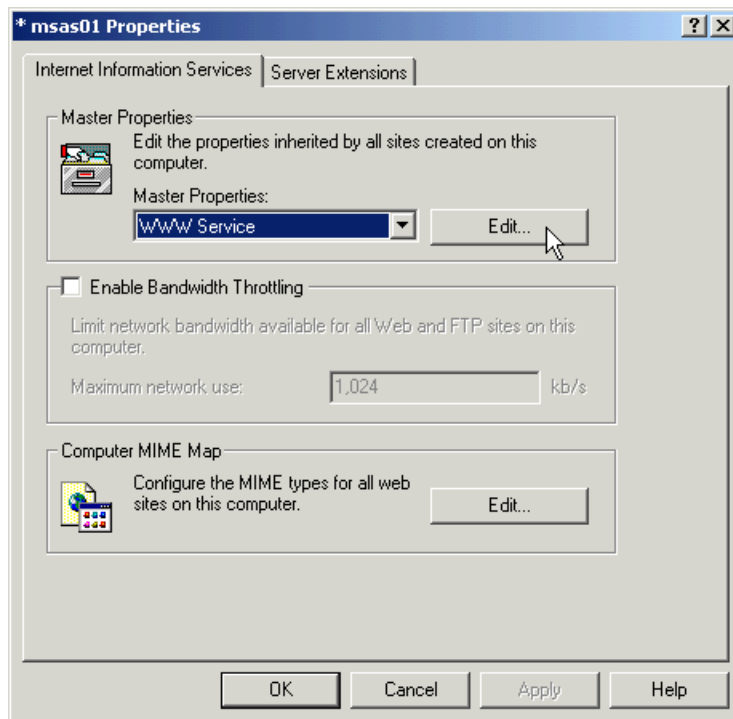


Figure 9-3 The Internet Services Properties dialog

- d. Click **ISAPI Filters** -> **Add** to open the Filter Properties window.
- e. Type `iisWASPlugin` in the Filter Name field.
- f. Click **Browse** in the Executable field.
- g. Browse to the directory where you copied the HTTP plugin library and click the `iisWASPlugin_http.dll` file. When completed, the window should appear as shown in Figure 9-4 on page 374.

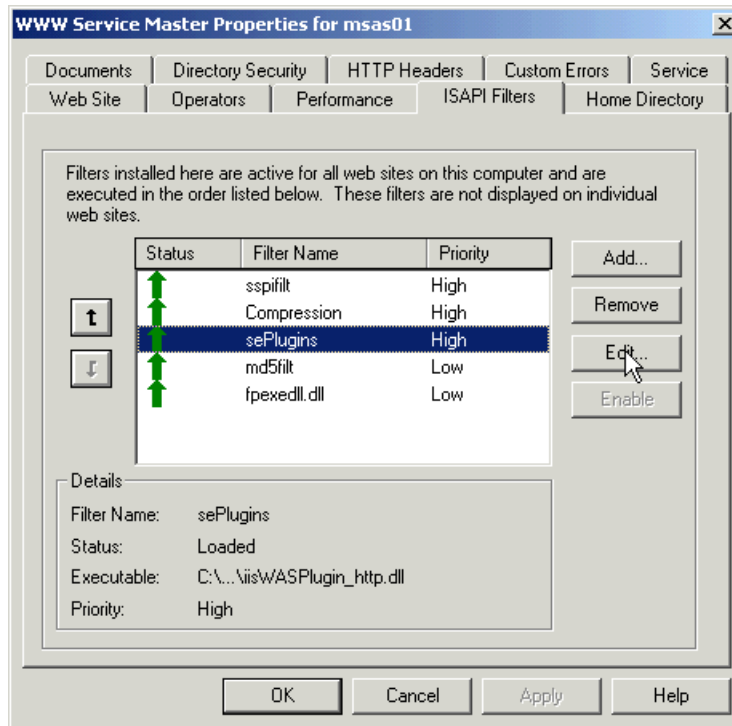


Figure 9-4 The WWW Services Properties dialog

4. Click **OK** until all open windows close.
5. Add the variable Plugin Config to the registry under the path **HKEY_LOCAL_MACHINE -> SOFTWARE -> IBM -> WebSphere Application Server -> 5.0.0.0**.
6. Set the value to the location of the configuration file, plugin-cfg.xml, which you copied across earlier.

Testing the configuration

Once you have completed the configuration, you can test that Microsoft IIS is servicing WebSphere Application Server, Microsoft IIS and static resources as follows:

1. Start the WebSphere Application Server and Microsoft IIS instances.
2. Open a browser and enter the following URL to test that IIS is able to service WebSphere content:

<http://<hostname>/snoop>

Where <hostname> is the hostname of the WebSphere Application Server machine. If you have performed the configuration correctly, the following window should appear.

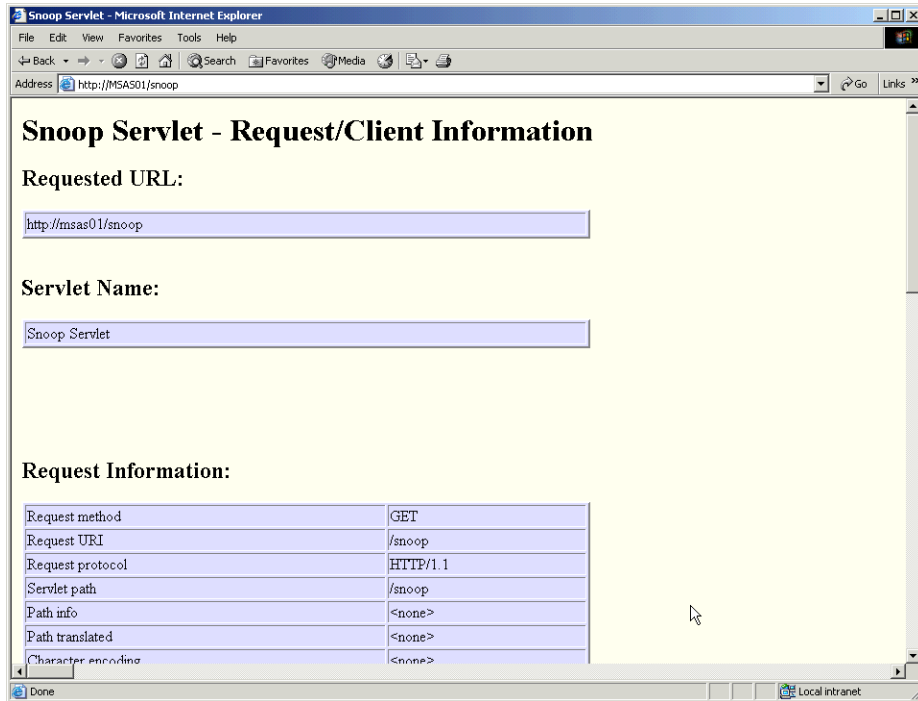


Figure 9-5 Testing the IIS Configuration for WebSphere Application Server

3. Next, test that Microsoft IIS is able to serve ASP.NET and static content by entering the following URL into the browser:

<http://localhost:9307>

If the configuration is correct, a window similar to the one shown in Figure 9-6 on page 376 should appear.

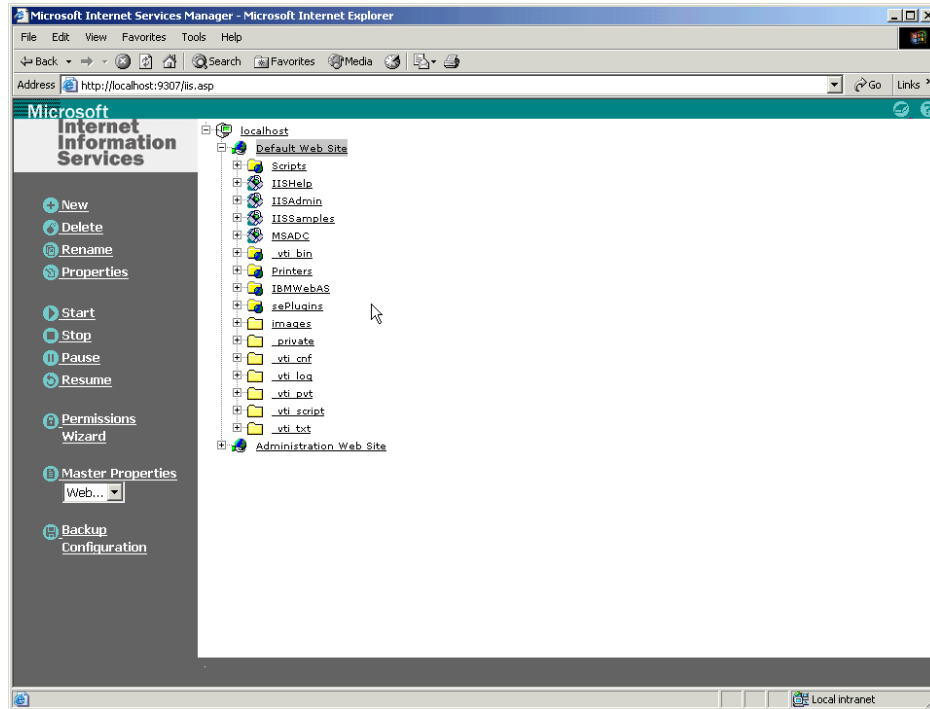


Figure 9-6 Testing the configuration for Microsoft IIS

9.3 Session state interoperability

The purpose of this section is to discuss Web interoperability issues concerning state management within Web applications. This section begins by defining the problem of state management in heterogeneous environments. We then discuss how state management is implemented in both WebSphere Application Server and Microsoft .NET. This discussion acts as a precursor to the remainder of this section, which deals with some of the particular difficulties in maintaining state across Web applications executing in both environments.

9.3.1 Problem definition

The Hypertext Transfer Protocol (HTTP) is a stateless protocol. In other words, the protocol as it is designed has no mechanism for maintaining user or application state information between successive requests. To implement an effective Web application, it is necessary to provide some sort of state management mechanism that allows requests from a particular client to be

associated with each other. Figure 9-7 illustrates how each technology addresses this problem.

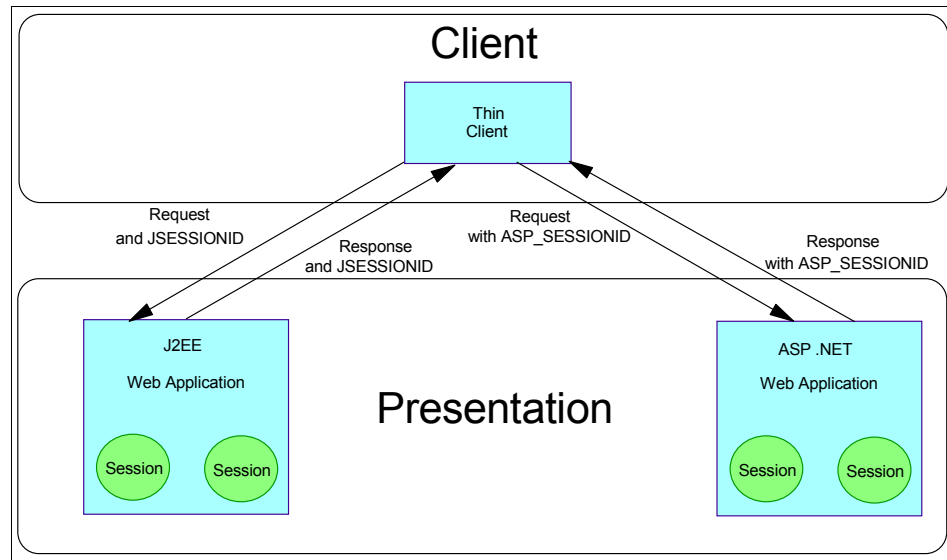


Figure 9-7 Session ID tracking within a heterogeneous environment

Both WebSphere Application Server and Microsoft .NET provide state management APIs and runtime services to enable stateful client interactions with Web applications. These implementations have many similarities, but what both implementations lack is any support for sharing of application state in a heterogeneous environment.

The problem that this section attempts to address is: how can Web applications running in WebSphere Application Server and Microsoft .NET be made to interoperate and share state information?

9.3.2 WebSphere Application Server session management

This section discusses how state management is implemented in J2EE; specifically, the discussion focuses on how WebSphere Application Server handles session management. The topics covered in this section include the state management objects within J2EE, the session life cycle, how sessions are identified and options for persisting session data within WebSphere Application Server.

For a complete discussion of session management implementation and configuration in WebSphere Application Server, refer to the IBM Redbook

WebSphere Application Server state management objects

State management in J2EE application servers is implemented via a single interface, *HttpSession*. This object is analogous to the ASP.NET *HttpSessionState* object; see “ASP.NET state management objects” on page 381.

In WebSphere Application Server, session state objects are managed by the *session manager*. The session manager is a component or function of the J2EE Web container, which executes within the application server process. The session manager implements the following functionality on behalf of Web applications executing within the container:

- ▶ Creating *HttpSession* objects.
- ▶ Generating the unique session identifier.
- ▶ Implementing the chosen session tracking mechanism.
- ▶ Storing and retrieving objects and session data within the session.
- ▶ Providing a mechanism for persisting sessions.
- ▶ Managing the session cache.
- ▶ Invalidating *HttpSession* objects.

Session state life cycle

On a logical level, the life cycle of a WebSphere Application Server *HttpSession* object closely mirrors that of an ASP.NET *HttpSessionState* object. Where the two technologies differ is in the implementation. See “Session state life cycle” on page 383 for a description of the session state life cycle within ASP.NET applications.

The life cycle of a session effectively begins when the first request is received from any given client. When this event occurs, the session manager component of the Web container will create an *HttpSession* object and generate a unique session ID for tracking purposes. The session persists across a number of requests from that user, during which time objects and data may be added to and retrieved from the session object as a means of preserving some sort of application state.

The session expires in one of two ways. In the first instance, a session can end when the client explicitly terminates the session, for instance by logging out of the application, in which case the session manager will invoke the session object's *invalidate* method to end the session. In the second instance, the

session expires because the session timeout interval has been exceeded. In WebSphere Application Server, each time a response is sent back to the client, the session manager updates the last accessed time of the session object. Periodically, the session manager will examine all the active session objects and invalidate those where the current time minus the last accessed time exceeds the session timeout value specified in the session manager configuration.

Session tracking mechanisms

The Servlet 2.3 specification that forms part of the J2EE 1.3 specification implemented by WebSphere Application Server provides for three session tracking mechanisms.

► Cookies

WebSphere Application Server provides support for tracking sessions using cookies. When enabled, a cookie containing the unique session ID for the client is generated. During the life cycle of the session, this cookie is passed back and forth between the client and server. The actual session data is stored on the server and is located using the session ID in the cookie.

Cookies in WebSphere Application Server can be configured to operate over secure connections (HTTPS) and can be restricted to a particular domain and virtual path. By default, the cookie is set to expire at the end of the browser session, but it is also possible to specify a Time to Live value for the cookie in accordance with the servlet specification.

► URL rewriting

It is also possible in WebSphere Application Server to track sessions by using the URL rewriting mechanism. URL rewriting works by encoding the session ID as a parameter to the URLs that are embedded within the returned HTML page.

In order to implement URL rewriting, additional coding effort is required since APIs for encoding the URL with the session identifier need to be used. Care also needs to be taken when designing the flow of the Web application to ensure that the encoded session ID is not lost. When using URL rewriting, it is not possible to use static HTML pages, because the session ID cannot be embedded within these.

► SSL sessions

SSL sessions are the final session tracking mechanism required by the servlet specification. This session tracking mechanism simply leverages a feature of the secure socket layer protocol to provide a session tracking mechanism. In this scenario, the SSL session ID that is created as part of the SSL handshake process that occurs when establishing a secure connection between client and server is used to track sessions instead of the WebSphere Application Server generated session ID.

Persisting a session

WebSphere Application Server offers three mechanisms for persisting session data:

- ▶ None

If this configuration setting is selected then the session objects are simply persisted in an in-memory session cache managed locally by the session manager that created them. Sessions are not shared among application servers in a cluster and are not persisted to any permanent storage medium, so, with this option selected, they will not survive an application server restart.

- ▶ Memory-to-memory replication

This session persistence mechanism enables sharing of sessions across a replication domain without the need for a remote database. This mechanism uses WebSphere Application Server internal messaging to publish and subscribe session state across the members of the replication domain.

Without going into specifics, a replication domain can be considered to be a collection of clustered application servers that communicate with each other in order to share data.

Sessions are shared between the replication domain members and stored in memory on each application server. If an application server fails, other servers in the replication domain are still able to service requests without loss of session state.

The replication domain and persistence policy are configured via the WebSphere Application Server administrative console and application or Web module level deployment descriptors.

- ▶ Database

The third session persistence mechanism employs a database as a persistent store for session data. When sessions are created and on each subsequent occasion that they are updated, they are serialized and stored in a remote database table. Thus, if an application server participating in a cluster fails, requests can still be serviced by other cluster members with no loss of session data.

As with memory-to-memory replication, database persistence can be configured via the administrative console and/or the application or Web module deployment descriptors. In addition to configuring the session persistence, it is also necessary to configure a remote database server and associated JDBC provider and data source.

9.3.3 Microsoft .NET session management

In this section, we provide an overview of state management with Microsoft .NET applications, or, more specifically, ASP.NET. The discussion focuses on the various objects that participate in state management. In particular, we focus on session state management by examining the life cycle of a session within ASP.NET applications, how session objects are identified and options for persisting session state.

ASP.NET state management objects

State management in ASP.NET is provided at four levels: application, session, page and request. At each level, a container object is available to allow the application developer access to state management facilities. Table 9-1 documents the various state management objects available in ASP.NET and their characteristics.

Table 9-1 ASP.NET state management objects

Level	Object	Life cycle	Scope
Application	HttpApplicationState	Created when first request is received by Web server. Destroyed when application shuts down.	Global to all sessions.
Session	HttpSessionState	Created upon initial user request. Destroyed when user ends session.	Global to all requests issued by the same user.
Page	ViewState	State maintained across successive calls for a page.	Limited to all requests for a specific page.
Request	HttpContext	State maintained for the life cycle of the request.	Visible to all objects associated with a request.

► Application state

The `HttpApplicationState` object is created when the Web server receives the first request for an application resource from a client. For every application running in an application server, there will be a separate instance of an `HttpApplicationState` object. The application state is only accessible within the context of the application that created it. Application state cannot be shared

across applications, even when they are executing within the same application server.

- ▶ Session state

The `HttpSessionState` object is created when the first request from a particular user is received by the application server. There will be an instance of the `HttpSessionState` object created for every user who submits a request for an application resource. The session state remains valid until either the user explicitly closes the session, or the session is invalidated either by timeout or abandonment. Each instance of an `HttpSessionState` object is associated with a particular client. Session state cannot be shared across clients.

- ▶ Page state

The `ViewState` object is useful where applications need to maintain the state of a page, or the controls that are embedded within it, across successive requests for a page. The state of the page is typically stored on the client, though it can be maintained on the server. The state is passed between the client and server and travels with the page.

- ▶ Request state

The `HttpContext` object differs from other state objects in that it has a short life cycle, existing only for the duration of a single request. It is also different in that it is visible to all objects that participate in the processing of an HTTP request.

ASP.NET session state

ASP.NET implements a simple and easy to use session state model which can be used to store objects and other application data across user requests. The store itself is implemented as a dictionary-based cache of object references. By default, the session state cache resides in memory within the Microsoft IIS process. This is known as *in process session state mode*.

In the Microsoft .NET Framework, there is a session state module that is analogous to the session manager within WebSphere Application Server. The session state module implements the following functions:

- ▶ Creating the *HttpSessionState* object.
- ▶ Generating unique session identifiers.
- ▶ Providing a session tracking mechanism.
- ▶ Storing and retrieving state information on behalf of Web applications.
- ▶ Implementing the chosen session persistence mechanism.
- ▶ Destroying the session.

The session state module is part of the Microsoft IIS application server process.

Session state itself is implemented in the *SessionState* class. This class provides for two collections of objects: the *Contents* and the *StaticObjects* collections. The *Contents* collection acts as a store for all variables that have been added to the session state as a result of executing the application code. The *StaticObjects* collection exposes all variables that have been added to the session state collection through the Global.asax file.

Session state life cycle

The session state life cycle in an ASP.NET application is very similar to that of a J2EE Web application, though, of course, the underlying implementation is different between the two technologies.

In ASP.NET, the life of a session state does not begin until the first item is added to the in-memory dictionary. Note, however, that when the first request is received for a user, the session state module will create a new session ID for that user and fire a *Session_OnStart* event. The session state remains active for subsequent requests for that user, with objects being stored and retrieved from the session state as required by the ASP.NET application.

The session state life cycle ends when the *Session_OnEnd* event is fired. This can occur in one of two ways, either by a user request to end the session, for instance by logging out of the application, or by the session timing out.

A brief discussion here of how the timeout mechanism works in ASP.NET may be of interest, since it differs from the J2EE model. When the first value is added to the session dictionary, an entry is made in the ASP.NET cache. This entry contains the ID for the session and is given a special expiration policy that is essentially a sliding expiration with the timeout value set to that of the session timeout. During the processing of each subsequent user request, the cache entry is updated, effectively by performing a read on the cache, which resets the sliding window. If the user does not make any further request within the session timeout period then the cache entry will expire, which will result in its removal from the cache. As a result, when the cache entry expires, it is implied that the session has timed out.

Session tracking mechanisms

In ASP.NET, sessions are identified by means of a 120 bit session identifier. The session ID is generated using the random number generator cryptographic provider, which returns a sequence of 15 randomly generated numbers. These numbers are then mapped to URL-encoded characters and returned as a string.

ASP.NET offers the following session tracking mechanisms:

- ▶ Cookies

This is the default session tracking mechanism within ASP.NET. In this mechanism, a cookie is generated for each user containing the session ID. This cookie is then passed back and forth between the client and server. In ASP.NET, it is possible to configure cookies to operate on a particular virtual path and over secure connections.

- ▶ URL rewriting

It is also possible to enable URL rewriting as a session tracking mechanism with ASP.NET applications. In this mechanism, the URL is modified by embedding the session ID within it. This mechanism provides an alternative session tracking mechanism whereby the application is accessed by clients who have disabled cookies in their browsers.

Persisting a session

Within ASP.NET, there are three *modes* available for session state persistence:

- ▶ In process mode

This is the default session state persistence mechanism within ASP.NET. This mode provides for persisting session state in memory within the Microsoft IIS process space. This mode, while it is the easiest to configure, provides no resilience, that is, the session state is local to the Microsoft IIS server in which it was created and does not survive an application server restart.

- ▶ State server mode

The state server mode enables a more resilient session state persistence mechanism since the session state is now stored on a remote state server, which enables the session state to survive an application server restart.

For an ASP.NET application to support this mode, it is necessary to write extra code to perform the serialization and deserialization of the session state to and from the remote server.

ASP.NET performs the serialization/deserialization of certain "basic" types using an optimized internal method. "Basic" types include numeric types of all sizes (for example `Int`, `Byte`, `Decimal`, etc.), `String`, `DateTime`, `TimeSpan`, `Guid`, `IntPtr` and `UIntPtr`.

ASP.NET will serialize/deserialize a session variable that is not one of the "basic" types using the `BinaryFormatter`, which is relatively slower.

Before you can use an out-of-process state method for managing and storing the session state, you must ensure that the objects defined and used by your ASP.NET application are serializable. Making an object serializable is usually a matter of adding the object class with the `<Serializable>` attribute. Only in

special cases can you build a custom serializer (in this case with additional coding effort) implementing the ISerializable interface.

See also:

<http://support.microsoft.com/default.aspx?scid=kb;EN-US;q325056>

for a note about load balanced Web farm scenarios.

A remote state server must also be configured as the session store. For the state server persistence mode to work in ASP.NET, state service must be running on that machine. The final step is to configure the Microsoft IIS instance, via the Web.config file, to use the state server mode.

► **SQL server mode**

This is the third and most resilient session state persistence mode. In this mode, session state is persisted to an SQL server database. In SQL server mode, session state can also be configured to work in a failover cluster. A failover cluster is essentially two or more identical application servers that can store their session data in a remote SQL server database. Should one application server fail, requests can still be serviced by another machine in the cluster without loss of session state.

As with the state server mode, there is additional coding effort required to perform the serialization of session state, and additional configuration of the Microsoft IIS application server. Also, it is necessary to create the tables within the SQL server to hold the session state, but scripts are provided for this task.

9.3.4 Considerations

In this section, we examine some of the key considerations that need to be addressed in sharing state between Web applications executing in WebSphere Application Server and Microsoft .NET. In particular, we examine the following:

- How do we establish a relationship between a user session in WebSphere Application Server and a user session in Microsoft .NET?
- What mechanisms are available for sharing session data between the two technologies?
- How can the session life cycle be extended across the technology boundary?
- Can forwarding and redirecting of requests be implemented across technologies?

Session object mapping

This section discusses possible solutions for mapping a user session in WebSphere Application Server to a corresponding user session in a Microsoft .NET application.

When developing a Web application from the ground up, there are seldom good business reasons why you would choose to host a Web application across both WebSphere Application Server and Microsoft .NET technologies. In the scenario where a new application is being developed, one would simply choose the best technology stack for the application and focus development in that direction. However, it is not inconceivable that certain situations may arise, such as in a mergers and acquisitions scenario, where there may be an immediate business need to provide Web interoperability in a heterogeneous environment. In the long run, however, it is imperative that a sound migration strategy be designed and implemented to eliminate having to provide these interoperability fixes.

Earlier in this chapter, we outlined what session tracking mechanisms are available in WebSphere Application Server and Microsoft .NET and briefly discussed some of the details of each implementation. This section extends these models to discuss how one might provide for the tracking of sessions in a heterogeneous environment. In particular, we discuss how session tracking across technologies might be implemented by leveraging existing technologies, such as cookies or URL rewriting, and also how we might implement our own custom session tracking mechanisms.

Note that this discussion focuses purely on ways in which applications might share session identifiers in a heterogeneous environment. Actual sharing of session data is discussed in “Sharing session data” on page 390.

Cookies

As discussed earlier in this chapter, both technologies provide for session tracking by using cookies. It therefore makes sense to leverage this ability and the client side capabilities of the browser to use cookies as a mechanism to exchange session IDs between Web applications running in WebSphere Application Server and Microsoft .NET environments.

Figure 9-8 on page 387 illustrates how cookies can be used as a mechanism to track sessions across applications in a heterogeneous environment.

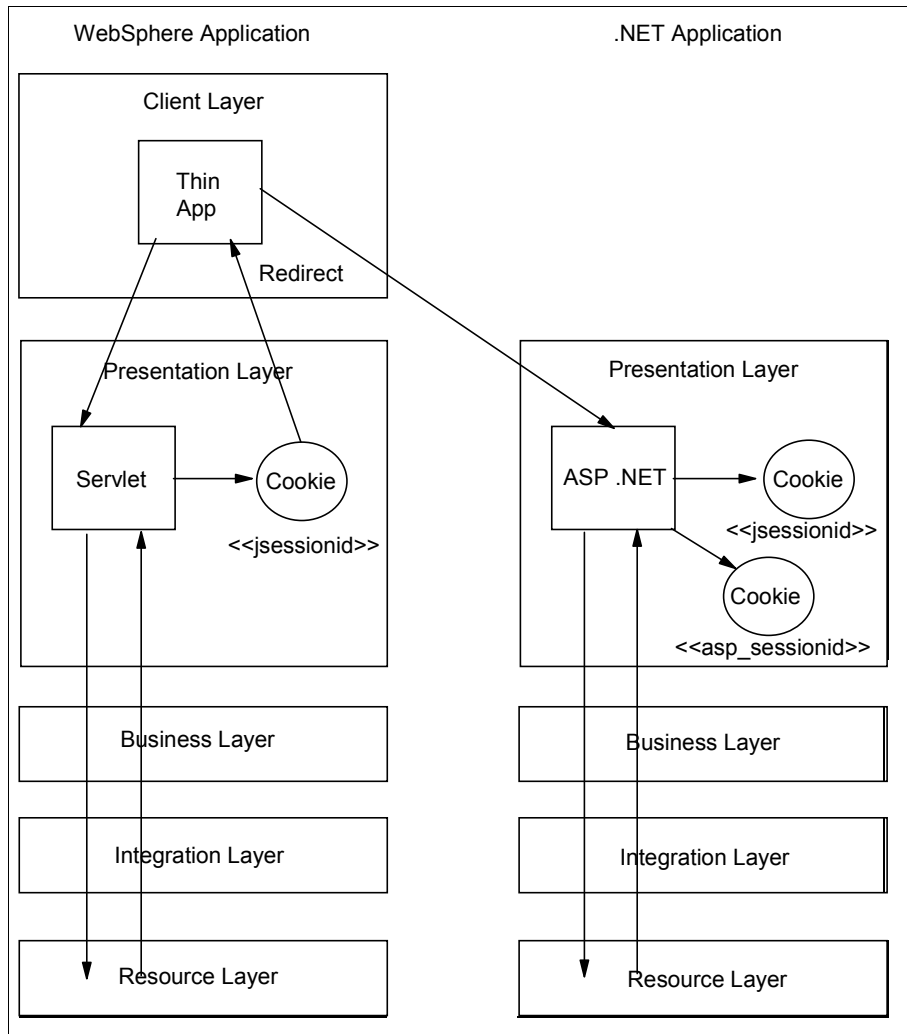


Figure 9-8 Session tracking with cookies in heterogeneous environment

In the above scenario, the client interacts with both a J2EE Web application running in WebSphere Application Server and an ASP.NET application running in a Microsoft .NET Framework. While it is assumed that the Presentation layers in both technology stacks will interact with other application layers, the details of those interactions are not shown here, for reasons of clarity.

The client initiates a session in the WebSphere Application Server environment by sending a request for some Web resource. The WebSphere Application Server creates a session for the client and delegates the request to other

application layers for processing of the request. In this scenario, it is assumed that in order to completely satisfy the request, it is necessary to execute some business logic residing on the Microsoft .NET technology stack. Therefore, the servlet controlling the application flow sends a redirect to the client, passing the WebSphere Application Server session identifier back to the client in the form of a cookie. The client then redirects the request to the specified resource in the Microsoft .NET application where an ASP.NET session is created on behalf of the client and a session ID assigned.

Note that we use a redirect in the solution since it is not possible to forward to a resource external to WebSphere Application Server. Care must also be taken when the ASP.NET application resides in a different domain from the WebSphere Application Server to ensure that the domain of the cookie is explicitly set. Failure to set the domain will result in the cookies not being sent as part of the redirect.

In this way, we have managed to establish a relationship between a session residing in an WebSphere Application Server application and a session residing in an ASP.NET application. Having done so, the cookies containing the session identifiers for each application can be passed back and forth between the environments.

URL rewriting

URL rewriting is the other existing session tracking mechanism common to both technology stacks. However, it is not a suitable mechanism for exchanging the session identifiers since the implementation of how the session identifiers are embedded into the URL differs, resulting in HTTP errors when attempting to redirect between the two technology stacks. For instance, when WebSphere Application Server encodes the URL with the session identifier, the resulting URL looks something like this:

<http://myHost/myServlet;jsessionid=0000oqsZo6fsRbPue0yZYiGuwFl:-1>

Whereas in ASP.NET the session identifier is encoded in the URL in the following manner:

[http://myHost/myApp/\(5jddu0nkyjbqfv45ont2ad55\)/myASP.aspx](http://myHost/myApp/(5jddu0nkyjbqfv45ont2ad55)/myASP.aspx)

Thus the two encoding mechanisms are totally incompatible and URL rewriting should not be considered an option.

Custom session tracking mechanisms

A third, but less attractive option is to implement a custom mechanism for tracking sessions in the heterogeneous environment. This has drawbacks in that it requires writing additional code.

Options for implementing custom session tracking mechanisms include:

- WebSphere Application Server as a façade to Microsoft .NET

Figure 9-9 illustrates the concept of the façade. In this scenario, the two environments are fronted by a WebSphere façade that handles the complexities of session tracking on behalf of the WebSphere Application Server and Microsoft .NET applications.

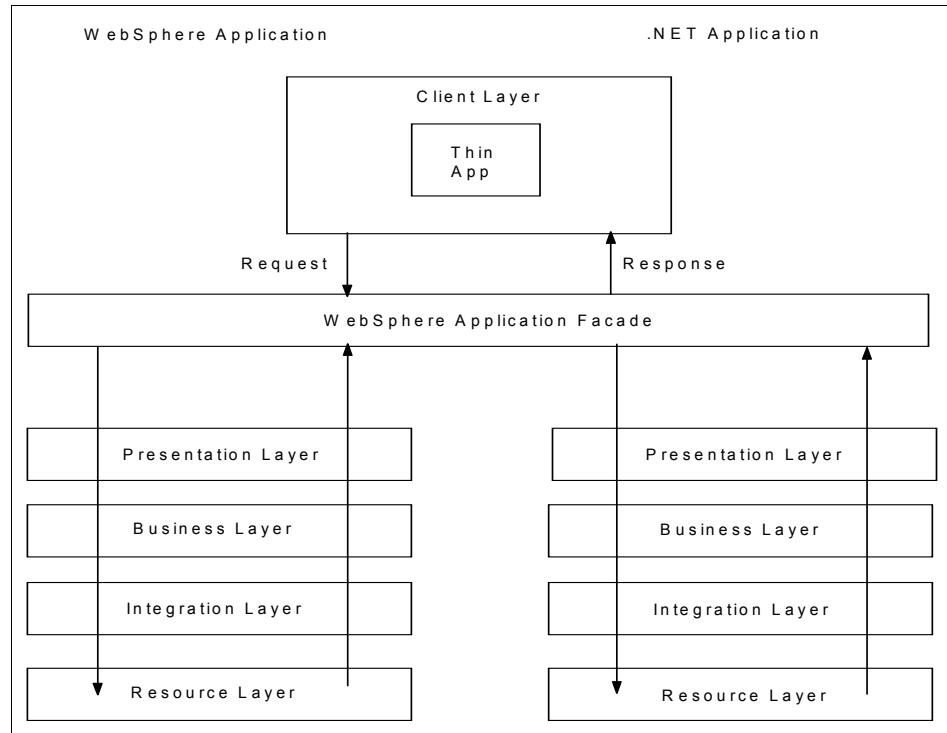


Figure 9-9 WebSphere Application façade

We have not actually attempted an implementation of this scenario in this book. However, there is a technology in J2EE that lends itself to this type of scenario; this technology is called filter servlets and is part of the servlet 2.3 specification.

A filter is a reusable piece of code that can transform the content of HTTP requests, responses, and header information. Generally, filters do not create a response or process a request as an `HttpServlet` would. Instead, they modify or adapt requests for a resource and modify or adapt responses from a resource. The type of functionality supported by filter servlets includes:

- Accessing a resource before a request to it is invoked.
- The processing of a request for a resource before it is invoked.
- The modification of request headers and data.
- The modification of response headers and data.

Thus it can be seen that filters are well adapted to performing the kind of tasks required to implement the façade illustrated above.

► Passing the session identifier as a parameter

In this scenario, whenever the redirect is issued to pass the user from one Web application to another, the session identifier is added to the redirect URL as a parameter. This implementation would require additional control logic in both Web applications to extract the session identifier and store it. The most obvious place to store it would of course be in the session object of the Web application to which it was passed.

► Passing the session identifier as an HTTP header

The servlet filter would modify the HTTP request and add a new HTTP header key-value pair with the session identifier. On the application server's side, the application has to extract the HTTP header value to get the session identifier.

► Embedding the session identifier as a hidden field

In this solution, the session identifier is embedded in the HTML response returned to the client as a hidden field in a form. When the client makes the next request, the session identifier is returned to the server as a parameter. While this at first may seem a relatively straightforward way to implement interoperability, it is in fact highly undesirable. This is because, to implement this option, you would have to add the form and hidden field in your pages in both Web applications. Again, it would also be necessary to implement some controlling logic in both WebSphere Application Server and Microsoft .NET to handle the embedding and extraction of the session identifier, storage and retrieval of the session identifiers.

Sharing session data

In this section, we explore the following mechanisms for sharing session data in a heterogeneous environment:

- Database
- Messaging point-to-point
- Messaging publish/subscribe
- Push

Once again, when reading this section, it is important to keep in mind that while the solutions can be implemented, one has to consider whether it is desirable to do so, since the alternative is often to eliminate interoperability issues altogether by carrying out a migration from one technology stack to the other.

Database

In this scenario, illustrated in Figure 9-10, session data is stored in a relational database and shared between a WebSphere Application Server and a Microsoft .NET Web application.

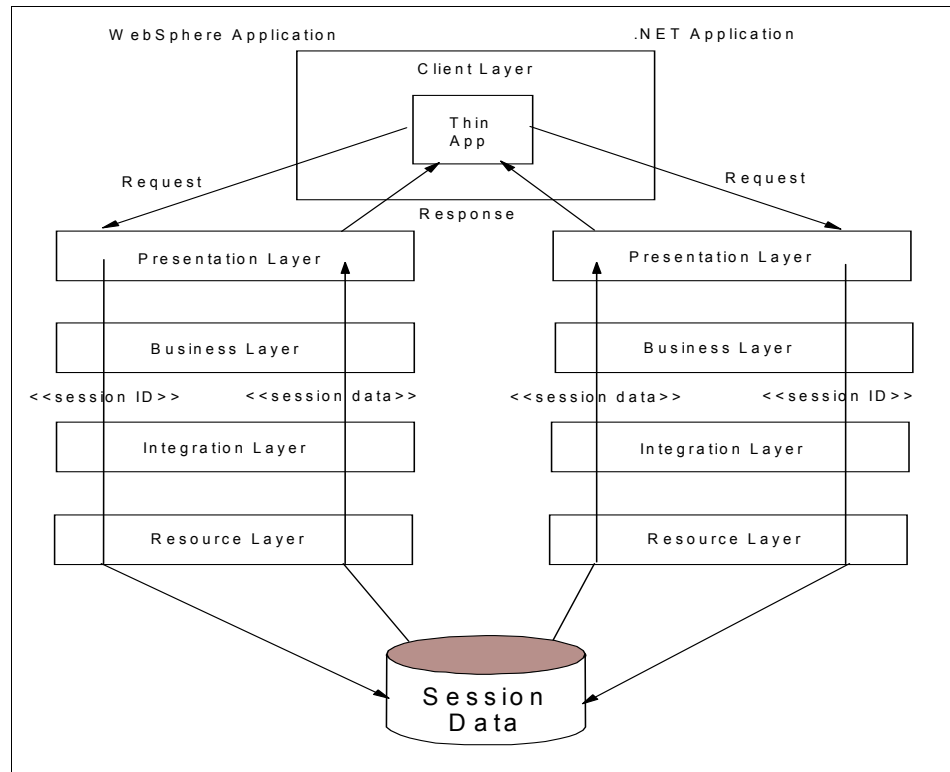


Figure 9-10 Sharing session data using a relational database

To access the data, both Web applications need some common key that maps to the data in the database table. This key could be exchanged between the two Web applications using one of the mechanisms discussed in “Session object mapping” on page 385.

When a request is received from a client who is not in session, the Web application must create a session, create a unique key for the session and persist the session data into the database. The key can then be sent back to the

client, probably in a cookie so that the session data can be retrieved and updated on subsequent calls by whichever application is processing the request.

While this solution, at first, appears to be a relatively straightforward mechanism for sharing session data, there are a number of problems that need to be addressed in actually implementing the solution:

- ▶ Session tracking

In order for each Web application to be able to retrieve the session data, it will be necessary to implement some common session tracking mechanism.

- ▶ Data serialization and deserialization

In order to write the session data to and retrieve it from the database, it is necessary to serialize and deserialize the data. Both Java and the Microsoft .NET languages provide APIs for doing this. However, the problem that needs to be addressed is how to represent a Java object in a form that can be understood by a .NET application and vice versa.

- ▶ Implementing session persistence

As we have seen earlier, both technologies support persisting session data to a relational database. However, it may not be possible to leverage the mechanisms built into WebSphere Application Server and Microsoft .NET. In this case, it will be necessary to implement a bespoke persistence solution.

- ▶ Performance

When implementing a bespoke session persistence mechanism, great care has to be taken to ensure that application response times are not adversely affected. Both WebSphere Application Server and Microsoft .NET provide database persistence mechanisms that have been heavily optimized for the environment that they service, but even these can increase response times by 20% to 30%. It is unlikely that any bespoke development that has to operate across technology stacks will be able to better this.

Given the problems discussed above, this solution becomes less attractive when compared to migrating your Web application to a single technology stack.

Messaging point-to-point

In this scenario, session data is shared between Web applications by using message queueing middleware, such as IBM's WebSphere MQ, to synchronize session data; Figure 9-11 on page 393 illustrates this scenario.

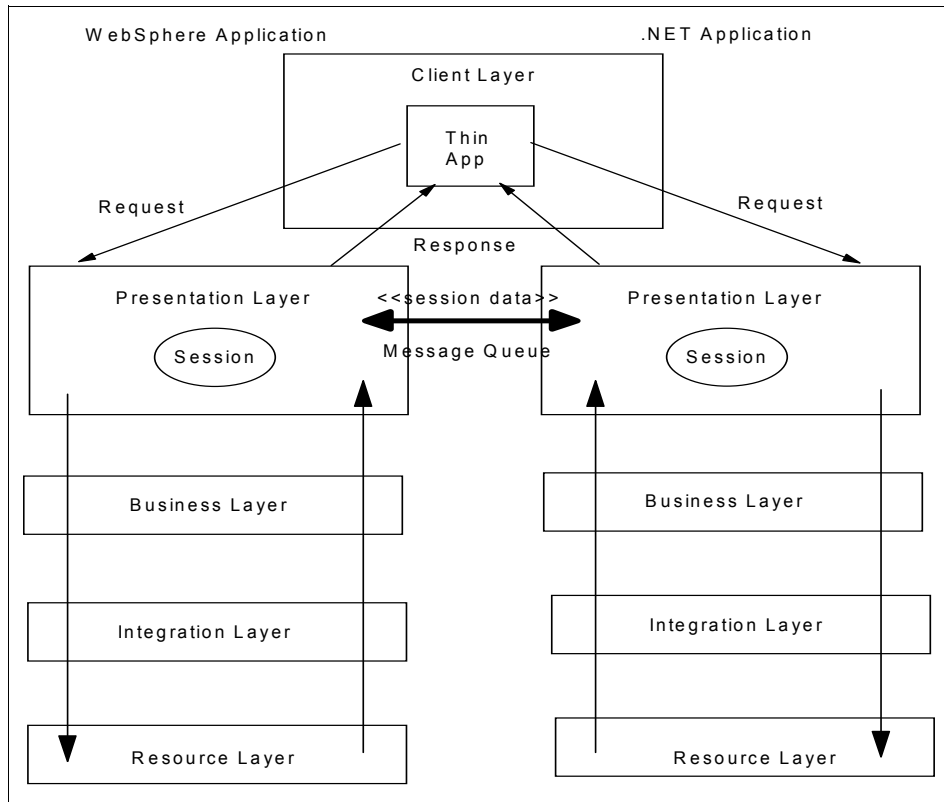


Figure 9-11 Sharing Session data using message queuing middleware

When the first request is received from a user, the receiving Web application must create the session on behalf of the user and assign a session ID. Each time any action is performed upon the session, such as creation, updating or deleting, the session has to be synchronized between the Web applications using the queuing mechanism.

While this solution may provide a mechanism for sharing session data in a heterogeneous environment, it suffers from many of the problems that were highlighted in the previous section on sharing session data using a database, such as object serialization and session tracking issues. Also, there is most likely going to be a considerable amount of development effort to implement the synchronization and to install and configure the messaging middleware.

Messaging publish/subscribe

Point-to-point messaging fails in load balanced environments, where multiple WebSphere and .NET application servers are listening for the user's request. In such a situation, the publish/subscribe messaging pattern can be a solution. One

application server distributes the session update information to a topic, instead of a single queue. On the other end, all the other application servers that have subscribed to the topic receive the session update information.

Session invalidation has to be solved by using publish/subscribe pattern also, to make sure that all servers invalidate the session.

Push

This scenario is similar to the messaging scenario in that session data is pushed between the Web applications to ensure that they both have a local, but consistent view of the session data.

One way in which session data might be pushed between WebSphere Application Server and Microsoft .NET is by using HTTP POSTs. For a full discussion of how session data may be propagated between technology stacks, see 9.4, “Data propagation” on page 395

Session life cycle

Sessions exist for a defined life cycle; they are created when a user makes an initial request to a Web application. They are updated, stored and retrieved in some storage mechanism, whether persistent or not over consecutive requests from the user. Eventually, they are destroyed either by explicit user action or by the runtime environment when some predetermined event, such as a timeout, occurs. Earlier in this section, we examined how the session life cycle is implemented in both WebSphere Application Server and in Microsoft .NET.

When implementing session state management in heterogeneous environments, consideration must be given to how the session life cycle is properly enforced. This is particularly important where a session is replicated across both environments. The design of the solution must address, for instance, how session creation and invalidation are enforced across the two technology stacks. For instance, if a session is invalidated in a WebSphere Application Server Web application then it must also be invalidated in the Microsoft .NET Web application. Any solution must address such issues.

Forwarding and redirecting

Consideration also has to be given as to how requests and responses are passed between heterogeneous Web applications. 9.4, “Data propagation” on page 395 provides a discussion of the issues surrounding forwarding and redirection.

9.3.5 Recommendations

Typically in a section of this nature, one would expect to see some recommendations as to when and how to best apply the solutions which were discussed earlier. However, this time we are going to approach things from a slightly different viewpoint.

This section has illustrated the problems in managing session data across a heterogeneous environment. It has proposed solutions, some of which have been implemented, others of which are untested theory. There can, however, be only one recommendation arising from this chapter and that is that, when presented with a scenario which requires Web interoperability, one should not consider which of the solutions presented in the chapter is the best. Rather, one should consider whether integrating the technologies in the first place is the best option, or whether it is more appropriate to perform a structured migration of one environment to the other.

9.4 Data propagation

Data propagation is the movement of data from one medium to another. It is often a necessary component of Web-interoperability in an environment containing Web applications running on WebSphere Application Server and ASP.NET applications running on an IIS application server. This section discusses data propagation, as well as some of the problems associated with data propagation in a coexistence scenario, lists some items to consider, and gives some examples of propagating data between a WebSphere and ASP.NET application.

9.4.1 Problem definition

Under certain circumstances, it may be necessary to share data between existing Web applications. This becomes an interesting problem in situations where the applications reside on different application servers and employ different Presentation layer technologies. Such is the case in an environment where Web interoperability is required between Java-based applications running on IBM WebSphere Application Server and ASP.NET applications running on Microsoft IIS. An example of data propagation in an environment such as this is a purchasing application which runs on WebSphere and needs to interact with an catalog item selection application running on IIS. The WebSphere purchasing application is a typical shopping cart style Web application. What makes this application different is that it must interact with an item catalog system located on an external IIS system. These systems must interact in a seamless manner. When the purchaser requests to browse an item catalog located on the IIS server, he is directed to that server. The WebSphere application needs to

propagate data such as the catalog name and item category to the IIS server so that it can display the correct information to the user. The user selects an item or possibly many items from the ASP.NET item catalog. These selections and all related information must then be propagated back to the original application.

A simpler scenario would be to simply propagate data from one application to the other, not returning to the original application. An example of this scenario, using portions of the previous example, would be to start with the item catalog on the ASP.NET system, select items, and then transfer to the WebSphere system to carry out the purchase of those items. In this scenario, data propagation would occur only once in the transfer of items from one application to the other.

These are only two examples of data propagation in an environment where WebSphere and .NET must coexist and interact. Also, this interaction is to occur at the Presentation layer. Keep in mind that these examples do not include session/state management or security. These topics are covered in other sections of this chapter.

9.4.2 Description of the problem

The problem surrounding Presentation layer data propagation can be made into a simple abstraction. At an abstract level, data propagation is very straightforward. Data needs to be copied by some Web application and transported to another Web application, Figure 9-7 on page 377 shows a more abstract view of the problem.

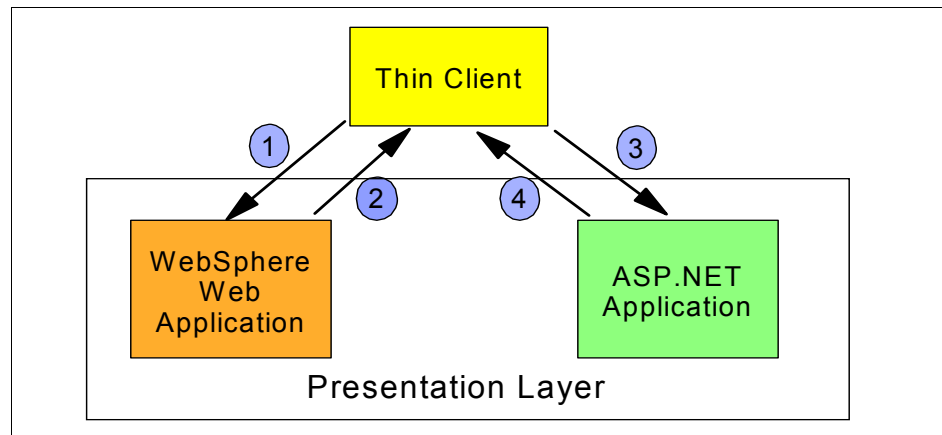


Figure 9-12 Data propagation from application servers with a thin client intermediary

In Figure 9-7 on page 377, we have two different application servers servicing a single client.

1. The client sends a request to the WebSphere Web application.
2. The Web application responds.
3. At this point, the thin client has the data it needs sent to the ASP.NET application. It propagates this data to the application with a request.
4. The ASP.NET consumes this data and produces a response.

9.4.3 Considerations

In designing and implementing a solution for the scenario, there are several considerations that should be taken into account. This section presents a discussion of these considerations.

Data transport

The data transport mechanism is the method used in transporting data from one Web application to another. There are only a few options available to transport data from one Web application to another that do not require an direct connection between the Web applications. These are:

- ▶ URL redirection

URL redirection is a method of notifying a thin client, typically a browser, that it should invoke a different URL than the one it requested. Redirection requests can be sent by a Web application with a single URL. The thin client, if it chooses to perform the redirect, will invoke the URL.

- ▶ Form-based

Form-based transport is the use of standard HTML forms to transport data from a Web application server to another. Using Figure 9-7 on page 377 to demonstrate, the client makes a call to the WebSphere Application Server for some content. The application server responds with an HTML form. The action attribute on the form directs the form to be submitted to a URL hosted by the ASP.NET application server. When the form is submitted, the data it contains is sent to the ASP.NET server.

- ▶ JavaScript URL construction

In a JavaScript-enabled thin client such as Netscape Navigator or Microsoft Internet Explorer, a URL can be constructed within the script at runtime and the client can be directed to the URL.

- ▶ HTTP request forwarding

HTTP request forwarding is the forwarding of a HTTP request to a secondary URL. Forwarding is currently only available in WebSphere Application Server. In addition, the current implementation only allows URL forwarding within the same Web application. Due to these restrictions, request forwarding is not

considered as a possible transport mechanism. For more information on WebSphere/J2EE request forwarding, see the RequestDispatcher interface at:
<http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/RequestDispatcher.html>

Table 9-2 Comparison of data transport mechanisms

Method	Advantages	Disadvantages
URL Redirection	<ul style="list-style-type: none"> ▶ Very simple to implement under both .NET and WebSphere. ▶ Can be used to redirect a client to any URL. ▶ Supported by most newer thin clients/browsers. ▶ Involves no direct interaction between application servers. The thin client serves as an intermediary. ▶ Data can be manipulated and validated before it is mapped into a redirection URL. 	<ul style="list-style-type: none"> ▶ May not be supported by the thin client. ▶ Some clients can be configured to not allow redirection. ▶ All data must be included on the URL string. The maximum length of a URL string varies for clients and servers. ▶ Redirection uses the HTTP GET command. Some applications may be designed to expect data from a HTTP POST command. ▶ The application server requesting the redirect must generate the correct URL to give to the client. ▶ Requires an additional connection to the originating application server.

Method	Advantages	Disadvantages
Form-based	<ul style="list-style-type: none"> ▶ The thin client handles passing data to the secondary server. No extra code is required on the primary server. ▶ Can be used to send multi-part data, including files. ▶ Does not require multiple trips to the application server. ▶ Not limited by URL length. ▶ Supported by most thin clients/browsers. ▶ Hidden elements may be used to send data to the secondary server. 	<ul style="list-style-type: none"> ▶ Does not occur automatically. The client must submit the data to the secondary server. ▶ Originating application server does not validate data before it reaches the secondary server.
JavaScript URL Construction	<ul style="list-style-type: none"> ▶ Server interaction is not required. ▶ Reduction in network traffic. ▶ Very dynamic. ▶ Supported by major browser vendors. 	<ul style="list-style-type: none"> ▶ Not all thin clients support JavaScript. ▶ More difficult to implement. ▶ JavaScript often does not function the same way for all thin clients. A script that runs on one may not run on another.

Data types

There are several issues regarding the propagation of data between Web applications using the methods discussed above. The most glaring issue is generally that only character data may be propagated between Web applications. All numeric values and binary data must be converted to a character string representation before it can be propagated between Web applications using the methods above. The HTTP protocol also allows direct binary transmission to occur if a client accesses the source of the data. For example, accessing a graphical image file is a direct binary transmission of data over HTTP. However, this type of data propagation is not what this section discusses. Instead, this section discusses data propagation in relation to interfacing two separate Web applications.

In addition to the character data only restriction, URL redirection and JavaScript URL construction must adhere to the additional restriction imposed by RFC 1738. This RFC specifies restrictions for URLs. In a nutshell, URLs should contain only a subset of characters in the US-ASCII character set. Fortunately,

this restriction can be thwarted by encoding the URL. Encoding takes those characters outside of the acceptable range and encodes them with characters that are acceptable to use in a URL. Encoding is a fairly simple process. Characters that are outside the valid range for URLs are encoded by prefixing them with a % symbol, followed by their hexadecimal code. For example, a space character (ASCII value 32) is encoded as %20. Some encoders may also use other special characters when encoding data. For example, the java.io.URLEncoder uses the + (plus) symbol for encoding space characters.

Since encoding is often required when forming a URL, both WebSphere and the Microsoft .NET Framework contain classes for encoding URLs. See Example 9-1 for an example of using URL encoding techniques from Java and the .NET Framework.

Example 9-1 URL data encoding in Java (WebSphere) and C# (.NET Framework)

Java example:

```
import java.net.URLEncoder;

...
String myUrl = "http://myserver/myapp/myservlet?data=";
myUrl = myUrl + URLEncoder.encode("my data");
// Encoded url is: http://myserver/myapp/myservlet?data=my+data
...
```

C# Example:

```
using System.Web; // Must also add a project reference to System.Web.dll

...
String myUrl = "http://myserver/myapp/myaspnet?data=";
myUrl = myUrl + HttpUtility.UrlEncode("my data");
// Encoded url is: http://myserver/myapp/myaspnet?data=my+data
...
```

You may have noticed in Example 9-1 that the entire URL is not encoded. If the entire URL were encoded, the thin client would not be able to process it correctly. For example, `http://myserver` becomes `http%3a%2f%2fmyserver`. As you can see, using this as a URL will not cause the correct page to be loaded by a browser. Therefore, instead of encoding the entire URL, only the parameter data should be encoded.

URL parameters follow the base URL and potential path info. Parameters follow an initial question mark and are of the format *parameter=value*. Additional parameters are separated by an ampersand. For example, this could be `http://myserver/myapp?parm1=value1&parm2=value2`. The value portion of these parameters should be encoded to ensure the data they represent is propagated correctly. It is also good practice to only use parameter names that do not require encoding.

Note: Handling of encoded URLs varies for thin clients (Web browsers): in some cases, a URL that works with one thin client may not work with another thin client. The encoding practice used by this book is supported by most current browsers.

The WebSphere J2EE Java `HttpServletResponse` interface contains a method named `encodeRedirectURL()`. So why not simply use this method to encode in one step? The reason is that most J2EE implementations of this method, WebSphere included, do not work quite as one may expect. They do not encode the actual parameter data. Instead, this method is typically used in situations where URL rewriting is used for session management. This method simply adds an additional encoded parameter to allow URL rewrites to persist over a redirection request. Usage has not shown it to perform actual encoding of URL data. To ensure proper encoding of parameter data, use the method above.

Data format

Propagating data from one Web application to another is very similar to calling an API from a normal program. The callee must construct a parameter list and parameter data such that it is what the caller expects. Proper formatting of data is very important. If a Web application is expecting a parameter containing a string in numeric format, it may fail if it receives badly formatted data.

Proper formatting of data is both the responsibility of the calling Web application and the application being called. The caller should make certain it has both parameters and data in the correct format. The end point application should make certain it can produce meaningful error messages if it does not get the data it expects. At the heart of this issue is arguably the biggest shortcoming regarding Web application interoperability: the lack of a defined, published interface into Web applications built in this manner. The end point application can change its URL, parameter lists, and data format and all clients coded to use that format no longer produce the correct behavior.

Other propagation methods

This section is intended to discuss Web application to Web application data propagation issues. Generally, the techniques discussed above are used in the following situations:

- ▶ Situations where Web applications must be integrated very quickly.
- ▶ Applications that reside on the Internet and must function over firewalls.
- ▶ Situations where a secondary Web application is used, but there is no direct agreement with the Web application provider. An example of a situation like this would be a Web application that links to another Web application to

provide a service such as driving directions or weather information. Web applications using a provider like this typically have no control over the level, quality, and availability of the service. They also have no control over changes made to the interface of the service.

Other sections of this book discuss many other methods for providing data propagation in a Web-interoperability scenario. Each of these methods has its limitations and considerations. Most of these limitations center around one of the three issues above. For example, Web Services function over Internet firewalls and provide a standard interface and data types, but may not allow quick and easy integration of applications. A shared database will allow information sharing but will not work if the application servers are separated by an Internet firewall. Other technologies and solutions share similar issues.

9.4.4 Solution model

This section discusses the design and implementation of an example solution for data propagation over WebSphere and ASP.NET applications. Two approaches will be covered: URL redirection and form-based integration. The JavaScript URL construction approach is left to the reader for further investigation.

Solutions to the problem are based upon a simple average calculator application. The application provides a very limited set of functionalities. The reason for this is to keep the example as simple as possible, so that the objective of how to integrate WebSphere Application Server and Microsoft .NET Web application artifacts is not obscured by the complexities of our chosen scenario.

In our solution, the average calculator provides only the ability to calculate the average of two arguments and display the results.

URL redirection solution

The URL redirection solution, as shown in Figure 9-13 on page 403, shows the steps involved in using redirection to propagate data from an IBM WebSphere Web application to a ASP.NET application running on IIS. The steps involved to produce this interaction are as follows:

1. The thin client requests a static page from the IBM HTTP Server. The server returns the page to the client.
2. The user enters data into the form and submits it back to the IBM HTTP Server. The server examines the URL and determines the request must be forwarded to WebSphere Application Server.
3. WebSphere Application Server gets the request, processes it, and generates a redirect request. The Web application must repackage the data and build a URL to direct the thin client to the application running on IIS. The URL is

generated and a redirect request is handed to the IBM HTTP Server. The HTTP Server sends a redirect request to the thin client.

4. The thin client performs the redirect. The Microsoft IIS server is contacted and data is propagated within the URL it is given.
5. IIS examines the URL context and directs it to an ASP.NET application. The ASP.NET application receives the parameter data that has been propagated in the URL. It validates the data and generates a reply in the form of HTML content. The reply is forwarded to IIS.
6. IIS receives the reply and it is returned to the thin client.

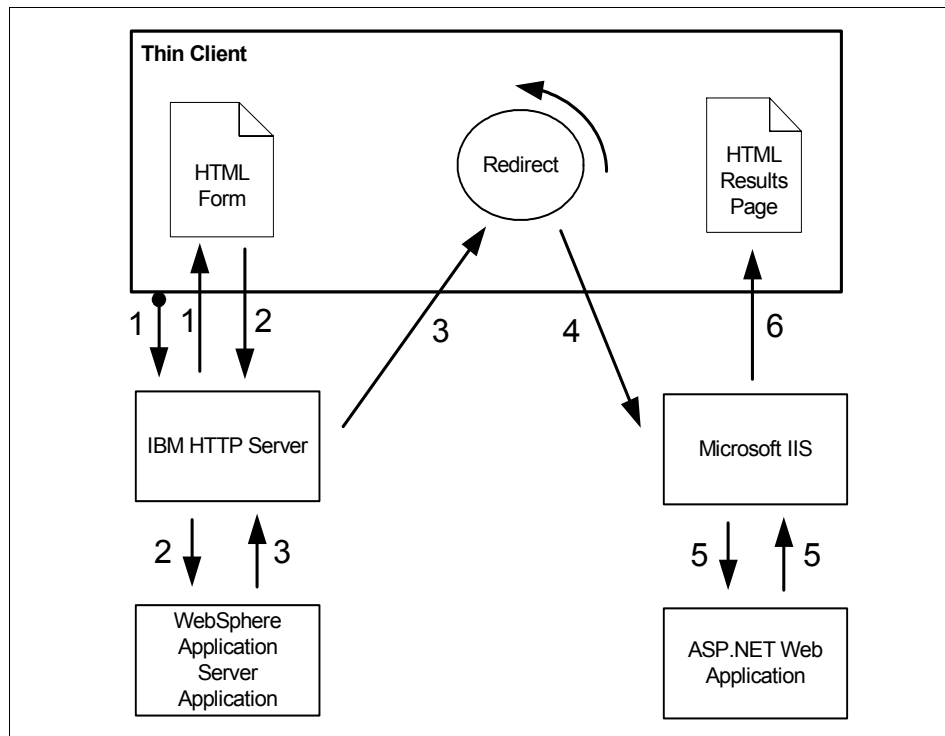


Figure 9-13 URL redirection: WebSphere to Microsoft ASP.NET

Tip: Steps 5 and 6 in Figure 9-13 could also be a redirection request back to WebSphere Application Server. This is ideal in many situations since it may help to keep all output logic within a single application server, thus making it simpler for all pages within an application to maintain the same look and feel.

Form-based propagation solution

In the URL redirection approach, we chose to go from IBM HTTP Server/WebSphere Application Server to Microsoft IIS/ASP.NET. To demonstrate our form-based solution, we will start with the Microsoft server and propagate data to the IBM server. This solution involves the following steps.

1. The thin client contacts an Web application through IIS. IIS examines the URL and determines it is an ASP.NET Web application. The request is forwarded to the ASP.NET application.
2. The ASP.NET Web application produces HTML content containing a form. It sets the action attribute on the form to direct to an application running on IBM WebSphere Application Server.
3. The user enters data into the form and submits it. The thin client sends the request and form data to the IBM HTTP Server. The HTTP Server examines the request URL and forwards the request and data to the WebSphere Application Server.
4. The application server processes the data and generates a response. The response is forwarded to the HTTP Server.
5. The HTTP Server returns the response to the thin client.

Tip: As with the URL redirection solution, steps 4 and 5 in this solution could process the data, package the solution, and redirect it to the ASP.NET application. This would ensure all content generation is performed by a single application server.

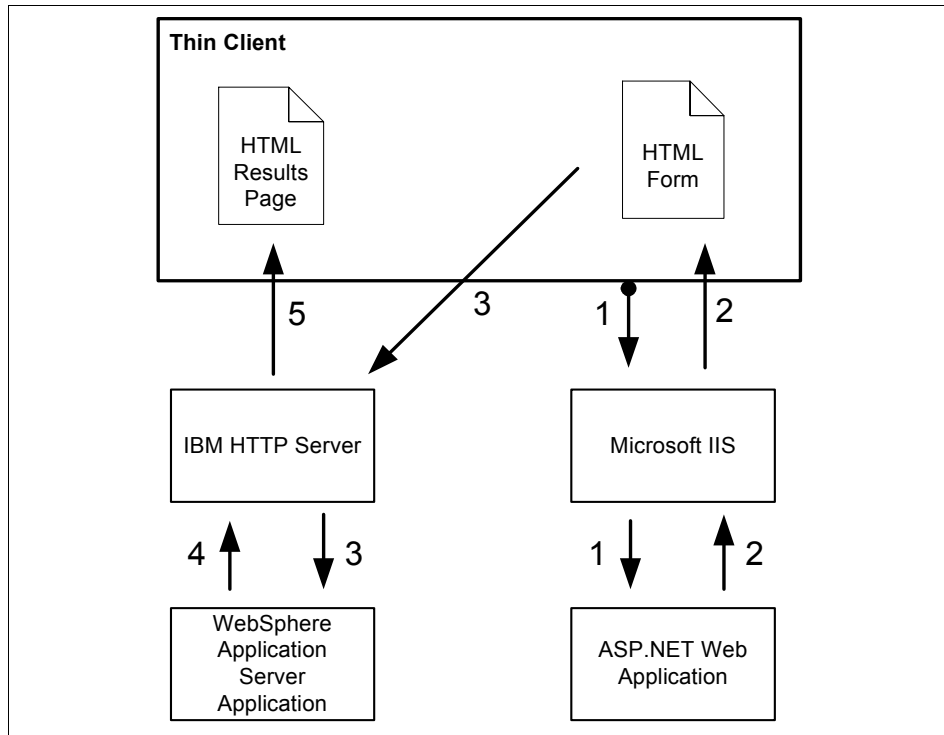


Figure 9-14 Form-based propagation from ASP.NET to WebSphere

9.4.5 URL redirection implementation

In keeping with the theme of the book, a simple calculator example was implemented to show a URL redirection solution between WebSphere Application Server and ASP.NET. The application is a simple calculator that takes two numbers, calculates the average, and returns the request to the client. This solution involves the following steps:

1. Create a WebSphere application to process form data and the redirection request.
2. Generate a WebSphere static HTML page.
3. Publish the WebSphere HTML and servlet.
4. Create a ASP.NET application to process the WebSphere request.
5. Publish the ASP.NET application.

Creating the servlet

Use IBM WebSphere Studio Application Developer to create a new Enterprise Application Archive (EAR). Name the Enterprise Application Archive WASWebInteropEAR. Create a Dynamic HTML project named WASWebInterop. It should reside within the Enterprise Application you just created. Dynamic HTML projects can contain servlets, JSPs, and static HTML pages. The next step is to add a servlet to the project. We named our servlet WS2NETAvgCalcRedirector. Naming is very important because, by default, the name of the servlet is used to reference the servlet from the client.

Example 9-2 contains the code for implementing the redirection servlet. Note the repackaging of URL parameter data and the call to `sendRedirect()` on the `HttpServletResponse` object. The URL that is constructed directs the thin client to an ASP.NET page running a server. In this example, both IIS and WebSphere Application Server are running on localhost. However, the IBM HTTP Server is using port 9080, while IIS uses the default port 80 for HTTP. In a normal environment, these would be different servers entirely, so the URLs would need to match.

Example 9-2 WS2NETAvgCalcRedirector.java

```
package redbook.coex.webinterop.ws2net.presentation;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.URLEncoder;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class WS2NETAvgCalcRedirector extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        doGetPost(req, resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        doGetPost(req, resp);
    }

    // doGetPost handles both GET and POST requests
    public void doGetPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        try
```



```

    {
        String arg1 = req.getParameter("WASArg1");
        String arg2 = req.getParameter("WASArg2");
        // Validate parameters (throws exception if bad)
        Double.parseDouble(arg1);
        Double.parseDouble(arg2);
        // Build the redirect to the ASP.NET page.
        StringBuffer netURL = new
StringBuffer("http://localhost/NETWebInterop/AvgCalculator.aspx?");
        // Parameters for the ASP.NET page are named ASPNETArg#
        netURL.append("ASPNETArg1=");
        netURL.append(URLEncoder.encode(arg1));
        netURL.append("&ASPNETArg2=");
        netURL.append(URLEncoder.encode(arg2));
        resp.sendRedirect(netURL.toString());
    }
    catch (NumberFormatException nfe)
    {
        // Arguments were not numeric, display a message
        PrintWriter pw = resp.getWriter();
        pw.println("<HTML><BODY><H2> ERROR: Arguments are not numeric.
</H2></BODY></HTML>");
        pw.close();
        return;
    }
    catch (NullPointerException npe)
    {
        // Arguments were not specified, display a message
        PrintWriter pw = resp.getWriter();
        pw.println("<HTML><BODY><H2> ERROR: Arguments were not specified.
</H2></BODY></HTML>");
        pw.close();
        return;
    }
}
}

```

Creating the static HTML page

A static HTML page may be easily generated in WebSphere Studio Application Developer. The page must contain a form with the action attribute referencing the the ASP.NET Web application and have the proper variables for propagating data from one application to another.

WebSphere Studio will help generate and verify the HTML produced. It will also verify links within the document. For example, if the `WS2NETAvgCalcRedirector` application did not exist within the current context, it would be flagged as a

possible problem. See Example 9-3 below for the static HTML used to provide the main interface into the simple application.

Example 9-3 AverageCalculator.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
    <META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <META name="GENERATOR" content="IBM WebSphere Studio">
    <META http-equiv="Content-Style-Type" content="text/css">
    <LINK href="theme/Master.css"
          rel="stylesheet" type="text/css">
    <TITLE>WAS Average Calculator</TITLE>
  </HEAD>
  <BODY>
    <P><FONT size="+3"><B>WAS Average Calculator</B></FONT></P>
    <P><FONT size="+1">Calculate Average of Two Numbers</FONT></P>
    <FORM name="AvgCalc" action="WS2NETAvgCalcRedirector" method="post">
      Argument 1: <INPUT type="text" name="WASArg1" size="10"><BR>
      Argument 2: <INPUT type="text" name="WASArg2" size="10"><BR>
      <BR>
      <INPUT type="submit" name="Submit" value="Calculate">
    </FORM>
    <P></P>
  </BODY>
</HTML>
```

The HTML in Example 9-3 produces the simple page in Figure 9-15 on page 409. The interface is very simple. The user enters two numbers and clicks the calculate button. The action on the form instructs the browser to direct the form input to the WS2NETAvgCalcRedirector servlet.

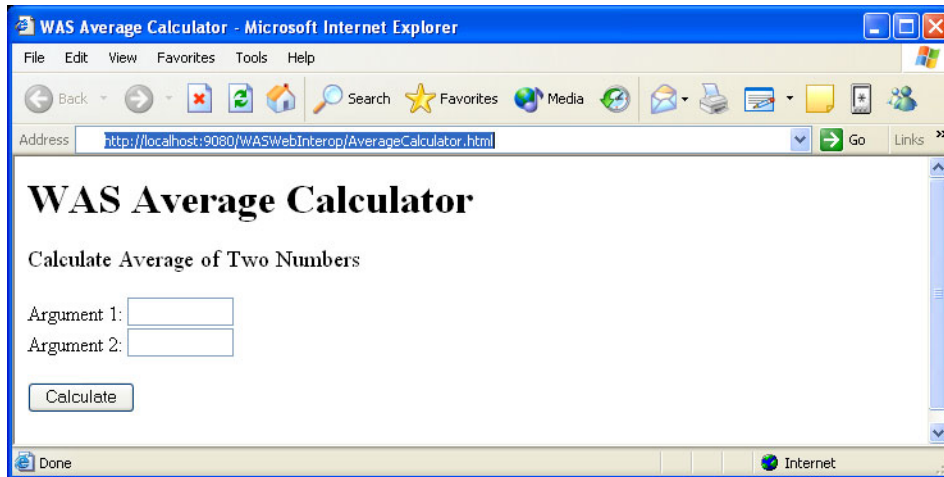


Figure 9-15 *AverageCalculator.html as Displayed by Browser*

Creating the ASP.NET application

Use Microsoft Visual Studio .NET to create an ASP.NET Web application. Name this new application `NETWebInterop`. The application can be created in one of several languages. We chose C# as our language of choice. The C# language is recommended for Java programmers since it has very similar constructs and syntax. By default, Visual Studio .NET automatically connects to IIS and publishes the new Web application on the local server. Publishing on a remote server is also possible by simply changing *localhost* to the server name. See Figure 9-16 on page 410 for the project screen used to create a new ASP.NET Web Application using the C# language.

Create a new ASP.NET Web form within your project. This can be accomplished by right-clicking the **NETWebInterop** project and selecting **Add -> New Item...** from the context menu. Choose **Web Form** from the available templates. Name this new Web form `AvgCalculatorResult.aspx`. Add two label Web controls to the page. The top label is the page title. The second label control shows the result of the average calculation. This control should have an ID of *avgLabel*. The control ID is important in mapping the Web element to a server side label object. Figure 9-17 on page 410 shows the ASP.NET form in the form designer.

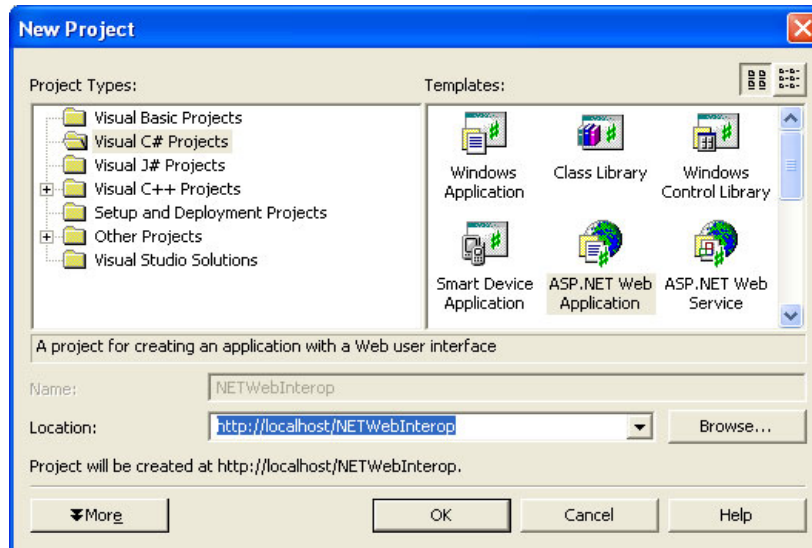


Figure 9-16 Creating a new ASP.NET Web application

In addition to creating the Web form, some code must be added to do the actual processing. In this simple example, the code was simply added directly into the Page_Load method of the server side class behind the Web form. See Example 9-4 on page 411 for the code used to process the external request. Note the parameters ADONETArg1 and ADONETArg2 and how they are used to propagate data from the WebSphere Web application to this application to be processed.

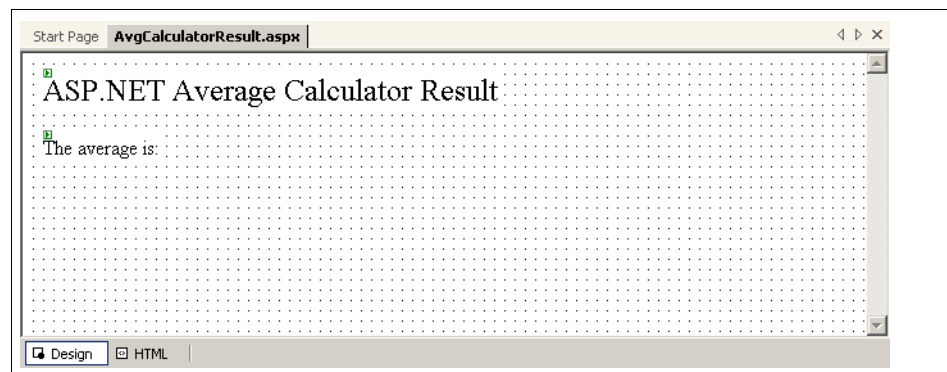


Figure 9-17 Using the ASP.NET form generator to create the average results form

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace redbook.coex.webinterop.ws2net.presentation
{
    public class WS2NETAvgCalculator : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.Label pageLabel;
        protected System.Web.UI.WebControls.Label avgLabel;
        // Page_Load is called each time the ASP.NET page is accessed
        private void Page_Load(object sender, System.EventArgs e)
        {
            try
            {
                // Verify the parameters
                double arg1 = Double.Parse(Request.Params["ASPNETArg1"]);
                double arg2 = Double.Parse(Request.Params["ASPNETArg2"]);
                // Compute the average
                double average = (arg1 + arg2) / 2;
                // Set avgLabel's Text property to the average
                avgLabel.Text = "The average is: " + average;
            }
            catch (FormatException)
            {
                // Arguments were not numeric
                avgLabel.Text = "ERROR: Arguments are not numeric";
            }
            catch (ArgumentNullException)
            {
                // Both arguments were not specified
                avgLabel.Text = "ERROR: Arguments were not specified";
            }
        }

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            InitializeComponent();
            base.OnInit(e);
        }
    }
}
```

```

    }
    private void InitializeComponent()
    {
        this.Load += new System.EventHandler(this.Page_Load);
    }
    #endregion
}
}

```

These are the major components for building a simple application using URL redirection to propagate data from WebSphere Web application to a ASP.NET application. The application is started by loading the static HTML page from the WebSphere side. Data is entered into the form and posted to the WebSphere servlet based Web application. This application repackages the data, generates a URL redirection request and hands it to the client. The client accepts this request and forwards it to the ASP.NET application. The ASP.NET application receives the numeric data, calculates the average, and returns a HTML page with the results to the client. In the next section, we implement a similar application using form-based propagation.

9.4.6 Form-based propagation implementation

In the URL redirection solution, we propagated data from a WebSphere Web application to a ASP.NET application. In this scenario, we build a similar application, going from an ASP.NET Web application to a Web application deployed on WebSphere. Another difference is that instead of using URL redirection to propagate data, this implementation will use standard HTML forms to post data from one Web application to another. The application is a simple calculator which takes two numbers, calculates the average, and returns the request to the client. This solution involves the following steps:

1. Create an ASP.NET application to generate a HTML form containing the action to send data to the WebSphere Web application.
2. Publish the ASP.NET application so it may be used by thin clients.
3. Create a WebSphere Web application to process data propagated from the ASP.NET application
4. Publish the WebSphere application.

Steps 1-2 and 3-4 may be done in reverse order. The important thing is planning the names of the applications in advance so they can be referenced by one another.

Creating the ASP.NET application

In the scenario we have chosen, we could have simply created a static HTML page and published it on the Microsoft IIS server. Instead, we chose to implement the HTML form using raw HTML output with ASP.NET. This was necessary because standard designer generated ASP.NET forms do not allow you to post form data anywhere except to the same application.

If you completed the URL redirection based scenario, you should already have an ASP.NET application named `NETWebInterop`. If not, Use Microsoft Visual Studio .NET to create an ASP.NET Web application. Name this new application `NETWebInterop`. The application can be created in one of several languages. We chose C# as our language of choice. The C# language is recommended for Java programmers since it has very similar constructs and syntax. By default, Visual Studio .NET automatically connects to IIS and publishes the new Web application on the local server. Publishing on a remote server is also possible by simply changing *localhost* to the server name. See Figure 9-16 on page 410 for the project screen used to create a new ASP.NET Web application using the C# language.

The next step is to create a new Web form. This Web form will be used to produce the initial HTML form for the simple average calculator. In the Visual Studio .NET Solution Explorer, right-click the **NETWebInterop** project and select **Add -> New Item...** from the context menu. Choose **Web Form** from the available templates. Name this new Web form `AvgCalculatorForm.aspx`. Nothing will be added to the form using the form design editor. Instead, bring up the HTML view for `AvgCalculatorForm.aspx` and remove the tags such that the file only contains the XML tag shown in Example 9-5. It should be the first line of the .aspx file.

Example 9-5 AvgCalculatorForm.aspx

```
<%@ Page language="c#"
    Codebehind="AvgCalculatorForm.aspx.cs"
    AutoEventWireup="false"
    Inherits="redbook.coex.webinterop.net2ws.presentation.AvgCalculatorForm" %>
```

The next step is to add the code to generate the HTML form. This code is added to the `Page_Load` method in the implementation class behind the aspx page. See Example 9-6 on page 414 for the code listing used to generate the HTML form.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace redbook.coex.webinterop.net2ws.presentation
{
    public class AvgCalculatorForm : System.Web.UI.Page
    {
        private void Page_Load(object sender, System.EventArgs e)
        {
            // Generate the raw HTML to display the form
            Response.Output.WriteLine("<HTML>");
            Response.Output.WriteLine("<HEAD><TITLE> ASP.NET Average Calculator
</TITLE></HEAD>");
            Response.Output.WriteLine("<BODY>");
            Response.Output.WriteLine("<H1> ASP.NET Average Calculator </H1>");
            Response.Output.WriteLine("<H3> Calculate Average of Two Numbers: </H3>");
            // The action on the form points to the WebSphere web application
            Response.Output.WriteLine("<FORM
action=\"http://localhost:9080/WASWebInterop/NET2WSAvgCalcResult\" method=post>");
            Response.Output.WriteLine("Argument 1:&nbsp");
            // Add the arguments using names the WebSphere application is expecting
            Response.Output.WriteLine("<INPUT type=text maxlength=10 size=10 name=\"WASArg1\">");
            Response.Output.WriteLine("<BR>");
            Response.Output.WriteLine("Argument 2:&nbsp");
            Response.Output.WriteLine("<INPUT type=text maxlength=10 size=10 name=\"WASArg2\">");
            Response.Output.WriteLine("<BR><BR>");
            // Add the calculate button
            Response.Output.WriteLine("<INPUT type=submit name=CalculateAvg value=Calculate>");
            Response.Output.WriteLine("</FORM>");
            Response.Output.WriteLine("</BODY>");
            Response.Output.WriteLine("</HTML>");
            // Close the output stream
            Response.Output.Close();
        }
        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            InitializeComponent();
            base.OnInit(e);
        }
    }
}
```



```

    }
    private void InitializeComponent()
    {
        this.Load += new System.EventHandler(this.Page_Load);
    }
}
#endregion
}
}

```

The listing in Example 9-6 on page 414 produces an HTML form very similar to the implementation of the previous example. Figure 9-18 shows the output produced by the code in AvgCalculatorForm.aspx.cs.

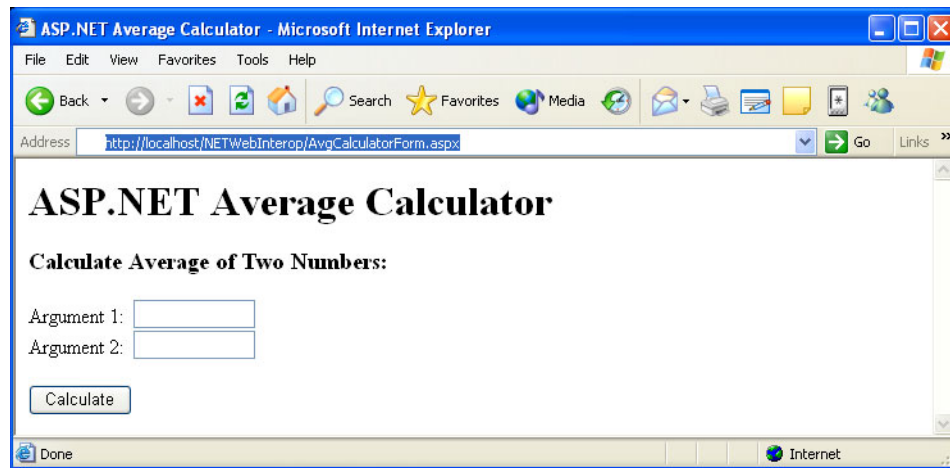


Figure 9-18 The ASP.NET average calculator form

The ASP.NET application above automatically gets published to the IIS server when it is created and compiled. No additional work needs to be done to start the application from a standard thin client.

Creating the WebSphere Web application

In this scenario, data is propagated from the ASP.NET application to the WebSphere application. After receiving the data, the WebSphere application must process the numeric data, calculate an average, and return results to the user.

If you have not done so in the previous example, use IBM WebSphere Studio Application Developer to create a new Enterprise Application Archive (EAR). Name the Enterprise Application Archive WASWebInteropEAR. Create a dynamic

HTML project named WASWebInterop. It should reside within the Enterprise Application you just created. Dynamic HTML projects can contain servlets, JSPs, and static HTML pages.

The next step is to add a servlet to the project. We named our servlet NET2WSAvgCalcResults. Naming is very important because, by default, the name of the servlet is used to reference the servlet from the ASP.NET form. This servlet will get the request and numeric data, process it, and return results to the user. See Example 9-7 for a code listing of the servlet.

Example 9-7 NET2WSAvgCalcResults.java

```
package redbook.coex.webinterop.net2ws.presentation;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class NET2WSAvgCalcForm extends HttpServlet implements Servlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        doGetPost(req, resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        doGetPost(req, resp);
    }

    public void doGetPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        // Write the document heading
        PrintWriter pw = resp.getWriter();
        pw.println("<HTML>");
        pw.println("<<HEAD><TITLE>WAS Average Calculator Results</TITLE</HEAD>");
        pw.println("<<BODY><H2> WAS Average Calculator Result</H2>");
        try
        {
            String arg1 = req.getParameter("WASArg1");
            String arg2 = req.getParameter("WASArg2");
            double dblArg1 = Double.parseDouble(arg1);
```

```

        double dblArg2 = Double.parseDouble(arg2);
        // Compute the average
        double average = (dblArg1 + dblArg2) / 2;
        // Write the average
        pw.println("The average is: " + average);
    }
    catch (NumberFormatException nfe)
    {
        pw.println("ERROR: Arguments are not numeric");
    }
    catch (NullPointerException npe)
    {
        pw.println("ERROR: Arguments were not specified");
    }
    // Write closing tags
    pw.println("</BODY></HTML>");
    pw.close();
}
}

```

The code for `NET2WSAvgCalcResults` in Example 9-7 on page 416 above produces a results page very similar to the previous example. See Figure 9-19 for the resulting HTML page.

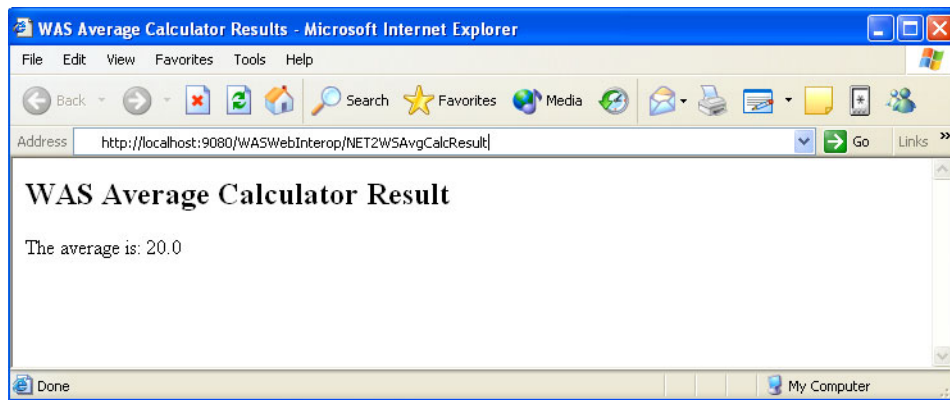


Figure 9-19 Results from ASP.NET to WebSphere using form-based propagation

These are the major components for building a simple application using form-based redirection to propagate data from an ASP.NET Web application to a WebSphere application. The application is started by loading the dynamic HTML page from the ASP.NET server. Data is entered into the form and posted to the WebSphere servlet-based Web application. The WebSphere application receives the numeric data, calculates the average, and returns an HTML page with the results to the client.

9.4.7 Recommendations

The implementations above give you an idea of how simple it is to create an interoperable Web environment using WebSphere and ASP.NET. The number of interaction types that could be produced using these simple methods is endless. Nearly any two Web environments can be integrated using the techniques discussed in this section. That said, this type of solution should not be used for every Web interoperability situation. The key points on page 401 should be kept in mind when deciding if you should build a solution of this nature. To recapitulate, those points are:

- ▶ Situations where Web applications must be integrated very quickly.
- ▶ Applications which reside on the Internet and must function over firewalls.
- ▶ Situations where a secondary Web application is used, but there is no direct agreement with the Web application provider.

Unless all these conditions are true, we recommend using another means such as Web Services, a shared relational database, or message queue to perform data propagation. There are also other things to consider, such as stateful versus stateless interaction, security, and performance when creating a Web interoperable solution. For example, you may not want to shuttle sensitive data from one environment to another over the Web. Performance is also a major issue. Form-based posting of data or URL redirection can be slow because of the extra steps required to encode, package, unpackage, and decode the data.

An additional recommendation and good design practice for using the solutions described is to use a layered approach. Keep business logic and data within classes or beans. For the sake of simplicity, the provided implementations did not do this. By separating logic and data from the presentation, when data propagation is required, it is simply a process of gathering the necessary data from the underlying logic and propagating it. Using this design practice will make it simpler to change the Presentation layer while leaving the business layer untouched.

Simply put, use the methods discussed in this section if and when it makes sense. They are tactical solutions to providing Web interoperability between WebSphere and .NET applications. This may work well if the applications are not mission- or business-critical. For these types of applications, a more strategic solution using technologies such as Web Services should be implemented.

9.5 Integrated security

The discussion of WebSphere and .NET coexistence security is a major topic; this chapter only focuses on Web interoperability between the two platforms,

therefore in this section we only discuss security in the context of Web applications.

In the following sections, we clarify some of the terms used in this chapter.

Authentication

Authentication is the process of establishing whether a client is valid in a particular context. A client can be either an end user, a machine, a service or an application.

Impersonation, delegation, re-authentication

When the current subject of the security context needs to act as another subject, this is called impersonation. In the process of impersonation, the current subject has to authenticate in the name of another subject. The new credentials are added to the current security context then the subject can switch personality before performing the next process.

During delegation, the current subject has to switch to another subject according to the delegation rules. For example, a server component has to call over to another component on another server and it has to use a certain subject for the call that the other server is expecting.

Re-authentication can be used to switch subjects for a client, while the original subject goes away. It can be used for users to get further permissions, for example, a normal user logs into the system, then re-authenticates as root (root cannot log in to the system in the first place).

Authorization

Authorization is the process of checking whether the authenticated user has access to the requested resource. There are two fundamental methods for authorization.

- ▶ **Access Control List**

Each resource is associated with an Access Control List, which is a list of subjects and attributes. The attributes define what the subjects can do with the resource.

- ▶ **Capability list**

Each user has a capability list associated with him/her, that is, a list of resources that the user can access and the corresponding privileges held by the user.

Role-based security

Roles introduce another level of abstraction to security. In authorization, the resource-subject association is made at a registry or directory level.

Secure communication

Secure communication involves authentication and authorization on the communication layer.

9.5.1 WebSphere security

Without going into details about WebSphere Application Server security, this section only focuses on Web application security. For more information about WebSphere security, refer to Chapter 1, “J2EE introduction” on page 3 and Chapter 11, “Quality of service considerations” on page 471, or check the redbook *IBM WebSphere V5.0 Security - WebSphere Handbook Series*, SG24-6573.

WebSphere has a virtualized security architecture. This security architecture can be tied to several different implementations, including the local operating system, external directory services, external security services.

Security for the Web is configured on the application server level, and also on the application level. The server settings configure security for the application server, including the User registry, SSL configuration, authentication mechanisms and so on.

On the application level, the following security settings are available for the Web module:

- ▶ One of the following authentication methods: basic authentication, form-based, certificate-based.
- ▶ Security constraints to protect resources. Multiple constraints can be defined for one module and one constraint can group multiple URLs and access methods (GET, POST, etc.).

The configuration settings for the application are stored in the web.xml file as it is defined in the J2EE specification.

9.5.2 .NET security

Without going into details about the .NET security, this section only focuses on Web application security. For more information about .NET security, refer to Chapter 2, “.NET introduction” on page 49 and Chapter 11, “Quality of service considerations” on page 471.

.NET security is very much tied to the operating system, Windows, and the directory service provided by the operating system is the Active Directory.

Security for the Web applications can be set in the web.config files under IIS. These files can exist under each directory in a hierarchy. The settings are inherited from parent directories, while others are defined or redefined for the directory. For further information about ASP.NET application security, refer to:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh19.asp>

9.5.3 Integrating authentication

There are different options and levels to make security interoperable between the two platforms. This chapter discusses the possibilities of integrating the authentication process. Authentication is the very first step performed when a *subject* is interacting with a secured application.

Note: *Subject* in the previous paragraphs stands for the user, service, application, or machine participating in the interaction with the secure application.

Different authentication mechanisms and implementations use different terminologies to identify the subject.

We discuss the following options for integrating authentication:

- ▶ Shared User registry
- ▶ Externalizing authentication

Shared User registry

Sharing the User registry between applications, or in our case, between the platforms is the most obvious first step. Unfortunately it is not a simple task at all; in most cases, the User registry exists in a system that requires customized code for the application or for the application server to access user data.

There are two scenarios in this case:

- ▶ .NET application uses the WebSphere application's User registry
- ▶ WebSphere application uses the .NET application's User registry

Accessing the .NET User registry from WebSphere

There could be different implementations for a .NET User registry, but the most common and most obvious is to use Active Directory.

WebSphere Application Server has a built-in LDAP User registry module that can be configured to use Active Directory. For detailed information about how to configure WebSphere, refer to the redbook *IBM WebSphere V5.0 Security - WebSphere Handbook Series*, SG24-6573.

A problem with this current solution could be that the connection between WebSphere Application Server and Active Directory cannot be secured using SSL.

WebSphere also supports custom User registries; this means that developers can use the User registry SPI to develop a specific module for WebSphere and perform authentication checks with any User registry.

Custom User registry provides the opportunity to integrate WebSphere with any other User registry, not only the one that is used for the .NET application. In other cases where the .NET application is using user registries other than Active Directory, WebSphere has the flexibility again to be able to connect to that User registry if the customized module is available.

Accessing non-Active Directory User registry from .NET

The primary User registry for .NET applications is Active Directory. Because of the fact that .NET is tightly integrated with the operating system Active Directory is almost the only solution for .NET applications.

Of course, customized solutions can be developed for the .NET applications, but .NET does not provide any API to attach the application to User registries other than Active Directory.

There are third party solutions for MS Internet Information Services to use LDAP for authentication.

Externalizing authentication

A quicker and easier solution for integrating authentication for both platforms on the Web is to use a third party authentication server. This solution not only provides authentication services for both platforms, but also provides a centralized (shared) User registry.

Tivoli Access Manager WebSEAL is a security reverse proxy server that provides authentication services for the Web. It works as a reverse proxy and captures the requests from the users accessing Web applications. The common language between the security reverse proxy and the Web application servers is HTTP. The security reverse proxy does not care what type of application is sitting behind it, as long as they understand HTTP and the authentication information encoded in the HTTP request.

The following simple diagram shows a scenario with WebSEAL and two application servers, WebSphere Application Server and MS Internet Information Services.

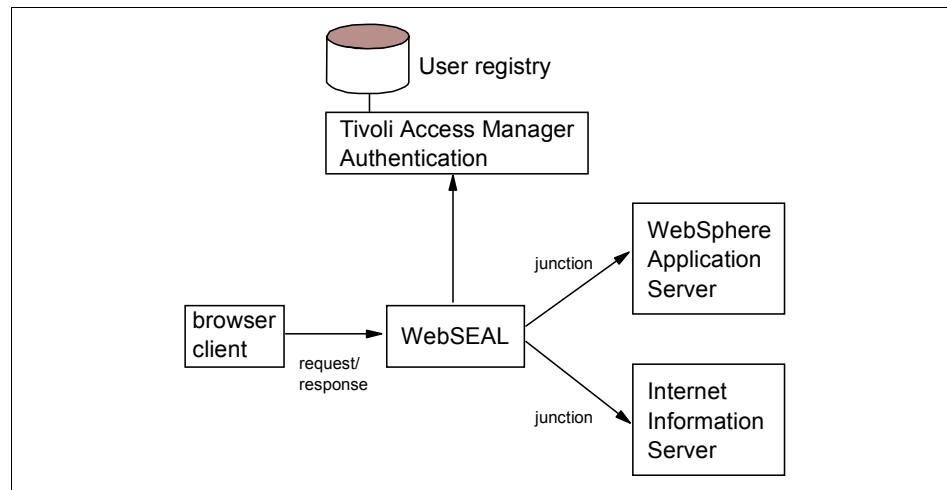


Figure 9-20 Externalized authentication

WebSphere Application Server and WebSEAL

There are several integration options between WebSphere and WebSEAL; these include:

- ▶ Basic authentication, where WebSEAL performs the authentication with the client using any of the following options: basic authentication, form-based authentication, certificates. Then WebSEAL passes the authentication information to the Web or application server in an HTTP request, using basic authentication.
- ▶ WebSEAL junction over to WebSphere using LTPA. WebSEAL performs authentication and sends the information to the application server in a form of a LTPA token.
- ▶ WebSEAL junction over to WebSphere using TAI. WebSEAL performs authentication and sends the information to the application server in special HTTP header elements. These can be picked up by WebSphere using a Trust Association Interceptor (TAI) module.

MS Internet Information Services and WebSEAL

There are multiple options to integrate MS IIS and WebSEAL.

- ▶ Basic authentication, as detailed in the previous section. IIS gets the information from WebSEAL in the form of basic authentication.

- ▶ WebSEAL can use SPNEGO to authenticate the client, then passes the SPNEGO token to IIS. For more information about SPNEGO, refer to:
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/http-sso-1.asp>
- ▶ In a manner similar to the WebSphere TAI junction, WebSEAL can send authentication information in the HTTP header that IIS can capture and understand. In this case, some development and customization is required on the IIS side to extract the information from the HTTP header.

Integrated solution

In the integrated solution, both application server platforms use and share the same WebSEAL security reverse proxy server. Since the authentication happens on the WebSEAL node, it also means that the User registry used for authentication is shared between the application servers. There are the following options for the User registry:

- ▶ Tivoli Access Manager uses Active Directory for its registry; follow the product documentation for more details.
- ▶ Tivoli Access Manager uses an LDAP directory and the .NET application has to have a customized solution to use the LDAP directory as a User registry.
- ▶ Tivoli Access Manager uses an LDAP directory and the .NET application uses Active Directory. The two directories have to synchronize with one another; this can be done with either customization or with a third party product.

In some cases, credential mapping might be necessary between the two directories to ensure that the right identity is presented from both sides.

Custom solution using SPNEGO

There is a custom solution to implement Single Sign-On for WebSphere in a Windows 2000 domain using the SPNEGO protocol. In this solution, the user is logged on to the Windows domain on the company's intranet. Whenever the user accesses a secured resource in WebSphere, the user is authenticated with his/her domain user name.

MS Windows 2000 and MS Windows 2003 uses Kerberos under the hood for authentication services. MS Internet Explorer is capable of performing negotiation with the Web server using SPNEGO.

The solution is a development of a custom Trust Association Interceptor (TAI) that intercepts the HTTP requests on behalf of WebSphere and follows the SPNEGO negotiation protocol to acquire user credentials from the client.

WebSphere Application Server V5.0.2 has all the necessary libraries to implement such a TAI. It requires the following libraries:

- ▶ Java GSS API, including the Kerberos security mechanism
- ▶ IBM SPNEGO implementation

For further information about SPNEGO, refer to:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/http-ss0-1.asp>

You can also use a network monitor tool to see how Internet Explorer and MS Internet Information Services perform a SPNEGO authentication process.

9.5.4 Integrating authorization

Integrating authorization for the two platforms can be quite challenging. Authorization settings for applications on both platform are entirely different; some of the differences are:

- ▶ Different application components require different authorization settings.
- ▶ Authorization attributes are assigned to different subjects. WebSphere is based on roles, ASP.NET is based on users and groups.
- ▶ Configuration for authorization persists in different places, different files in different formats.

Externalizing authorization

Externalized authorization makes system and security management much easier, since it is a demanding task in its own right. This section only targets authorization for Web applications, where the resources are identified as a URL or as part of a URL.

Authorization requires some sort of mapping between subjects and resources. Different Web and applications servers implement this mapping in different ways. As you may already know or have figured out from this section, the two platforms have different ways to configure authorization for applications.

Tivoli Access Manager WebSEAL can also handle authorization for Web sites. Authentication is taken care of by WebSEAL as described in “Externalizing authentication” on page 422. Once the user is authenticated, WebSEAL checks with the TAM server for authorization. In the TAM registry, ACLs (Access Control List) are defined for authorization purposes. ACLs define the subject(s) and the method(s) of access for the resource(s).

In this case, as with authentication, the User registry is shared by the two platforms; for further details, refer to “Integrated solution” on page 424.

The following diagram depicts the scenario where authorization for the Web is externalized for both WebSphere and MS IIS.

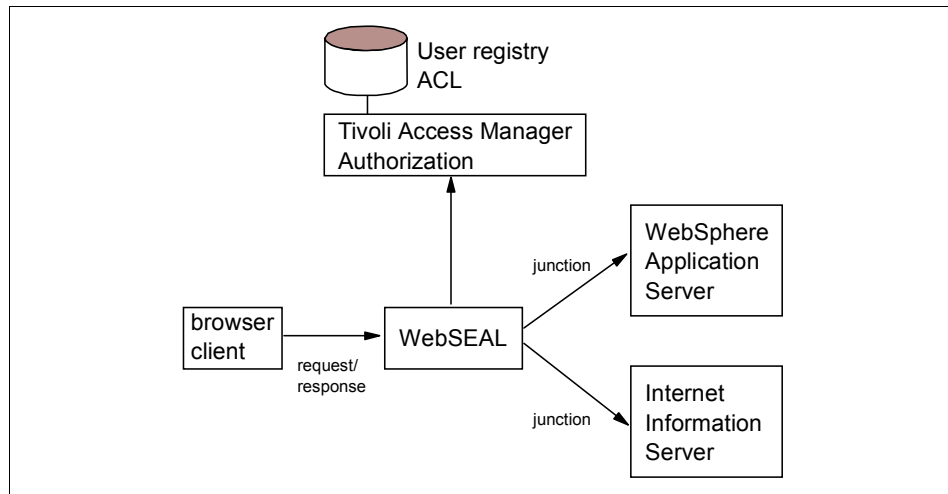


Figure 9-21 Externalized authorization



Part 3

Guidelines



Supporting technologies

In this chapter, we explain the different components of the runtime technologies covered in this book. Also, we describe the nature of Web Services and how they fit in this environment as an integration technology.

10.1 Web Services

Web Services are becoming the platform for application integration by defining common ways for applications to interact with one another across heterogeneous programming languages and operating systems. This is possible because the fundamental building blocks of Web Services are based on XML (eXtensible Markup Language) technologies. The use of XML allows one to describe and invoke Web Services in a way that is neutral in regards to platform and programming language.

Both WebSphere and .NET solutions provide similar levels of support for the open standards for Web Services technology. In Part 2, “Scenarios” on page 107, we show how to use Web Services to invoke services and to exchange data between WebSphere and .NET.

Web Services are software components capable of being accessed via standard network protocols such as SOAP over HTTP; these services can be new applications or just wrapped around existing legacy systems to make them network-enabled.

10.1.1 Technologies for Web Services

This section describes the technologies related to Web Services.

WSDL

The Web Service Description Language (WSDL) is used to describe the invocation interface of a Web Service and its location using a XML format. It can be composed of one or more files; those files describe what the Web Service does, its interface, where the Web Services reside and how to access them. They also include the method that is called, the parameters that are passed and the encoding that is used. If there is more than one file, an import element is required; it creates the reference to locate the other WSDL documents used.

A WSDL file is composed of six major elements, as follows:

- ▶ Type, which is a container that provides data type definitions using some type system, such as an XML schema.
- ▶ Message, which represents an abstract, typed definition of the data being communicated.
- ▶ PortType, which specifies the name and operation (method name) and points to the input and output message.

- ▶ Binding, a concrete protocol used to invoke the service and data format specification for a particular port type. It contains the protocol name, the invocation style, a service ID, and the encoding for each operation.
- ▶ Port, which specifies the location of the service for binding.
- ▶ Service, a collection of related ports.

WSDL 1.1 distinguishes two different message styles: document and RPC, and also two encoding styles: Literal or SOAP Encoded.

Table 10-1 Message styles

Encoding style	Document	RPC
Literal	Standard choice for MS tools	-
'SOAP' Encoded	-	Standard choice for Java tools

The style attribute within the SOAP protocol binding can contain one of two values: RPC or Document. RPC stands for Remote Procedure Call and it may fit well if your service implementation is indeed a procedure that can be invoked via the Web Service interface.

The Document style would fit in environments with more of a messaging style. You should use the Document style any time you are not interfacing to a pre-existing remote procedure call. The interpretation of the message, the mapping of the data types, etc. is done in the implementation so there is not much information in the interface when you use the Document style.

When the attribute is set to Document style, the client understands that it should make use of XML schemas rather than remote procedure calling conventions.

In the Document style, the message is placed directly into the body portion of the SOAP envelope, either as is or encoded. If the style is declared to be RPC, the message is enclosed within a wrapper element, with the name of the element taken from the operation name attribute and the namespace taken from the operation namespace attribute.

RPC implies that the Web Service <operation> is representing a procedure that can be invoked remotely. For the RPC style, the following rules are defined:

- ▶ The procedure name is the operation name and is sent as the root element within the SOAP body.
- ▶ The root element contains one child element for each parameter. The parameters can contain additional children if they are complex types.

- The response message is directly contained in the SOAP body.

In the Document style, the XML that is sent within the SOAP body is an instance of the defined WSDL message. Therefore, it allows one or more child elements (called parts) for the body.

This does not imply that there are parameters, procedure names, etc. The interpretation of the message content is completely left to the receiver. Generally, it is harder to manage by the service provider because the entire message must be parsed to identify which service implementation should be invoked. Also, there is a risk of ambiguous messages. For example: `deleteOrder(Order)` and `addOrder(Order)` may result in the same SOAP message at runtime.

The encodings define how data values defined in the application can be translated to and from a protocol format. These translation steps are referred to as *serialization* and *deserialization*, or, synonymously, *marshalling* and *unmarshalling*. SOAP encodings tell the SOAP runtime environment how to translate from data structures constructed in a specific programming language into SOAP XML, and vice versa.

We will focus on two types of encoding styles:

- Literal - the data is serialized according to the XML schema.

The Literal encoding style allows you to directly convert existing XML DOM tree elements into SOAP message content and vice versa. This encoding style is not defined by the SOAP standard, but by the Apache SOAP implementation.

Example 10-1 WSDL encoding example: Literal

```
...
<binding name="CSharpTempConverterSoap" type="s0:CSharpTempConverterSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document" />
  <operation name="celsiusToFahrenheit">
    <soap:operation
soapAction="http://wsbootcamp.com/webservices/celsiusToFahrenheit"
style="document" />
    <input>
      <soap:body use="literal" />
    </input>
  </operation>
</binding>
...
```

- SOAP Encoded - this defines a set of rules that specify how objects, arrays, structures and object graphs should be serialized.

The SOAP Encoded style allows you to serialize/deserialize values of data types from the SOAP data model. This encoding style is defined in the SOAP 1.1 standard.

The encoding used by the SOAP runtime can be specified at deployment or runtime.

Example 10-2 WSDL encoding example: SOAP Encoded

```
...
<binding name="TemperatureConverterBinding" type="tns:TemperatureConverter">
  <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="fahrenheitToCelsius">
      <soap:operation soapAction="" style="rpc"/>
      <input name="fahrenheitToCelsiusRequest">
        <soap:body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:TemperatureConverter" use="encoded"/>
      </input>
    </operation>
  </binding>
</wsdl:binding>
...
```

Generally speaking, we should use RPC/Literal or Document/Literal if we are talking about RPC or messaging.

WebSphere supports RPC/Literal, Document/SOAP Encoded and RPC/SOAP Encoded and WebSphere Studio Application V5.1 and WebSphere Application Server 5.0.2 already support Document/Literal.

Microsoft .NET, by default, creates Web Services using Document style encoding and does not provide support for creating a Web Service that accepts messages described by an RPC/Literal binding services.

We will cover some considerations in the Quality of Service chapter regarding performance of the encoding styles.

SOAP

Simple Object Access Protocol (SOAP), also known as service-oriented architecture protocol, is a lightweight protocol for the exchange of information in a decentralized, distributed environment using XML. SOAP is the protocol that is between the service provider, the service requestor and the service broker, enabling the publication and invocation of Web Services.

SOAP consists of three parts, as described below:

- ▶ *The envelope*, which provides control information, the address of a message and the message itself. The envelope can contain one or more headers and it must have one body. The header transports any control information such as quality of service attributes. The body contains the actual message with the parameters and result.
- ▶ *Encoding rules* define a serialization mechanism that can be used to exchange instances of application-defined data types.
- ▶ *RPC* defines a convention that can be used to represent remote procedure calls and responses.

Refer to *WebSphere Version 5 Web Services Handbook*, SG24-6891 and <http://www.w3.org/TR/SOAP/> for more details.

The W3C (World Wide Web Consortium) defines the SOAP specification and there are several SOAP implementations available at the moment. For example, the Apache SOAP V2.X implementation is an open-source Java-based implementation based on the IBM SOAP4J implementation and is a part of several IBM products including WebSphere Application Server V5 and WebSphere Application Developer V5. The Apache Axis implementation is a follow-on project of the Apache SOAP V2.X project and is often referred to as the Apache SOAP 3.0 implementation. Microsoft has its own SOAP implementation in the Microsoft SOAP Toolkit.

It is also important to mention that there is a Java API for XML-based RPC (Remote Procedure Calls), called JAX-RPC; this API enables developers to create interoperable and portable SOAP-based Web Services.

UDDI

UDDI stands for Universal Description, Discovery, and Integration; it is a registry mechanism based on standards such as XML and SOAP which can be used to publish and find Web Services descriptions. The Web Services can be discovered in two different ways: manually, meaning that a human can explore the UDDI registry, search for the service or the WSDL file and then use that information when programming the client; or programmatically, by the client application using the UDDI programming APIs, allowing dynamic binding and changing service providers runtime.

We are not going to use the UDDI registry in this book but it is important to give an overview of it since it is one of the technologies used for Web Services and for both WebSphere and .NET Web Services.

WebSphere uses the UDDI4J, which is a Java class library that provides an API which can be used to interact with a UDDI registry. This class library generates

and parses messages sent to and received from a UDDI server. The Microsoft .NET Framework uses the UDDI SDK, which is a collection of client development components, sample code, and reference documentation to enable developers to interact programmatically with UDDI-compliant servers. This is available for Visual Studio .NET and separately for Visual Studio 6.0 or any other COM-based development environment.

WebSphere Application Server V5 Network Deployment comes with a private UDDI registry.

The UDDI data model includes six types of information:

- ▶ Business entity, which describes a company or organization. Those entities provide general information about the Web Services such as business name, contacts, descriptions, identifiers, and categorization.
- ▶ Business service, which provides a business level description of a group of Web Services. A business service maps to a WSDL service.
- ▶ Binding template, which specifies the entry point at which you can access a Web Service. In many cases, a binding template points to an implementation address (for example, a URL) and maps to a WSDL port.
- ▶ tModel (technical model), which contains information about the technical specification that defines how to access a service. Its attributes are key, name, optional description, and URL.
- ▶ Taxonomy is a scheme for categorization. There is a set of standard taxonomies, such as the North American Industry Classification System (NAICS) or the Universal Standard Products and Services Classification (UNSPSC).
- ▶ Publisher assertions, also called business relationships: these make it possible to model complex businesses, such as subsidiaries, external business partners, or internal divisions. There are different kinds of relationships: parent-child, peer-peer, and identity.

The central source of information about UDDI is the Web site given below; it is operated by OASIS, which is a non-profit, global consortium that drives the development, convergence, and adoption of e-business standards.

<http://www.uddi.org>

DISCO

This is a Microsoft technology for publishing and discovering Web Services. We are not going to use DISCO in this book.

The users can browse to a specific discovery file (.disco) or to the root of the Web server to locate the files. The .disco files contain links to other discovery

documents, XSD schemas, and service descriptions. The discovery process is the preliminary step for accessing a Web Service. At design time, the Web Services clients can find the Web Service, see its description and how to interact with it.

WSIF

The Web Services Invocation Framework (WSIF) is a simple Java API for invoking Web Services, no matter how or where the Services are provided. It gives flexibility to the developers because it adds extensibility to WSDL, allowing the description of service implementations other than SOAP. It separates the API from the actual protocol, so they do not work directly with the Simple Object Access Protocol (SOAP) APIs. For example, it is possible to switch protocols or locations without having to recompile the client code. The API provides binding-independent access to any Web Service. So, whether it is SOAP, an EJB, JMS, or any other software framework, for example .NET, you have an API centered around WSDL which can be used to access the functionality.

Using WSIF, Java clients have one simple programming model to invoke any service described in the WSDL file; see Figure 10-1.

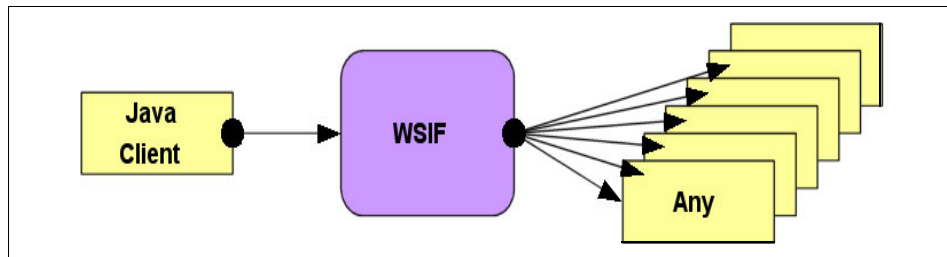


Figure 10-1 WSIF concept

The WSIF architecture has two distinct subsystems: build-time and runtime. This is illustrated in Figure 10-2 on page 437.

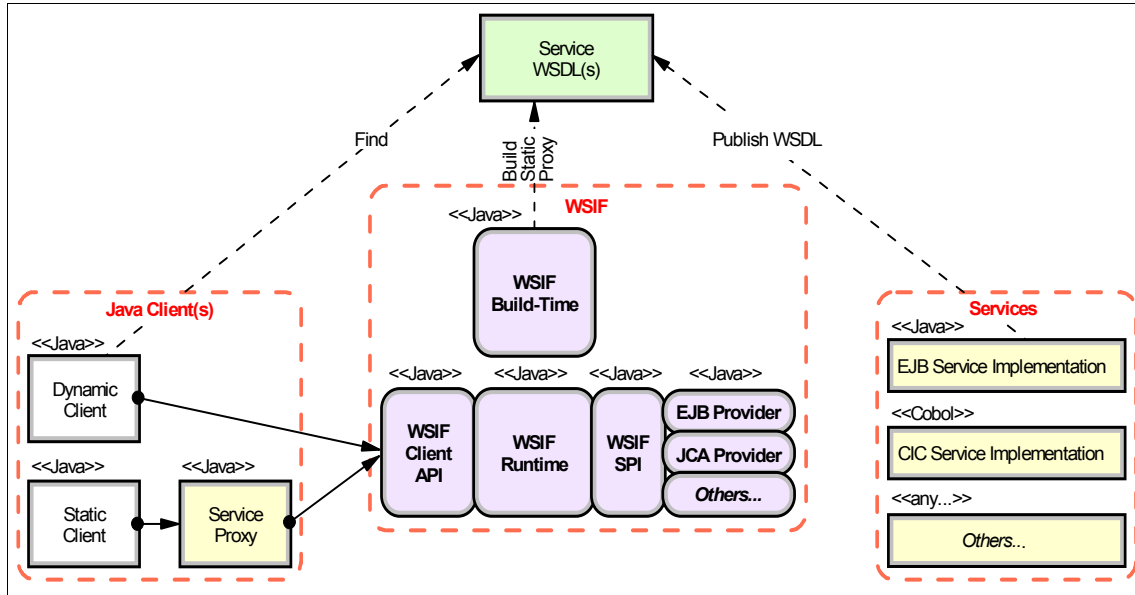


Figure 10-2 WSIF build-time, runtime and WSIF providers

The build-time subsystem provides tooling for creating Java service proxies to WSDL-described target services. These target services can be implemented in any supported WSIF technology, meaning any that supports a WSIF provider.

Note: A *provider* is a piece of code that supports a WSDL extension and allows invocation of the service through that particular implementation. WSIF providers use the J2SE JAR service provider specification, making them discoverable at runtime.

The runtime subsystem provides a client API and a service provider interface. A service client can either use the client API directly, or via a generated service proxy; these are implemented as Java classes.

Web Services Gateway (WSGW)

The Web Services Gateway is part of WebSphere Application Server V5 Network Deployment. We will not go into details, since the Web Services Gateway is not going to be used in the scenarios.

The Web Services Gateway component provides a framework for invoking Web Services between Internet and intranet environments. Its main benefit is that it allows interoperability between Web Services deployed on different vendor

platforms. The gateway will abstract the vendor details and publish and serve Web Service requests based on the standard protocols and transports.

One of the features of the gateway is the protocol transformation; the requester application might use one particular communication protocol to invoke Web Services, while the provider application uses some other protocol. Using the Web Services Gateway, it is possible to trap the request from the client and transform it to another messaging protocol.

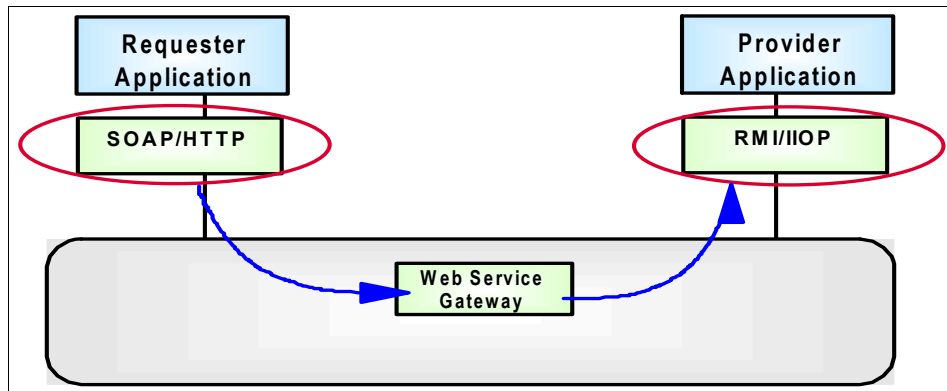


Figure 10-3 Web Services Gateway

Business Process Execution Language (BPEL)

Business Process Execution Language for Web Services (BPEL4WS) is another technology for Web Services which we will not explore in this book. It allows specifying business processes and how they relate to Web Services. This includes specifying how a business process makes use of Web Services to achieve its goal, as well as specifying Web Services provided by a business process.

A business process specifies the potential execution order of operations from a collection of Web Services, the data shared between these Web Services, which partners are involved and how they are involved in the business process, joint exception handling for collections of Web Services, and other issues involving how multiple services and organizations participate. This allows specifying long-running transactions between Web Services, increasing consistency and reliability for Web Services applications.

Web Services security

Web Services security is a message-level standard, based on securing Simple Object Access Protocol (SOAP) messages through an XML digital signature, confidentiality through XML encryption and credential propagation through

security tokens. It defines the core facilities for protecting the integrity and confidentiality of a message and provides mechanisms for associating security-related claims with the message.

The Web Service security specification is available at <http://www.ibm.com/developerworks/library/ws-secure/>; it proposes a standard set of Simple Object Access Protocol (SOAP) extensions that you can use to build secure Web Services. These standards confirm integrity and confidentiality, which are generally provided with digital signature and encryption technologies. In addition, Web Services security provides a general purpose mechanism for associating security tokens with messages. A typical example of the security token is a user name and password token, in which a user name and password are included as text. Web Services security defines how to encode binary security tokens such as X.509 certificates and Kerberos tickets.

Web Services security for WebSphere Application Server, Version 5.0.2 and above is based on standards included in the Web Services security (WS-Security) specification.

JAX-RPC

JAX-RPC stands for Java API for XML-based RPC and is part of the J2EE1.4 specification but can also be developed and deployed in J2EE 1.3 as a technology preview. It is a Java API for building Web Services and clients that use remote procedure calls (RPC) and XML. The remote procedure calls are represented by an XML-based protocol, for example SOAP.

This API defines how to map Java code to WSDL definitions and vice versa. On the server side, developers specify the remote procedures by defining methods in a Java interface. On the client side, developers provide the service endpoint by specifying a URL and invoking the methods on a local object that represents the remote object.

JAX-RPC uses technologies such as HTTP, SOAP, and WSDL; it is platform-independent, allowing interoperability across heterogeneous platforms and environments, including Microsoft .NET.

10.2 Client applications

The client applications are the first layer of the logical application layers diagram shown in Figure 3-2 on page 95. They are typically defined by GUI (Graphical User Interfaces) programs that execute on a user's machine or applications that execute in a Web browser. This means that the clients usually run on a client machine separated from the application server.

An application combines Web clients (such as Web browsers), Web application servers, and standard Internet protocols to access data and applications across one or more enterprises. Many technologies can be used during the development of the client side of a Web application (including Java, TCP/IP, HTTP, HTTPS, HTML, DHTML, XML, MIME, SMTP, IIOP, and X.509, among others). Often, project success and future adaptability depend upon the technologies used. Therefore, we encourage you to make the client side of any Web application rather lightweight, or to be more specific, “thin.”

10.2.1 Web browser

A Web browser is a client program which initiates requests to a Web server and displays the information that the server returns.

The browser is used to locate and display Web pages. The two most popular browsers are Netscape Navigator and Microsoft Internet Explorer, but there are others such as Mozilla and Opera. All of these are graphical browsers, which means that they can display graphics or text.

The Web pages can be written using markup languages such as HTML (Hypertext Markup Language), DHTML (Dynamic Hypertext Markup Language), or scripting languages which typically provide enhanced access to objects within the browser interface and HTML document, as well as many generic language features. An example of scripting languages would be JavaScript or VBScript.

A browser also supports a Java Virtual Machine (JVM) for execution of Java applets and supports browser plugins which are proprietary software programs that extend the capabilities of browsers in a certain way, giving them the ability to play audio or view movies, for example.

JavaScript is a compact, object-based scripting language for developing client and server Internet developed by Netscape to design interactive sites. It shares many features and structures of Java but was developed independently. JavaScript can interact with HTML source code, which makes it possible to add dynamic content to the Web sites.

VBScript is based on the Visual Basic programming language but it is simpler. In many ways, it is similar to JavaScript. It enables you to include interactive controls, such as buttons and scrollbars, on Web pages.

Other types of applications should also be mentioned in this category, such as the Java Applets and ActiveX programs.

Java Applet is an application program, written in Java, which can be retrieved from a Web server and executed by a Web browser. It can also run in a variety of other applications and devices. An applet must be loaded, initialized and run by a

Web browser. It can be used to process presentation logic, because it provides a powerful user interface for J2EE applications.

An ActiveX control is similar to a Java applet. It can be described as a set of rules for how applications should share information. It can be downloaded and executed by a Web browser. ActiveX controls can be developed in a variety of languages, including C, C++, C#, Visual Basic and Java. The ActiveX controls have full access to the Windows operating system, but the applets do not. Another difference between Java applets and ActiveX controls is that Java applets can be written to run on all platforms, but ActiveX controls are limited to Windows environments.

Be aware that not every browser supports all languages, so there is a limitation on running client applications. Even if the browser supports the language you are using, the users may have the support disabled.

10.2.2 J2EE clients

This section describes the different types of J2EE clients.

Thin Java client (not J2EE)

The thin Java client is also known as a *Web client*. It refers to the Java standalone program that is not running within the J2EE client container.

A thin Java client consists of two parts: *dynamic Web pages* containing various types of markup languages (HTML, XML, etc.), which are generated by Web components running in the Web layer, and a *Web browser*, which renders the pages received from the server. Thin clients usually do not do things like query databases, execute complex business rules, or connect to legacy applications.

If you need to execute one of these operations, it is better to use enterprise beans executing on the J2EE server where they can leverage the security, speed, services, and reliability of J2EE server-side technologies, instead of using thin Java clients.

There are several thin client technologies, such as HTML, JavaScript, DHTML, and Java applets. The design of the client side of a Web application depends upon which technology is employed, but you cannot choose your client technology without considering the server side of your application, too. If your server depends on JavaScript and complex DHTML to get its message across, your client should be able to handle these requirements. In other words, the client and server choices must be made as part of an end-to-end design.

Java applets

An applet is a component that typically executes in a Web browser. It is a Java class that can also run in a variety of other applications or devices.

An applet must be loaded, initialized and run by a Web browser; it can be used to process presentation logic, and it provides a powerful user interface for J2EE applications. However, simple HTML pages may also be used.

Applets embedded in an HTML page are deployed and managed on a J2EE server although they run in a client machine. They are considered to belong to the HTML page and an HTML page is managed by a J2EE server.

Refer to the Web site below for a better understanding of the technologies which can be used for building both client and server side Web applications:

<http://www-106.ibm.com/developerworks/java/library/j-framework2/understanding.html>

J2EE application clients

A J2EE application client is a standalone program launched from the command line or desktop and runs on a client machine. For example, it can access EJBs running on the J2EE application server. Although a J2EE application client is a Java application, it differs from a standalone Java application client in that it is a J2EE component that runs in the J2EE client container. This means that it has access to all the facilities of J2EE. The container provides the runtime support for the standalone application. Furthermore, it runs in its own JVM (Java Virtual Machine).

Since the J2EE application client is part of a J2EE application, it is portable, which means that it will run on an J2EE-compliant server. Also, it may access J2EE services as security authentication and JNDI lookups.

J2EE applications are packaged in JAR files with a deployment descriptor. The deployment descriptor for an application client describes the enterprise beans and external resources referenced by the application.

A J2EE application client provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from Swing or Abstract Window Toolkit (AWT) APIs, but a command line interface is certainly possible.

10.2.3 Windows .NET clients

This chapter describes the different types of .NET clients.

Console clients

.NET offers a console-based client which can run from any Windows command line and take advantage of the full range of Windows APIs and .NET libraries. Console clients can be either thick or thin depending on the use of technologies such as Web References (to call Web Services), .NET Remoting (to call .NET code on a remote server), classic COM, etc.

Graphic User Interface clients

.NET offers a new GUI technology library in Windows Forms (WinForms). WinForms are very similar to the look, feel, and coding style of previous versions of Visual Basic. GUI clients are able to take advantage of the same technology choices as console applications.

10.3 Server pages

Server pages are components of a Web application that run on the server side. They are part of the Presentation layer. In this section, we will explain both Java and .NET supporting technologies for server pages.

10.3.1 Servlets and JSPs

Servlets are server-side software components written in Java, and because of that, they inherit all the benefits of the Java language, including a strong typed system, object-orientation, modularity, portability and platform independence. They run inside a Java enabled server or application server, such as WebSphere Application Server. Servlets are loaded and executed within the Java Virtual Machine (JVM) of the Web server or application server, in the same way that applets are loaded and executed within the JVM of the Web client. Since servlets run inside the servers, they do not use graphical user interface (GUI).

The Java Servlet API is a set of Java classes which define a standard interface between a Web client and a Web servlet. Client requests are made to the Web server, which then invokes the servlet to service the request through this interface. A client of a servlet-based application does not usually communicate directly with a servlet, but requests the servlet's services through a Web server or application server which invokes the servlet through the Java Servlet API.

In the MVC (Model-View-Controller model), in servlet-only applications, the servlet is used as both the controller and the view. Servlets process the requests

(control) and produce the HTML response (view). In some cases, such as using JDBC to access back-end data, the servlet can also act as the model. HTML, DHTML, JavaScript and XML are useful for the static components of the programming model, servlets (controller) and JSP (view) are the most useful components in generating dynamic content.

JavaServer Pages (JSP) technology allows you to easily create Web content that has both static and dynamic components.

A JSP is mostly a XML or HTML document with special embedded tags. It runs in a Web container, and at runtime, the JSP is parsed and compiled into a Java servlet and may call JavaBeans or Enterprise JavaBeans to perform processing on the server.

The JSP tags enable the developer to insert the properties of a JavaBean object and script elements into a JSP file, providing the ability to display dynamic content within Web pages.

In order for the JSP container to understand and execute a custom JSP tag, the code implementing the tag and information on how to translate the tag and find and invoke the code must be available to the container. Tags for a particular function are normally aggregated into a tag library. Therefore, a tag library is a collection of custom tags. Tag libraries, or taglibs, are normally packaged as JAR files.

JSP technology supports the use of JSTL (Java Server Pages Standard Tag Library) which defines a standard set of tags and is used to optimize implementation.

Figure 10-4 on page 445 shows how the JSP fits in the MVC architecture.

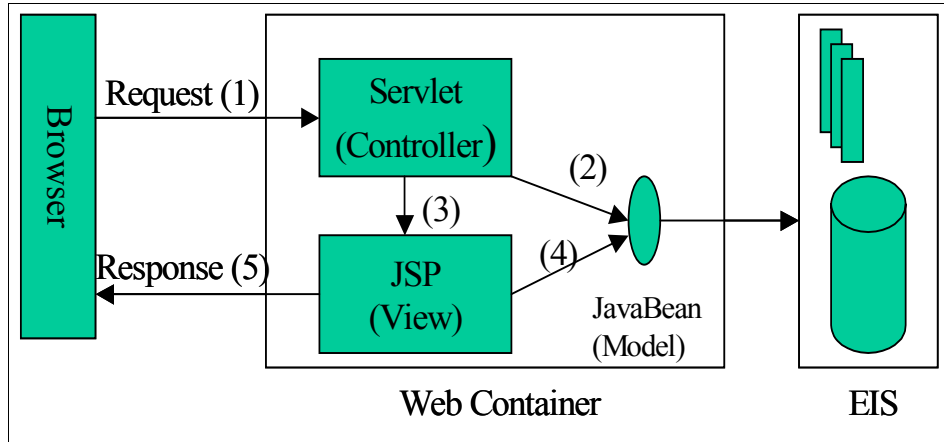


Figure 10-4 Typical JSP access model

1. The HTTP request comes in from the Web client and passes through the Web server to the application server. The servlet's `service()` method is invoked.
2. The servlet interacts with Java classes (in this case, a JavaBean) to process the request. The dynamic content is placed into a Java class (a JavaBean) that is to be shared with a JSP (in our example, we assume the request scope).
3. The servlet forwards the request to the JSP:
If this is the first request for the JSP, it is compiled into a servlet, and that servlet's `service()` method is invoked.
4. The JSP gets the shared object from the appropriate scope (we assume the request scope in this example).
5. Finally, the JSP generates the dynamic content and returns the generated Web page to the Web client.

Some advantages of using JSPs are:

- ▶ Separation of dynamic and static content - this allows for the separation of application logic and Web page design, reducing the complexity of Web site development and making the site easier to maintain.
- ▶ Portability - JSP technology is based on Java; this means that it is platform-independent. JSPs can be developed on any platform and viewed by any browser because the output of a compiled JSP page is HTML. No rewriting is necessary.

- ▶ Reusability - JSP technology emphasizes the use of reusable components such as JavaBeans, Enterprise JavaBeans and tag libraries, taking advantage of these technologies and improving the functionality.
- ▶ Scripting and tags - JSPs support both embedded JavaScript and tags. JavaScript is typically used to add page-level functionality to the JSP. Tags provide an easy way to embed and modify JavaBean properties and to specify other directives and actions.
- ▶ High quality of tool support - JSPs can be built using JSP-enabled development tools like WebSphere Studio Application Developer.
- ▶ XML - since the JSP 1.2 specification allows a JSP page to be an XML document, every tool that is capable of manipulating XML can now be used with JSPs. It is also easy to validate a JSP against a DTD or an XSD.

In this redbook, we are using the J2EE 1.3 specification; this means that we use JSP 1.2. The major change for the JSP 1.2 specification is the ability to encode the JSP in pure XML. There are also additional classes for the validation of tag libraries and new tags that facilitate iteration and handling life cycle events.

More details are available at:

<http://java.sun.com/products/jsp/>

10.3.2 ASP.NET

ASP.NET is a feature of Microsoft's Internet Information Services (IIS) and is the next generation of the Microsoft's Active Server Pages. Beyond a means of dynamically building Web pages, ASP.NET is a programming framework for building highly scalable Web applications.

A newer ASP

ASP.NET does not feel like the older versions of ASP. To list all of the new features is a little beyond our scope, but we will mention a few. Refer to www.asp.net for more information.

- ▶ CLR: Gone are the days of only being able to use Jscript and VBScript to give your pages access to back-end data. With the addition of Microsoft's Common Language Runtime (CLR) as a platform, ASP.NET now gives developers the ability to write their code as C#, VB.NET, J#, Jscript.NET, and in more than 30 other languages.
- ▶ Performance: ASP historically did not perform well. ASP.NET code is executed by the CLR and with the help of a Just-In-Time compiler, native code optimizations, and early bindings, performance is highly improved. Some applications have recorded 30 - 40% increases in performance when moving from ASP to ASP.NET.

- ▶ Web Services: ASP.NET, along with the CLR, constitutes the Web Services environment for IIS. Web Services operate as an object in ASP.NET, with similar treatment to the code-behind functionality mentioned below.
- ▶ Code-behind: ASP.NET offers several models for separating code from content; one mechanism is the code-behind feature. Using code-behind allows UI forms to make use of compiled code objects on the server. The code-behind feature involves creating a user interface in a .aspx file (Web Form) made up of HTML (plus ASP declarations) and inheriting from a code file living on the server. The separation is very clean between presentation logic and code to drive the presentation.

Model-View-Controller design pattern using ASP.NET

To demonstrate the design of a simple ASP.NET application and to illustrate the Model-View-Controller roles in the ASP.NET environment, let's talk about a common example that involves creating an ASP.NET application to access a database.

In our example, we use three files: SimpleForm.aspx, SimpleForm.aspx.cs, and SimpleDataGateway.cs

SimpleForm.aspx contains the description of our user interface and is defined in HTML with the following ASP.NET declaration:

Example 10-3 ASP.NET declaration

```
<%@ Page language="c#" Codebehind="SimpleForm.aspx.cs" AutoEventWireup="false"
Inherits="CodeBehindExample.SimpleForm" %>
```

The `Inherits` attribute tells the ASP.NET system which class the .aspx file inherits from at runtime. The `Codebehind` attribute is used by Visual Studio .NET but is not used at runtime. Note that using the same file name for the .aspx file and the .cs file is a convention used the Visual Studio .NET Application wizard. To make the UI SimpleForm.aspx use a different code-behind class, just change the `Inherits` attribute (and the `Codebehind` attribute if you are using Visual Studio). For a description of ASP.NET page directives, see:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconpage.asp>

SimpleForm.aspx.cs contains the event handling code for UI defined by SimpleForm.aspx and ties it to SimpleDataGateway. Because SimpleForm is named as the code-behind class for SimpleForm.aspx, the object must inherit from the .NET class `System.Web.UI.Page` or a subclass of `Page`.

Example 10-4 Inheritance for SimpleForm

```
/// <summary>
/// The description for SimpleForm
/// </summary>
public class SimpleForm : System.Web.UI.Page {
```

SimpleDataGateway contains all of the code needed to use the database.

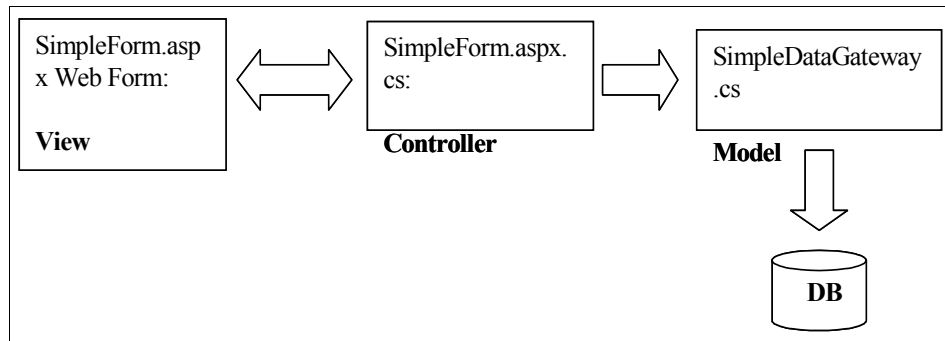


Figure 10-5 Model-View-Controller roles in ASP.NET

When a browser requests SimpleForm.aspx, the ASP.NET runtime looks at the @Page information and loads the precompiled assembly with the class specified in the Inherits attribute. The UI is then displayed and the events denoted by the code-behind object that the page inherits from are handled as they occur.

A common design flaw in ASP.NET applications is embedding the Model code in the Code-behind object filling the role of MVC role of Controller. Combining the Controller and Model makes it difficult to reuse the code filling the role of the Model. In our example, combining the Controller and Model could force us to refactor if we create another page that wants access to the database.

Example 10-5 Model implementation

```
private void InitializeComponent() {
    this.TestButton.Click += new System.EventHandler(this.TestButton_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}

private void TestButton_Click(object sender, System.EventArgs e) {
    // Don't put the database driving code in your codebehind object!!!
    // This violates the MVC design pattern.
    SqlConnection AConnection =
        new SqlConnection("server=(local);database=adb;Trusted_Connection=yes");
}
```

ASP.NET and Web Services

ASP.NET offers a framework to make, build and deploy Web Services simply. The mechanism used for Web Services is very similar to that of the code-behind feature from a developer's point of view. Internally with ASP.NET, much more goes on under the covers.

The two primary elements are an .asmx file and a .cs file. For demonstration purposes, we will call these MyService.asmx and MyService.asmx.cs (using the Visual Studio .NET convention). MyService.asmx is very much like the file SimpleForm.aspx without the HTML:

Example 10-6 MyService.asmx

```
<%@ WebService Language="c#" Codebehind="MyService.asmx.cs"
Class="SimpleWebService.CodeBehindTheService,SimpleWebService" %>
```

The important element here is the `Class` attribute which denotes the class used to implement the Web Service.

If we were to put the class in the .asmx file itself, it would be compiled by ASP.NET (via the specified language compiler) the first time the Web Service is requested. If the Web Service implementation lives in a different file, the assembly containing the source file should be compiled before the Web Service is deployed and placed in the bin directory of the Web Service location on the server (typically `wwwroot\<Web_Service_Name>\bin`).

For a complete list of `Class` attribute descriptions, see:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/gnconwebservicesdirectivesyntax.asp>

In this example, we are using a separate file and to get the best performance, we are specifying the name of the assembly along with the class name:

`Class="SimpleWebService.CodeBehindTheService,SimpleWebService"`.

Providing the name of the assembly `",SimpleWebService"` is optional, but it helps performance because ASP.NET does not have to look in all of the assemblies in the bin directory, only the one noted.

The actual implementation of the Web Service resides in MyService.asmx.cs in the class `CodeBehindTheService`. The class can inherit from `System.Web.Services.WebService` to get access to ASP.NET specific functionality provided by system objects such as `Application`, `Session`, `User`, etc. The objects can be useful for life cycle information, security, and other highly desirable bits.

The public methods of the class are not exposed to Web Service clients as callable methods by default. To expose a method to a client from a service, the [Web Method] attribute is required.

Example 10-7 Web Service implementation

```
public class CodeBehindTheService : System.Web.Services.WebService {  
    [WebMethod]  
    public string HelloWorld() {  
        return "Hello World";  
    }  
    ...  
}
```

When a call to our service comes from a client, IIS and ASP.NET help the message get from HTTP port 80 to our service. On our behalf, the ASMX Handler portion of ASP.NET uses the declaration in MyServices.asmx to figure out which class implements our service. It then determines the correct method to call by inspecting the message. The message to call HelloWorld would look as shown in Example 10-8.

Example 10-8 SOAP request

```
POST /SimpleWebService/MyService.asmx HTTP/1.1  
Host: localhost  
Content-Type: text/xml; charset=utf-8  
Content-Length: length  
SOAPAction: "http://tempuri.org/HelloWorld"  
  
<?xml version="1.0" encoding="utf-8"?>  
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
  <soap:Body>  
    <HelloWorld xmlns="http://tempuri.org/" />  
  </soap:Body>  
</soap:Envelope>
```

The ASMX Handler uses the SOAPAction header by default to find the method being called. Notice that the SOAPAction header includes the Web Service's namespace and the method. There are optional attributes that can change the namespace and other calling aspects of the Web Service, but for this example we will just keep the default basics.

Armed with the class name (from the .asmx file) and the method (from the SOAPAction Header in the message calling the service) the .asmx file inspects the class using .NET reflection to see if a Web Method is exposed from the

service implementation class. If no matching method is found, an exception is thrown.

Once the method is found, the ASMX Handler uses the .NET XMLSerializer to deserialize the XML message into .NET objects. Because our method does not have arguments, we can skip this step.

The ASMX Handler calls our HelloWorld method and can take our response and once again use the XMLSerializer to package our response into an XML message contained in a SOAP Response. The response message will look as shown in Example 10-9.

Example 10-9 SOAP response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <HelloWorldResponse xmlns="http://tempuri.org/">
      <HelloWorldResult>Hello World</HelloWorldResult>
    </HelloWorldResponse>
  </soap:Body>
</soap:Envelope>
```

ASP.NET has grown head and shoulders above the features and functionality traditionally offered by ASP. Any developer interested in using Web Services and .NET should count ASP.NET as a central technology to the Microsoft Distributed Computing effort.

10.4 Distributed components

Distributed technologies rely on reusable components that are distributed across physical machines, providing different services across the network. This means that the parts of your program can be deployed to as many different physical machines and in as many separate operational systems processes as appropriate to achieve the performance, scalability, and availability goals of your system.

There are some related standard technologies, such as:

- ▶ Component Object Request Broker Architecture (CORBA) which is a vendor independent architecture and infrastructure that applications use to work together over networks.
- ▶ Sun Java Remote Method Invocation (RMI) protocol which is a mechanism for invoking methods remotely on other machines.
- ▶ Microsoft Distributed Component Object Model (DCOM) which is a protocol that enables software components to communicate directly over a network.

The distributed architecture is very efficient in the multi-tier design of applications because it simplifies developing, deploying, and maintaining enterprise applications. The developers can focus on the business logic and rely on various back-end services to provide the infrastructure, and client-side applications (both stand-alone and within Web browsers) to provide the user interaction.

10.4.1 EJBs

An Enterprise Java Bean is a server-side component that encapsulates the business logic of an application. EJBs simplify the development of large, distributed applications by providing automatic support for system-level services, such as transactions, security, and database connectivity, allowing the developers to concentrate on developing the business logic.

According to the EJB specification, “*Enterprise JavaBeans is an architecture for component-based distributed computing. Enterprise beans are components of distributed transaction-oriented enterprise applications.*” More details about the specification can be found at:

<http://java.sun.com/products/ejb/>

Basically, the EJB environment can be described as follows: the EJB components run inside the EJB container of an J2EE-compliant application server, the container has the connection to the database or to other components. An EJB client can access the EJBs from the same Java Virtual Machine (JVM) or from another JVM over remote interfaces.

There are six main components of the EJB technology:

- ▶ **EJB server** - provides the primary services to all EJBs. An EJB server may host one or more EJB containers. It provides services to the EJBs such as naming, security, persistence, messaging, etc.
- ▶ **EJB container** - provides the runtime environment for the enterprise bean instances; it is between the EJB component and the server.

- ▶ **EJB component** - represents the EJBs themselves. There are three types of enterprise beans: entity, session, and message-driven beans. We will go into details later in this section.
- ▶ **EJB interfaces and EJB bean** - the interfaces for client access (EJB home and EJB object) and the EJB bean class. The home interface is used by an EJB client to gain access to the bean. The object interface is used by an EJB client to gain access to the capabilities of the bean. This is where the business methods are defined.

The EJB bean class contains all of the actual bean business logic. It is the class that provides the business logic implementation and it contains the bean life-cycle methods of create, find, or remove.

- ▶ **EJB deployment descriptor** - defines the runtime quality of service settings for the bean when it is deployed. Settings such as transactional settings are defined in the deployment descriptor. It also describes logical relationships among entity beans.
- ▶ **EJB client** - a client that accesses EJBs. The client can invoke the EJB via a local or remote interface. A local client is a client that is in the same JVM with the session or entity bean. A remote client accesses a session bean or an entity bean through the bean's remote interface and remote home interface. The local interface is supposed to be faster than the remote since there is no network traffic involved.

More information can be found in the IBM Redbook *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819.

There are three types of EJBs:

- ▶ **Entity beans** - Entity beans are modeled to represent business or domain objects. They usually represent data (entities) stored in a database or in any persistent data. Entity beans may employ either CMP (container managed persistence) or BMP (bean managed persistence). BMP provides additional flexibility by allowing the developer to fully manage the persistence of the bean. Any additional complexity involves manually writing the necessary SQL code. The persistence code is generated automatically in the case of CMP. The advantage of using container managed persistence is that the entity bean can be logically independent of the data source in which the data is stored.
- ▶ **Session beans** - A session bean is the simplest form of EJB you can create. Session beans are not persisted to a datastore, but rather, are transient objects that may or may not hold state during a series of client invocations in the context of a single user session. A session bean may, in fact, choose to save or retrieve data directly from a database or some other persistence mechanism (although the state of the bean itself is not saved). There are two types of session beans: stateful and stateless. The first type is dedicated to a

single client and maintains conversational state across all the methods of the bean. The latter type can be shared across multiple clients, so any information kept in instance variables should not be visible to a client.

- **Message-driven beans** - These are similar to session beans; message-driven beans (MDB) may also be modeled to represent tasks. However, a message-driven bean is invoked by the container on the arrival of a JMS message (asynchronous messages). They typically represent integration points for other applications that need to work with the EJB. The advantage of using the message-driven bean model is to make developing an enterprise bean that is asynchronously invoked to handle the processing of incoming JMS messages as simple as developing the same functionality in any other JMS `MessageListener`.

10.4.2 .NET Remoting

In the .NET Framework, Microsoft chose not to use a standard distributed technology like CORBA to provide distributed component capabilities. Instead, the company developed a new, .NET specific technology called *Remoting*. Simply put, Remoting allows remote access to objects designed to run on the .NET Framework.

Remoting is a simple technology to implement and is very flexible. Remote objects may be hosted by Internet Information Services, a Windows .NET-based service, or by a standalone application. Flexibility comes from the layered component based approach of the Remoting architecture. Figure 10-6 on page 455 shows a high-level overview of .NET remoting. Within the .NET Remoting component of this diagram are several components. These components are:

- **Formatter** - The formatter takes a remote request and formats it into something that can be sent over the network. The .NET Framework contains a binary formatter and a SOAP formatter. Custom formatters may be added.
- **Channel** - A remoting channel creates the physical channel between the remoting client and server. The .NET Framework provides two channels, a TCP channel and an HTTP channel. The TCP channel provides a straight socket connection, while the HTTP channel uses the HTTP protocol to send and receive messages. These channels build the appropriate headers and package the data to be sent. The channel interface is extendable. Therefore, custom channels may be created to allow remoting over protocols such as IIOP or FTP.
- **Transport** - The transport performs the actual sending and receiving of data. Transports are tied directly to channels and cannot currently be interchanged.

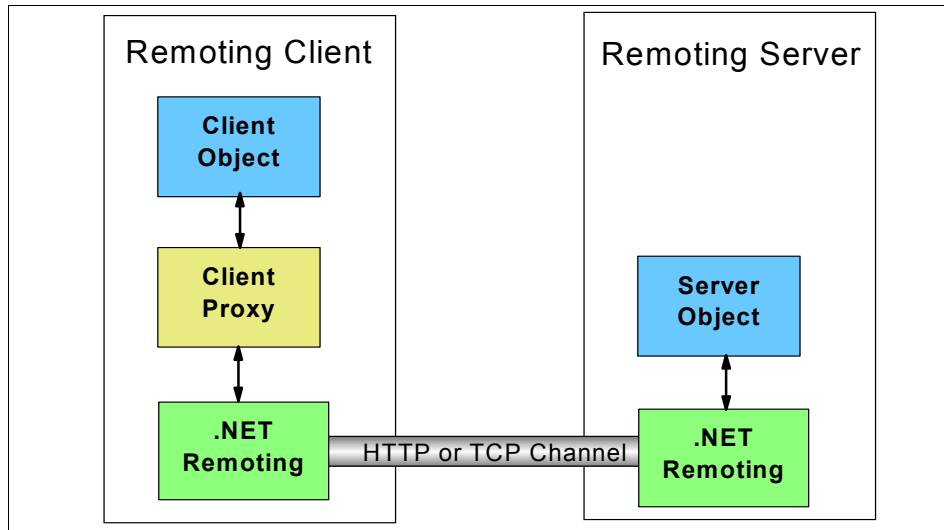


Figure 10-6 High-level view of .NET Remoting

Remoting activation

Activation is making an object available for use by instantiating the object. There are two distinct types of activation for .NET remote objects. The type of activation scheme you choose will depend on the type of application you are building. The two types of activation are:

- **Client activation**

In client activation, the remote object is instantiated immediately when the proxy is created on the client. Client activation is used in situations where the life of the object needs to be controlled by the client.

- **Server activation**

Server activation means that remote object instantiation is controlled by the server. There are two types of server activation, singleton and single call. When a remote object is created as a singleton, a single instance of the object is created for all clients of the object. If the server object is created as a single call object, each time the object is accessed remotely results in a new server object servicing the call. A singleton object is effective in maintaining state across multiple clients at the expense of requiring the object to be thread safe. In contrast, a single call object need not be thread safe but cannot maintain state.

Tip: Use caution when choosing the type of server activation to use. A singleton object may use significantly fewer resources but could be a bottleneck if it contains shared resources that must be synchronized.

Remote object life cycle

The life-cycle of an object includes creation, usage, and then finally destruction of the object. In a garbage collected environment such as Java or .NET, much of this is taken care of by the runtime environment. This is not the case in a distributed environment. Distributed environments add an additional, potentially problematic element to the object life-cycle. In a distributed environment, the real object lives on the server, while the client only has a proxy to that object. An additional mechanism must be added to ensure resources are managed correctly in a distributed environment. The .NET Framework uses a *Lifetime lease* to help manage resources.

Lifetime leases allow a measure of control over the remote object life-cycle, mainly remote garbage collection. Leases for .NET remoting objects work similarly to the DHCP protocol for IP address distribution and recovery. A client maintains a lease to a remote resource. The client must keep the lease current or the remote resource is freed. In .NET remoting, the client, a lease manager, and a sponsor all participate in handling leases. Similar to DHCP, a client may specifically request a lease to be renewed. This request is handled by the lease manager. Another technique is to use a sponsor or multiple sponsors. Sponsors may be registered for a remote object. When the lease manager determines it may be time to release a remote object, each sponsor is queried to see if the object is still required. If the sponsor or sponsors no longer need the object, it is garbage collected.

10.5 Database access

Examine almost any diagram of a software system and somewhere in the diagram you will find the ubiquitous line or arrow connected to a cylinder. The cylinder typically has a label containing the word *database*. *Database* is a generic term for a mechanism used for storage and retrieval of a collection of information. In software systems, many different types of databases are commonly used. When hearing the word database, many developers immediately think of large database management systems, capable of complex queries and storing millions of records. Although this is also correct, we tend to forget the generality of the term database. In fact, there are many different types of databases. In-memory databases are used for non-persistent, high-speed access. A flat text file such as a standard UNIX® password file containing a list of users, user information, and encrypted passwords is also a database. Web

search engines are a massive database of URLs and content, indexed by keywords.

Many applications require and access multiple databases. Frequently, these databases are located on remote systems and must be available over a network. Remote access invites issues such as security and concurrence. The list of issues continues to grow as system complexity increases. There are several points that may be taken from this discussion:

1. Databases are a common element of software systems.
2. Many different types of databases are commonly used in a software system.
3. Having different types of databases leads to different methods to access each type of database.
4. As the types of services required of a database increase in complexity, so must supporting technologies and interfaces to access the database.

These issues and others have prompted many changes in database access methods over the past decade. Many of the legacy interfaces are still in use today. However, as with most widely used technologies, standards have emerged. One of the most monumental standards to emerge was the Structured Query Language (SQL) for relational databases. The SQL standard helped to bring about the dominance of relational database systems.

During the surge of relational database systems, the shift from centralized to distributed client/server computing also began. In the client/server model, the database is a remote entity. Clients access the database remotely over the network. In order to connect to remote databases, database vendors began providing custom libraries and tools. This made interoperability between databases very difficult. Again, the wheels of standardization churned and Open Database Connectivity (ODBC) was created.

Open Database Connectivity was developed for the Microsoft Windows Operating System to provide a standard interface to access databases. The ODBC interface was designed such that software vendors could provide drivers to access a multitude of database types. ODBC drivers have been written to access everything from ASCII text files to all the major relational database systems. ODBC was one of the first major advances toward standardizing database access. To date, there are ODBC implementations for the Microsoft Windows, Unix, and Linux operating systems.

Following ODBC, Microsoft created a new interface for database access, Object Linking and Embedding Database (OLE/DB). The goal of OLE/DB was to provide a COM interface for data access. The move towards COM components was underway and the ODBC interface was designed mainly for applications

developed in the C programming language. OLE/DB proved to be a difficult and tedious technology for database access.

Simplification of the OLE/DB technologies was provided by a wrapper technology, the ActiveX Data Objects (ADO). ActiveX Data Objects also made it possible to use OLE/DB providers from COM based languages such as Visual Basic. ADO was the last major addition to the Windows COM-based component library. Microsoft .NET technology was soon to follow and with it came a new database access method, ADO.NET. Section 10.5.3, “ADO.NET” on page 460 contains an overview of the ADO.NET technology.

During the same period that Microsoft database access technologies were changing, the Java language became popular. One of the first extensions to Java was Java Database Connectivity (JDBC). JDBC functions in a way similar to ODBC except that it is object-oriented and built for Java. After a few releases, JDBC was bundled as a standard part of the Java runtime library. JDBC is currently the standard for Java database access.

We have provided a brief overview of the evolution of database access technologies. The purpose of the overview was to help understand the history of database access technologies in order to get insights into the technologies currently in use. These technologies are discussed in the sections that follow.

10.5.1 EJBs

Database access in an enterprise application normally requires additional features beyond using those services provided by JDBC. Additional services such as distributed transaction support, named lookup, multi-layer security, concurrency, and persistence are some of the most common required by enterprise applications. These additional features can be obtained by using Enterprise Java Beans (EJBs), more specifically, the enterprise entity bean.

Entity beans represent business or domain-specific concepts, and are typically the nouns of your system, representing fine-grained concepts such as customers and accounts. They usually represent data (entities) stored in a database. Since they represent data that is persistent in that database, changes to the bean result in changes to the database. See Figure 10-7 on page 459 for a high-level diagram of an entity bean.

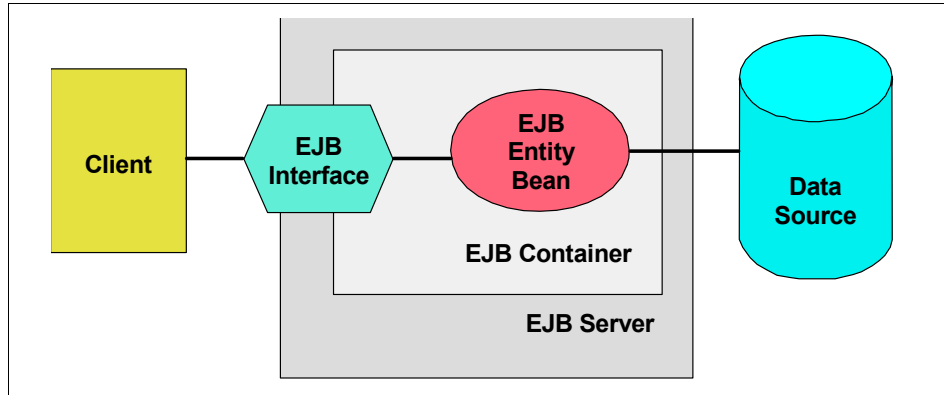


Figure 10-7 High-level overview of an entity bean

Entity beans allow us to objectify our data, and there are many good reasons for working with objects representing the data, versus implementing data access directly. For example, it is a lot easier to work with objects since they become reusable components in a system. Converting an employee record in a database into a reusable employee object is an example.

While the developer concentrates on the database to entity mapping, the EJB Entity container provides functionality such as concurrency, transactions, and security. In addition, by leveraging the persistence services of the container, the developer is freed from the burden of writing SQL.

This discussion has only given a very high-level overview of data access with Enterprise Java Beans. For additional information pertaining to entity beans, consult *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819.

10.5.2 JDBC

Java Database Connectivity (JDBC) is the standard for Java database access. JDBC provides standard interfaces for accessing data whether it be local or on a remote server. Java applications simply need to use the standard JDBC interfaces to perform database operations. By using JDBC, the underlying data source can be changed with minimal or no changes to the source code. Another major advantage of JDBC is that developers only need to learn one interface, instead of different interfaces for each database they wish to access.

You may be asking yourself how JDBC accomplishes such a feature. How can it possibly support all the many types of databases available? The answer is simple. Like ODBC, JDBC is mainly an interface. Database providers must

provide drivers, conforming to the JDBC specification, to provide access to their database. The JDBC component makes this interface nearly transparent by providing a driver manager. The driver manager manages drivers available to the Java Virtual Machine. Extensions to version 2 of JDBC also include a data source component. This component makes it simpler to use pre-configured JDBC connections and connection pools.

In addition to a standard interface, one goal of JDBC was to provide a simpler interface than its predecessor, ODBC. JDBC simplifies database access by providing a small number of classes and interfaces. The main JDBC interfaces are:

- ▶ **DriverManager** - The DriverManager class is a static class. Drivers register themselves with the driver manager and clients get connections from it.
- ▶ **Connection** - The Connection interface is returned by the driver manager when a connection is made.
- ▶ **Statement** - This interface represents a static SQL statement. A statement is executed by the database system and a ResultSet may be returned.
- ▶ **PreparedStatement** - This is similar to Statement with the addition of parameters.
- ▶ **CallableStatement** - This is similar to Statement with the addition of a standard call syntax and input and output parameters which may be required to call stored procedures.
- ▶ **ResultSetMetaData** - This provides MetaData access for a Statement (or one of its derivatives) ResultSet.
- ▶ **DatabaseMetaData** - This is used to gather information about a database. Catalog, schema, table, and data type information can be obtained using this interface.

In addition, several enterprise features were added in JDBC V2.0 and later which should be mentioned. Some of these features are:

- ▶ **Connection pooling** - Added as a JDBC 2.0 extension to allow the pooling of JDBC connections.
- ▶ **Statement pooling** - Added in JDBC 3.0 to allow connections to be pooled.
- ▶ **DataSource interface** - This interface was introduced in the V2.0 extensions package. Data sources allow connections to be managed externally through the Java Naming and Directory Interface (JNDI).

10.5.3 ADO.NET

ADO.NET is the standard database access mechanism for applications using the .NET Framework. Like database access technologies preceding ADO.NET, the

ADO.NET architecture is based on a set of interfaces. These interfaces are implemented by database vendors to provide native managed ADO.NET providers.

One striking difference between ADO.NET and other database access technologies is the lack of a driver manager component. Another difference is the thinness of the interfaces. Thinness in this case means the lack of standard requirements for implementors. On the positive side, this allows a near infinite amount of flexibility. A negative consequence is that code written directly to a particular provider database cannot be easily used with another provider.

ADO.NET provides eight interfaces for a standard data provider. These interfaces can be seen in Figure 10-8 on page 462. A list of high-level interfaces and a brief description of each follows:

- ▶ **IDbConnection** - The IDbConnection interface provides for basic connectivity services. In ADO.NET, connections are typically specified via a connection string. Most data providers include some form of connection pooling facility.
- ▶ **IDbCommand** - Database commands are created and executed through the use of the IDbCommand interface. Commands are specified as one of three types: Text, TableDirect, or StoredProcedure. Text commands are SQL statements run as-is. Table direct mode provides SQL access (select, insert, update, delete) to a single table. Stored procedure mode allows you to run stored procedures. The command interface also provides access to the parameter collection. The parameter collection stores statement and stored procedure parameters used during command execution.
- ▶ **IDataReader** - The data reader is provided as a forward-only interface for processing result set data. Data is processed row-by-row by calling a Read method on the interface. The data reader can also be used to return table metadata as a schema table. This schema table can then be written out as XML.
- ▶ **IDataAdapter** - The IDataAdapter interface operates directly on the .NET Framework provided DataSet class. The DataSet is a very robust class that allows in-memory storage, access, and updates of data. The main purpose of the data adapter is to allow the synchronization of the DataSet and data in the database. Microsoft provides a default implementation of a DataAdapter which database providers can inherit.
- ▶ **IDbTransaction** - Connection level transaction support is generally provided through the IDbTransaction interface. In addition, some providers, such as the SQL Server provider, provide distributed transaction capabilities. This support is normally provided by a custom attribute on the connection and not related to IDbTransaction.

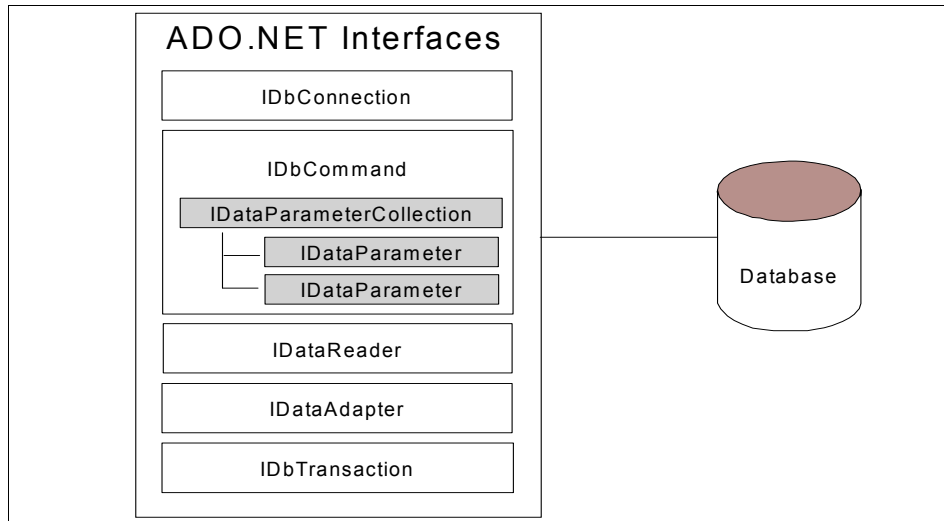


Figure 10-8 The ADO.NET standard interfaces

Command builders

The ADO.NET database access model does not use cursors. Cursors are the common way to do updates. Instead, ADO.NET relies on the data adapter and SQL INSERT, UPDATE, and DELETE statements to keep DataSets and remote tables synchronized. This means that someone has to specify these statements to get updates to occur. Fortunately, most database providers implement a command builder to help generate these statements. For example, the SQL Server provider has a `SqlCommandBuilder` class to help build these commands. Further, a command builder can be associated with a data adapter so the correct commands get generated at runtime, as needed.

Legacy support with ADO.NET

Since the .NET Framework is relatively new, a native driver may not be available nor ever be written to access some database systems. There are two solutions to this problem. The first solution is the OLE/DB bridge. The OLE/DB bridge allows .NET applications to use legacy OLE/DB providers as an ADO.NET provider. The OLE/DB provider performs a bridge between OLE/DB COM components and the ADO.NET runtime environment. Applications using the OLE/DB bridge will function, but performance will suffer due to the constant switch between COM and .NET.

The second option for not having a native ADO.NET provider is the ODBC Bridge. The ODBC bridge works very similarly to the OLE/DB bridge. The ODBC bridge could potentially perform better than the OLE/DB bridge due to the elimination of the COM Interoperability layer. Instead of using COM, the ODBC

bridge uses the more direct Platform Invoke (P/Invoke) interface to call C-style APIs on the ODBC Manager (odbc32.dll).

10.6 Messaging middleware

The programming components that comprise an IT system must exchange information when they collaborate in a complex business application. When they run on different operating systems and must communicate across the network, this presents a considerable integration challenge. A message-based approach provides the solution. Programmed components can exchange information in the form of messages, so the developers can focus on business logic instead of the complexities of different operating systems and network protocols. In short, a messaging middleware mechanism enables business applications, information and processes to exchange information with each other, regardless of where they are running.

JMS

The J2EE specification provides a Java API called Java Messaging Service that allows applications to create, send, receive, and read messages. It enables Java programs to communicate with other messaging implementations. Messaging enables distributed communication that is loosely coupled, asynchronous and reliable.

The JMS architecture is composed of a JMS client which can be a Java applet, application, servlet, JavaBean or EJB that produces and consumes messages; a JMS provider which has the responsibility of implementing the JMS API on top of an underlying messaging system; messages which are the objects that communicate information between JMS clients, and administered objects which are JMS objects created by the administrator; these objects enable the client to interact with a JMS provider and also to identify a destination for a message.

The JMS API has the following features:

- ▶ Client applications, EJBs, JavaBeans and Web components can send or synchronously receive a JMS message. Client applications can also receive asynchronous JMS messages.
- ▶ Message-driven beans which enable the asynchronous consumption of messages. The JMS provider can also implement concurrent processing of messages by message-driven beans.
- ▶ Message sends and receives can participate in distributed transactions.

More details about the J2EE specification can be found at:

<http://java.sun.com/j2ee/>

WebSphere MQ

IBM implements JMS as its messaging middleware software, which is called WebSphere MQ. It provides application programming services that enable application programs to communicate with each other using messages and queues. This kind of communication is referred to as *asynchronous messaging*. It provides assured, once-only delivery of messages. WebSphere MQ means that you can separate application programs, so that the program sending a message can continue processing without having to wait for a reply from the receiver. If the receiver, or the communication channel to it, is temporarily unavailable, the message can be forwarded at a later time. WebSphere MQ also provides mechanisms for generating acknowledgements of messages received and triggering that allows MQ to “wake up” an application when the message arrives.

The programs that comprise an WebSphere MQ application can be running on different computers, on different operating systems, and at different locations. The applications are written using a common programming interface known as the Message Queue Interface (MQI), so that applications developed on one platform can be transferred to another.

The applications communicate using messages and queues. One application puts a message on a queue and the other application gets that message from the queue.

WebSphere MQ messaging products make it straightforward for applications to exchange information among more than 35 IBM and non-IBM platforms, including Linux and Windows 2000, even if the target program is not running. They take care of network interfaces, assure delivery of messages, deal with communications protocols, and handle recovery after system problems. I Secure Sockets Layer (SSL) can be used for secure communications. These products also perform message transformation and routing, and systems management.

Another product of this family is the WebSphere Business Integration Message Broker; it distributes, transforms and enriches real time information to simplify end-to-end communications using different message structures and formats. It can take the message and manipulate it and then distribute it to many different applications running on different platforms. The two main functions of the product are message transformation and routing. WebSphere Business Integration Message Broker gets the message from the queue, understands it (the format of the message), transforms (manipulates) the message to the format that the applications can read, and finally, distributes the message to the other applications.

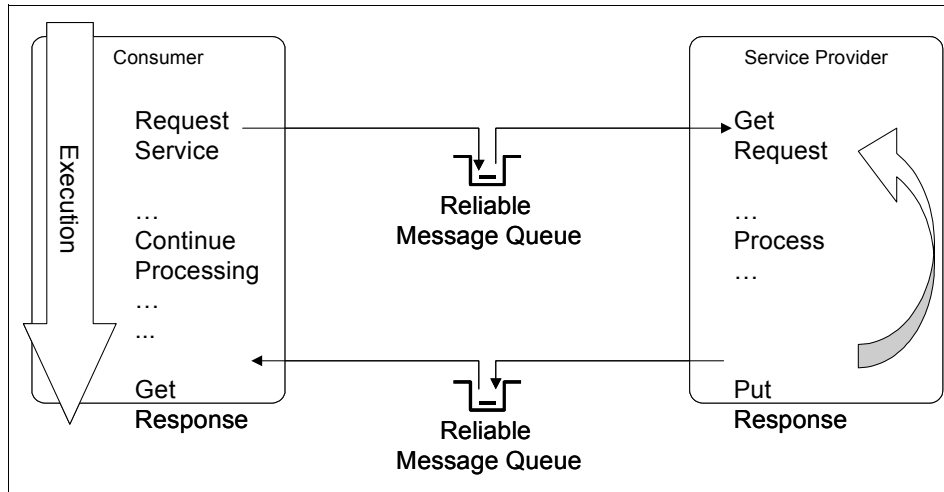


Figure 10-9 Messaging middleware architecture

MQ Classes for .NET

The MQ classes for Microsoft .NET allows WebSphere MQ to be invoked from Microsoft .NET applications.

The classes were available as a freeware SupportPac, but from WebSphere MQ V5.3 CSD05, the WebSphere MQ Classes for Microsoft .NET are fully incorporated in the WebSphere MQ product.

For more details, please refer to the redbook *WebSphere MQ Solutions in a Microsoft .NET Environment*, SG24-7012.

WebSphere MQ Transport for SOAP

At the time of the writing of this book, WebSphere MQ Transport for SOAP is provided as a category 2 (freeware) SupportPac. This SupportPac provides the ability to flow a SOAP message over a WebSphere MQ transport. The package reuses existing SOAP infrastructure and will operate with Axis SOAP (for Java) and the SOAP stack, embedded in the Microsoft.NET Framework.

It permits interoperation between Web Services clients and servers written to run in either of these environments.

It also enables the benefits of a WebSphere MQ infrastructure to be harnessed in popular Web Services environments. Benefits include:

- ▶ Reliability, queueing and simple clustering
- ▶ A controlled environment
- ▶ Access to services where it is not convenient to deploy HTTP servers.

The popular SOAP infrastructures provide convenient, high-level, well-tooled programming interfaces that provide a standard, interoperable message format (SOAP). This can make for easier programming of WebSphere MQ applications.

For more details, please refer to the redbook: *WebSphere MQ Solutions in a Microsoft .NET Environment*, SG24-7012.

10.7 Back-end integration

In this section, we are going to describe both WebSphere and Microsoft mechanisms to connect existing or new applications with legacy applications such as Enterprise Information Systems (EIS). The EIS can be: enterprise resource planning (ERP), mainframe transaction processing, and non-relational databases, among others.

10.7.1 J2C

J2EE Connector Architecture is a standard that provides integration between J2EE applications and existing Enterprise Information Systems (EIS). Each EIS requires just one implementation of the J2EE Connector architecture because an implementation adheres to the J2EE Connector Specification, so it is portable across all compliant J2EE servers. It is designed to facilitate sharing of data and to integrate new J2EE applications with legacy or other heterogeneous systems.

The J2EE application and the EIS communicate via a resource adapter. A resource adapter is a J2EE component that implements the J2EE Connector architecture for a specific EIS. It is stored in a Resource Adapter Archive (RAR) file and may be deployed on any J2EE server, much like the EAR file of a J2EE application. A RAR file may be contained in an EAR file or it may exist as a separate file.

A resource adapter is analogous to a JDBC driver. Both provide a standard API through which an application can access a resource that is outside the J2EE server. For a resource adapter, the outside resource is an EIS; for a JDBC driver, it is a DBMS. Resource adapters and JDBC drivers are rarely created by application developers. In most cases, both types of software are built by vendors who sell products such as tools, servers, or integration software.

10.7.2 .NET

.NET is a product strategy where J2EE is a specification standard. There is no .NET specification that competes with the Java Connector Architecture. Instead,

there are two products in the .NET family which make connecting to other applications viable: BizTalk Server and the Host Integration Server.

Host Integration Server is the name of the Microsoft's new version of SNA product and is billed as having 'unbeaten' support for clients, network protocols, and mainframe systems. With Host Integration Server, a single installation can support up to 30 000 host sessions with features such as single sign-on, transaction support, object-oriented programming model, and more.

BizTalk is Microsoft's Enterprise Application Integration product and is specifically tailored to help organizations automate and orchestrate applications and interactions with business partners. Beyond a server and a set of tools, BizTalk is also about accelerators and adaptors. BizTalk adaptors enable BizTalk to integrate with 'no code' to WebSphere MQ, SAP, Microsoft's SQL Server, with Web Services. BizTalk Accelerators are add-ons to BizTalk that target specific industries including Health Care, Financial Services, and many others.

While .NET is still in its infancy, the Back-end Integration story for .NET includes products and technologies that are both compelling and diverse. While JCA offers an open specification that many in the industry are using to build adaptors, Microsoft .NET products offer a competing view from existing products.

10.8 Other integration technologies

You can find information about other integration technologies if you read Chapter 6, "Scenario: Synchronous stateful" on page 261. There are two scenarios covered in this book for stateful synchronous communication between WebSphere J2EE and Microsoft .NET.

- ▶ Wrapping Java components in .NET.
- ▶ Wrapping .NET components in Java.

Note: The .NET platform is not entirely "correct" in these scenarios, because the technologies are developed for the pre-.NET world: COM objects and ActiveX controls.

In both cases, we used a technology to bridge the two platforms on the process level. The solution assumes that both platforms, WebSphere and .NET, exist on the same machine.

In the first case, the Windows component starts up a Java Virtual Machine (JVM) inside the process and accesses Java objects, using the ActiveX bridge.

In the second case, WebSphere has direct access from Java to Windows components using the IBM Interface Tool for Java.

For more information about these technologies, refer to Chapter 6, “Scenario: Synchronous stateful” on page 261.

10.8.1 ActiveX Bridge

WebSphere Application Server provides an ActiveX to EJB bridge that enables ActiveX programs to access WebSphere Enterprise JavaBeans through a set of ActiveX automation objects.

The bridge accomplishes this by loading the Java Virtual Machine (JVM) into any ActiveX automation container such as Visual Basic, VBScript, and Active Server Pages (ASP). There are two main environments in which the ActiveX to EJB bridge runs:

- ▶ **Client applications**, such as Visual Basic and VBScript, are programs that a user starts from the command line, desktop icon, or Start menu shortcut.
- ▶ **Client services**, such as Active Server Pages, are programs which are started by some automated means like the Services control panel applet.

The ActiveX to EJB bridge uses the Java Native Interface (JNI) architecture to programmatically access the JVM code. Therefore, the JVM code exists in the same process space as the ActiveX application (Visual Basic, VBScript, or ASP) and remains attached to the process until that process terminates.

For more information, refer to the WebSphere Application Server InfoCenter at the IBM Software Web site; under the application server, navigate to **Applications -> Client modules -> Using application clients -> Application clients -> ActiveX application clients**, or search for *WebSphere ActiveX application clients* in the InfoCenter.

10.8.2 IBM Interface Tool for Java

This is a technology, formerly known as Bridge2Java, available on the IBM Alphaworks Web site at <http://www.alphaworks.ibm.com/tech/bridge2java>.

Interface Tool for Java is a tool which allows Java programs to communicate with ActiveX objects. It allows easy integration of ActiveX objects into a Java environment. Using the Java Native Interface and COM technology, Interface Tool for Java allows an ActiveX object to be treated just like a Java object.

Using Interface Tool for Java requires simply running a proxy generating tool that creates Java proxies from the ActiveX controls's typelib. These proxies can then be used to allow a Java program to communicate with the ActiveX object.

For more information about this technology, refer to the Alphaworks Web site:

<http://www.alphaworks.ibm.com/>



Quality of service considerations

Quality of service (QoS) refers to the capability of a platform to provide better service to a selected application.

It is a challenge to build systems that deliver extremely high levels of service because many variables should be considered. The success of an application depends on scalability, performance, availability, security, manageability, among other things, and all these are critical to avoid failure.

In terms of Web Services, QoS covers a whole range of techniques that match the needs of service requestors with those of the service provider's based on the network resources available.

In this chapter, we will describe some considerations about the quality of service for both WebSphere and .NET applications. It is not our intention to compare the platforms, but to show what each one can provide in terms of QoS.

11.1 Scalability

Scalability refers to the capability of a system to adapt readily to a greater or lesser intensity of use, volume, or demand while still meeting business objectives. It is related to other QoS since applying appropriate scaling techniques can greatly improve availability and performance. For example, scalability should be considered carefully, otherwise some performance bottlenecks may develop. Scaling techniques are especially useful in multi-tier architectures when you evaluate components associated with the edge servers, the Web presentation servers, the Web application servers, and the data and transaction servers.

On demand computing requires the ability to scale up or scale down an application, depending on the current requirements. Thus, scalability is important to improve efficiency and reduce cost.

This section will introduce some scaling techniques used by WebSphere and .NET to help you understand best practices in optimizing the application's environment.

11.1.1 WebSphere

IBM WebSphere Application Server V5.0 implements Web containers and EJB containers on its application servers. The application servers each run in their own JVM (Java Virtual Machine). There are some strategies for scalability as follows:

- ▶ Physically separating some components such as the HTTP server, Web container, EJB container, and database, to prevent them from competing for resources (CPU, memory, I/O, network, and so on) or to restrict the access to a resource from another machine (for example, inserting a firewall in order to protect confidential information).
- ▶ Distributing the load among the most appropriate resources, and using workload management techniques such as vertical and horizontal scaling.

WebSphere Application Server cluster support

Clusters are sets of servers that are managed together and participate in workload management. The servers that are members of a cluster can be located on the same machine (vertical scaling) and/or across multiple machines (horizontal scaling). See details in 11.3.1, “WebSphere” on page 483.

The servers that belong to a cluster are members of that cluster set and must all have identical application components deployed on them. Other than the applications configured to run on them, cluster members do not have to share

any other configuration data. One cluster member might be running on a huge multi-processor enterprise server system while another member of that same cluster might be running on a small laptop. The server configuration settings for each of these two cluster members are very different, except in the area of application components assigned to them. In that area of configuration, they are identical.

Starting or stopping the cluster will automatically start or stop all the cluster members, and changes to the application will be propagated to all application servers in the cluster.

A cluster contains only application servers, and the weighted workload capacity associated with those servers.

WebSphere workload management (WLM)

WebSphere WLM enables the applications running under WebSphere to be scaled to any number of machines at any time, increasing the amount of requests the applications can serve.

The incoming processing requests from clients are transparently distributed among the clustered application servers. WLM enables both load balancing and failover when servers are not available, improving the reliability and scalability of WebSphere applications.

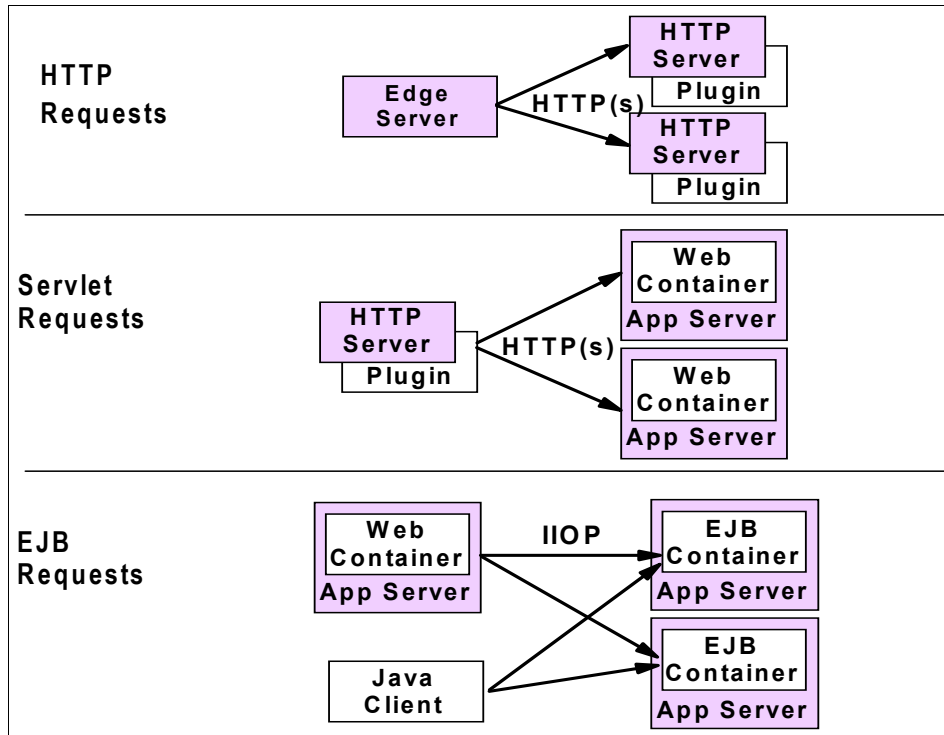


Figure 11-1 Types of requests that can be workload managed in WebSphere V5.0

- HTTP requests can be shared across multiple HTTP servers.
This requires a TCP/IP sprayer to take the incoming requests and distribute them. So, the HTTP requests are workload managed externally to WebSphere Application servers. There are both hardware and software products available to spray TCP/IP requests. The Network Dispatcher is a software solution that is part of the WebSphere Edge Server; it applies intelligent load balancing to HTTP requests. The WebSphere Edge Components are part of IBM WebSphere Application Server Network Deployment V5.0, and also provide caching proxy functions.
- Servlet requests can be shared across multiple Web containers.
The WebSphere Plugin to the HTTP server distributes servlet requests to the Web container in clustered application servers.

Clustering application servers that host Web containers automatically enables plugin workload management for the application servers and the servlets they host. The Web containers can be configured on the same machine or multiple machines.

- ▶ EJB requests can be shared across multiple EJB containers.

In IBM WebSphere Application Server Network Deployment V5.0, workload management for EJBs is enabled automatically when a cluster is created within a cell. There is no need for a special configuration to enable it. The workload management uses a plugin in the Object Request Broker (ORB) to distribute EJB requests among the application servers (cluster members).

EJB requests can come from servlets, Java client applications, or other EJBs. In the figure above, the Web container is in a separate machine from the EJB container.

11.1.2 .NET

Microsoft clustering technologies are key to improving scalability and availability in Windows environments. Both Windows 2000 and Windows 2003 use a Sterling strategy that includes:

- ▶ Server Cluster
- ▶ Network Load Balancing
- ▶ Component Load Balancing

Note: Component Load Balancing is *not* available on Windows Server 2003.

Windows Server 2003 Clustering

Clustering allows the users/administrators to access and manage the nodes as a single system. It provides availability to the the servers since it guarantees that the applications will be available even if one server crashes. It provides failover support for applications and services and also maintenance of data integrity.

Server cluster is included with Windows Server 2003, Enterprise Edition, and Windows Server 2003, Datacenter Edition. Clustering is installed automatically with Windows Server 2003 and some configuration is necessary in the Cluster Administrator.

Windows Server 2003 supports up to eight cluster nodes for each server cluster. (a node is a member of a server cluster) and the nodes may be configured in one of three ways:

- ▶ Single node server cluster

This is a cluster configuration which has only one node and can be configured with or without external cluster storage devices.

- ▶ Single quorum device server cluster

This is a cluster configuration which has two or more nodes and is configured in such a way that every node is attached to one or more cluster storage devices.

- ▶ Majority node set server cluster

This is a cluster configuration which has two or more nodes and is configured in such a way that the nodes may or may not be attached to one or more cluster storage devices.

Usually, server clustering is used for databases, e-mail services, line of business applications, and custom applications. Windows Server 2003 provides vertical scalability for clusters based on Server Cluster.

If Server Cluster detects a failure of the primary node for a clustered application, the clustered application is started on a back-up cluster node and all requests to the application are redirected to the back-up node. Each node is attached to one or more cluster storage devices which allow different servers to share the same data, and by reading this data, provide failover for resources.

The failover capability is achieved through redundancy across the multiple connected machines in the cluster. Redundancy requires that applications be installed on multiple servers within the cluster.

The cluster nodes can be active or passive:

- ▶ Active: when a node is actively handling requests
- ▶ Passive: when a node is on standby waiting for another node to fail

The Cluster Service is software that controls all aspects of a server cluster operation and manages the cluster database. Each node in a server cluster runs one instance of the cluster service. The cluster service can also be integrated with Active Directory; for example, if you publish a cluster virtual server as a computer object in the Active Directory, users can access the virtual server just like any other Windows 2000 server.

The management of a server cluster can be performed via the command line and the user can create, configure and administrate the server clusters. It can also be done using cluster.exe, which is a program, available in any Windows Server 2003, which is called by a command prompt and can be used to automate many administration tasks such as creating, configuring and administrating the server clusters.

Network Load Balancing

This is another technology included with Windows Server 2003 which is used for Web Services, Web servers, proxy servers, firewalls, and VPN. It is included in all versions of Windows Server 2003.

It distributes IP traffic to multiple copies (or instances) of a TCP/IP service, such as a Web server, each running on a host within the cluster. It load balances multiple server requests, from either the same client, or from several clients, across multiple hosts in the cluster. It is also designed to address bottlenecks caused by Web Services.

It scales up to 32 nodes and if a node in the cluster fails, Network Load Balancing automatically redirects incoming TCP traffic, User Datagram Protocol (UDP), or Generic Routing Encapsulation (GRE) requests to the remaining nodes.

Note: GRE traffic is not supported in Windows 2000, only in Windows Server 2003.

Network Load Balancing distributes the incoming network traffic among one or more virtual IP addresses (the cluster IP addresses) assigned to the Network Load Balancing cluster. The hosts in the cluster concurrently respond to different client requests, even multiple requests that come from the same client. For instance, a Web browser can get various images, each from a different host and load them into the Web page within a Network Load Balancing cluster. In terms of performance, it improves the response time of the requests to the clients.

The Network Load Balancing Manager allows you to create, configure and manage Network Load Balancing clusters and all the cluster's hosts from a single remote or local computer.

References for Server Cluster and Network Load Balancing can be found at:

<http://www.microsoft.com/windows2000/technologies/clustering/>

Component Load Balancing

This is a clustering mechanism supported in Windows 2000 and used to provide load balancing of COM+ components. In Component Load Balancing, the calls to activate COM+ components are load balanced to different servers within the COM+ cluster.

11.2 Performance

Performance involves minimizing the response time for a given transaction load. The performance of an application can be measured in terms of latency and throughput. Latency is the round-trip time between sending a request and receiving a response while throughput, when related to performance, defines the number of concurrent transactions that can be accommodated.

A good performance of an application is represented by a higher throughput and lower latency values. It is important to say that a number of factors relating to application design can effect performance. Also other factors such as network environment, messaging and transport protocols, memory, among others.

11.2.1 WebSphere

The vertical/horizontal techniques are useful to achieve a good performance of an application that runs on WebSphere.

- ▶ Vertical scaling involves creating additional application server processes on a single physical machine, each instance of application server running in its own JVM (Java Virtual Machine). Vertical scaling allows an administrator to profile an existing application server for bottlenecks in performance, and potentially use additional application servers, on the same machine, to get around these performance issues.
- ▶ Horizontal scaling involves creating additional application server processes on multiple physical machines to take advantage of the additional processing power available on each machine, increasing the performance of the application.

IBM WebSphere Application Server V5 provides tooling that enables intelligent end-to-end application optimization which means that administrators can handle volume and performance tuning dynamically.

A key factor in the performance of any Java application and hence any WebSphere Application Server application is the use of memory. Java does not require programmers to explicitly allocate and reclaim memory. The Java Virtual Machine (JVM) runtime environment is responsible for allocating memory when a new object is created, and reclaiming the memory once there are no more references to the object. This reduces the amount of coding required, as well as minimizing the potential for memory “leaks” caused by the programmer forgetting to deallocate memory once it is no longer required. Additionally, Java does not allow pointer arithmetic. Memory deallocation is performed by a thread executing in the JVM called the *garbage collector* (GC). The key to minimizing the performance impact of memory management is to minimize memory usage, particularly object creation and destruction. More details can be found in the IBM

Redbook *IBM WebSphere V5.0 Performance, Scalability, and High Availability WebSphere Handbook Series*, SG24-6198.

Java Management Extensions (JMX)

WebSphere's support for JMX provides Java components that will log and record statistics on usage and resources.

These objects are then exposed to third-party, JMX-compliant applications used for monitoring. Administrators can use best-of-breed tooling that is tightly integrated into their enterprise for managing performance data and the monitoring process.

Performance Monitoring Infrastructure

Performance Monitoring Infrastructure (PMI) is an API used by components of WebSphere to capture performance-related metrics in real time.

The PMI uses a client/server architecture. The server collects performance data on runtime and applications from various WebSphere Application Server components through Performance Monitoring Infrastructure (PMI). The performance data from one or more servers consists of counters such as servlet response time, data connection pool usage, among others, which are retrieved and processed by the client. The data can be monitored and analyzed with various tools.

The client application can be any of the following:

- ▶ A Graphical User Interface
- ▶ An application which monitors performance data and triggers different events according to the current values of the data.
- ▶ Any other application that needs to receive and process performance data.

For more details, please refer to the IBM Redbook *IBM WebSphere V5.0 Performance, Scalability, and High Availability WebSphere Handbook Series*, SG24-6198.

Tivoli Performance Viewer

Tivoli Performance Viewer is a Graphical User Interface (GUI) which retrieves the Performance Monitoring Infrastructure (PMI) data from an application server (in this case, WebSphere Application Server) and displays it in a variety of formats.

The Tivoli Performance Viewer provides access to a wide range of performance data for two kinds of resources:

- ▶ Application resources (for example, enterprise beans and servlets).
- ▶ WebSphere runtime resources (for example, Java Virtual Machine (JVM) memory, application server thread pools, and database connection pools).

This tool provides the ability to configure smart auto-tuning parameters which will automatically make recommendations to tune critical WebSphere parameters for maximized performance.

In order to monitor a resource with Tivoli Performance Viewer or any PMI or JMX client, you must enable the PMI service of the application server associated with the resource through the Administrative Console or by using the WebSphere Studio command interface.

There are some best practices considerations in the white paper *WebSphere Application Server Development Best Practices for Performance and Scalability* at:

http://www.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

11.2.2 .NET

The Windows platform also implements mechanisms that can improve performance of the applications and services.

- ▶ Vertical scaling refers to running a single application on a single server, with the ability to incrementally add system hardware resources such as processors and memory to increase overall system performance. The performance of an application depends on resources of the machine such as memory, CPU, processor, among others.
- ▶ Horizontal scaling means distributing the computing workload among multiple servers by clustering or load balancing. It improves the performance and availability of the overall service. The Microsoft platform supports horizontal scaling via server clustering and Network Load Balancing (NLB), described in “Network Load Balancing” on page 477.

Performance can also be increased with other mechanisms such as caching of the Web server, as we will see below.

Internet Information Services (IIS) caching

IIS 6.0 HTTP service (http.sys) gets all incoming HTTP requests providing high performance connectivity for Web applications. It introduces a kernel-mode driver which is responsible for connection management, bandwidth throttling, and Web server logging as well.

The http.sys implements a cache for HTTP responses which improves the performance. Both static and dynamic content can be cached, improving the performance of the applications. In previous versions of IIS (prior to V6.0), requests had to go from kernel mode to user mode for every dynamic request, and the responses had to be regenerated. By implementing a cache, the HTTP service handles caches HTTP responses in kernel mode with no transition to user mode, improving the performance.

IIS 6.0 has one feature called *Application Pool* which can be configured to optimize the performance of an application pool, allowing you to optimize the performance of your Web applications. An application pool can contain one or more applications and allows the isolation between different Web applications (the level of isolation can be configured). You can, for example, create an application pool for each ASP.NET application. Each application pool runs in its own worker process, so errors in one application pool will not affect the applications running in other application pools. Isolation of applications increases the reliability of the applications.

Web garden is an application pool which has multiple processes serving the requests routed to that pool. With Web garden, you can also increase scalability since a software block in one process does not block all the requests that go to an application.

Performance Monitor

Performance Monitor is a tool available in Windows which can be used for application tuning and by developers to customize Performance Monitor counters to diagnose problems. It allows sophisticated performance metrics across systems. Some counters that can be monitored using the Performance Monitor tool are:

- ▶ CPU utilization
- ▶ Current anonymous users
- ▶ Connections per second
- ▶ Requests

The System.Diagnostics namespace provides classes that allow you to interact with system processes, event logs, and performance counters, making it possible to instrument your .NET application, getting a detailed view of its performance regardless of where it is running.

The user is able to identify bottlenecks using tools such as Windows Task Manager, Windows Performance Monitor, and the Component Services administrative tool.

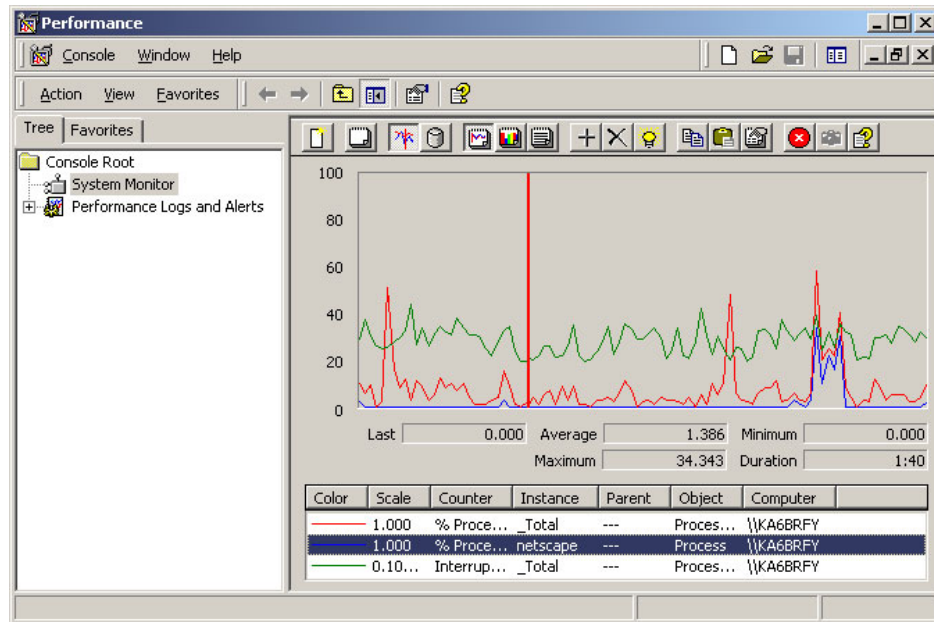


Figure 11-2 Performance Monitor screen

11.3 Availability

Availability is the measurement of the time the element is out of use, that is, experiencing an outage. Availability is usually expressed as a percentage of time the element is not out of service. A system, application, or component that can be used is considered to be available.

Another important definition is *high availability* which is the term usually associated with the ability to run for extended periods of time with minimal or no unplanned outage. In measurement terms, high availability is frequently considered to be 99.99% or greater. High availability typically involves horizontal scaling across multiple machines to avoid a single point of failure of a physical machine.

Continuous availability is the ability to provide high levels of availability all day, every day (24/7/365) with minimal planned or unplanned interruptions due to maintenance, upgrades, or failures.

Maintaining availability during system maintenance tasks, such as upgrades to hardware and applications, is a challenge. With the proper techniques, it is possible to achieve continuous availability and address the other requirements.

11.3.1 WebSphere

As we have mentioned before, WebSphere Application Server is designed to provide clustering. Clustering is a fundamental approach for accomplishing high availability. WebSphere Application Server Network Deployment V5.0 has a built-in server clustering technique (WLM).

WebSphere high availability should be considered as end-to-end system availability; this system includes a database, LDAP, firewall, Load Balancer, HTTP server, and WebSphere application server. This system is integrated with platform-specific clustering software to achieve maximum availability.

To avoid a single point of failure and maximize system availability, it is necessary to have some redundancy in the environment. Vertical scaling can improve availability by creating multiple processes, but the machine itself becomes a point of failure. High availability typically involves horizontal scaling across multiple machines; using a WebSphere Application Server multiple machine configuration eliminates a given application server process as a single point of failure.

- ▶ Horizontal scaling: provides the increased throughput of vertical scaling topologies but also provides failover support. This topology allows handling of application server process failures and hardware failures without significant interruption to client service.
- ▶ Vertical scaling: a vertical scaling topology provides process isolation and failover support within an application server cluster. If one application server instance goes offline, the requests to be processed will be redirected to other instances on the machine.

Single machine vertical scaling topologies have the drawback of introducing the host machine as a single point of failure in the system. However, this can be avoided by using vertical scaling on multiple machines.

Another important topic to mention is *failover*. Failover refers to the single process that moves from the primary system to the backup system in the cluster.

WebSphere Application Server Network Deployment V5.0 failover and recovery is also realized through the WebSphere Workload Management (WLM) mechanism. Other clustering software such as HACMP™, VERITAS Cluster Server, MC/ServiceGuard, Sun Cluster Server, and Microsoft Cluster Service can also be used to enhance application server failover and recovery.

11.3.2 .NET

Availability and high availability of .NET applications are related to the scalability mechanisms that we have mentioned previously.

Availability in .NET is mainly handled by Windows Server Clustering which we covered in “Windows Server 2003 Clustering” on page 475 and also “Network Load Balancing” on page 477.

Clustering is important because if one server goes down, the application is able to failover from one server to another. To avoid a single point of failure, a cluster should be considered for front-end (such as Web servers) and back-end servers (such as database servers). Besides clustering, Network Load Balancing is also important for high availability since it redirects the incoming network traffic to working cluster hosts if a host fails or is offline.

To guarantee availability of a .NET application, it is necessary to have redundant software, hardware and network connectivity for failover.

Monitoring the system is one of the factors that contribute to maintaining the environment free of failure; it can alert you when you need to increase the system’s capacity, find system bottlenecks, etc. There are some things that you can monitor, such as disk space, memory usage, CPU utilization, network load, application queues, errors, etc.

11.4 Security

In order to guarantee the system security, the components below should be considered:

- ▶ Hardware security, related to physically securing important systems.
- ▶ Network security, which refers to both physical and logical security. It involves tasks such as protecting an internal LAN or network from unauthorized access, making sure that no application is transmitting sensitive information across the network at any time, making sure each running service is accessible to only valid users, etc.
- ▶ Operating system security, protecting installed applications and their data files from illegal access.
- ▶ Application security, securing the resources on an application level and exercising the security features of the runtime platform (authentication and authorization).
- ▶ User/group management, monitoring permissions to ensure that applications are running as the correct user, adding system users as required, creating groups, etc.
- ▶ Resource permissions, making sure that the users are accessing only the resources they have access to.

We can group these components into physical and logical security. In this section, we are going to describe some features of both WebSphere and .NET regarding logical security.

11.4.1 WebSphere

WebSphere Application Server V5 provides the security infrastructure and mechanisms to protect J2EE resources and administrative resources and to address enterprise end-to-end security requirements for authentication, resource access control, data integrity, confidentiality, privacy, and secure interoperability.

WebSphere Application Server V5.0 is based on industry standards and provides a security model according to the J2EE 1.3 specification. It has an open architecture that allows secure connectivity and interoperability between Enterprise Information Systems (EIS) and other products from other vendors.

This topic is fully covered in the redbook *IBM WebSphere V5.0 Security WebSphere Handbook Series*, SG24-6573.

Security can be managed in terms of the application server environment or in terms of the application that is running in the server.

- ▶ Global security specifies the global security configuration for a managed domain and applies to all applications running on WebSphere Application Server. It determines whether security will be applied and sets up the User registry used to authenticate the users.
- ▶ Application security determines application-specific requirements. In some cases, these values may override global security settings, but in most cases they complement them. Application security includes such elements as a method for authenticating the users, a mechanism for authorizing the users into application-specific resources, roles-based access control to these resources, roles to user/user groups mapping, and so on. Application security is administered during the assembly phase using the Application Assembly Tool (AAT) and during the deployment phase using the Administrative Console and the WebSphere Studiomin client program.

This section discusses two fundamental security services also supported by WebSphere Application Server:

Authentication

WebSphere Application Server V5.0 supports two different types of authentication mechanisms:

► LTPA (Lightweight Third Party Authentication)

LTPA is supported in distributed environments where there are multiple application servers and machine environments. It supports forwardable credentials and Single Sign-On. It also supports cryptography which allows LTPA to encrypt and digitally sign, then securely transmit authentication related data and later decrypt and verify the signature.

Single Sign-On is the process with which the users provide their credentials, user identity, password and/or token, to connect to an application; then, these credentials are available to all enterprise applications that have Single Sign-On enabled without prompting the user for a user name and password again.

LTPA requires that the configured User registry be a central shared repository such as LDAP, a Windows domain type registry, or a custom registry.

The following LDAPs are supported by WebSphere Application Server V5.0.2:

- IBM Directory Server 5.1
- IBM OS/390® Security Server 2.10
- IBM SecureWay® Directory 3.2.2
- IBM z/OS Security Server 11, 1.2, 1.3 or 1.4
- Lotus Domino Enterprise Server 5.0.9a or 5.0.12
- Lotus Domino Enterprise Server 6.0.2 (on AIX, Linux/Intel, NT, W2K or Sun)
- Novell eDirectory 8.7
- Sun ONE Directory Server 5.0
- Windows 2000 Active Directory 2000

More information about supported hardware, software and APIs of WebSphere can be found at:

<http://www-3.ibm.com/software/webservers/appserv/doc/latest/prereq.html>

► SWAM (Simple WebSphere Authentication Mechanism)

This is useful for a single application server configuration and non-distributed environments. It does not support forwardable credentials or Single Sign-On. It relies on the session ID and supports SSL.

Authorization

WebSphere authorization is the J2EE authorization. The WebSphere resources are protected by the WebSphere authorization engine and the definitions are stored in XML files for each application.

WebSphere can also run with Java 2 security enabled, where the JVM itself takes care of authentication and authorization on the code level in the runtime environment.

WebSphere supports Java Authentication and Authorization Services API (JAAS), which provides the pluggable authentication mechanism for WebSphere; this makes the application independent of the authentication mechanism. It also supports user-based authorization. For more information, refer to the following URL:

<http://java.sun.com/products/jaas>

11.4.2 .NET

.NET Framework applications can use two different forms of security:

- ▶ Role-based security, also known as user security.

It allows the control of user access based on the user identity or its role in the application resources (for instance Web pages) and operations (for example business logic).

- ▶ Code-based security, also known as code access.

It controls which code can access the resources and perform the operations. It involves authorizing the applications to access the system-level resources, such as databases, directory services, the file system, etc.

In code-based security, the authentication is based on the evidence about the code, such as its strong name. The authorization is based on the code access permissions granted to the code by the security policy.

The following security namespaces are used to develop secure Web applications:

- ▶ System.Security
- ▶ System.Web.Security
- ▶ System.Security.Cryptography
- ▶ System.Security.Permissions
- ▶ System.Security.Policy
- ▶ System.Security.Principal

More details are on the Microsoft Web site:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh06.asp>

Authentication

ASP.NET supports several authentication schemes, including Microsoft Passport and forms-based, to control access to Web pages.

- ▶ Forms-based authentication

ASP.NET redirects unauthenticated users to an HTML form which is a custom login page. The user provides credentials such as username and password and if these are valid, the system issues an authentication cookie which allows the user to access the site. The cookie redirection and management is handled by ASP.NET. It is necessary to add code to verify the credentials and also to retrieve role membership from a user store.

- ▶ Passport authentication

It relies on the Microsoft Passport SDK and offers a single logon and core profile services for member sites.

- ▶ Client Certificate authentication

IIS supports the use of digital certificates and the Secure Sockets Layer (SSL). Either the server or the client has a certificate which is passed to the application with requests. Each request or response is validated by the authority of the certificate.

- ▶ Windows authentication

ASP.NET uses Windows authentication in conjunction with Microsoft Internet Information Services (IIS) authentication. IIS can perform the authentication in three different ways: basic, digest, or Integrated Windows Authentication (NTLM or Kerberos). Once the IIS authentication is done, ASP.NET uses the authenticated identity to authorize access.

- Basic authentication

The users are prompted to supply a user name and password which are returned to IIS and then can be used by the application. The password passed by the user to IIS is clear text, so it is not very secure.

- Digest authentication

It is similar to basic authentication, but uses encryption to send user information to the server. This one requires a Windows domain controller.

- Integrated Windows security

If the user is already authenticated in a Windows-based network, IIS can pass the encrypted token, indicating the user's security status when access to a specific resource is required.

External authentication through integration with Active Directory can also be used. Active Directory is a data store that contains information about objects such as users, groups, organizational units (OUs), computers, domains and

security policies. Active Directory is integrated with security through logon authentication and access control to objects in the directory.

Authorization

Once a user has been authenticated, the next step is to determine what he/she is allowed to do.

ASP.NET and IIS provide authorization points, also known as *gatekeepers*, within an ASP.NET application which can be used to control access to restricted resources:

- ▶ File Authorization

It is automatically active when Windows authentication is used. It runs an Access Control List (ACL) check of the .aspx or .asmx handler file to determine if a user should have access. File Authorization is performed by the *FileAuthorizationModule*.

- ▶ URL Authorization

The role-based security can be configured by using the `<authorization>` element in Machine.config or Web.config. This element controls which users and groups of users should have access to the application. Any application can use this authorization point. URL Authorization is performed by the *UrlAuthorizationModule*.

Note: Another security feature is the ability to control the identity under which code is executed. Impersonation is when ASP.NET executes code in the context of an authenticated and authorized client.

11.5 Transactionality

A transaction refers to the sequence of activities to be treated in a single unit of work. This means that all the activities have to be completed to make the transaction successful. When a transaction does not complete for any reason, the transaction has to be recovered back to a consistent state, so that all the changes made are rolled back.

The ACID properties of transactions are briefly described below:

- ▶ **Atomicity** - a transaction is an atomic unit of processing; this means that it is either performed entirely or not at all.
- ▶ **Consistency** - a correct execution of the transaction must take the system from one consistent state to another.

- ▶ **Isolation** - a transaction should not make its updates visible to other transactions until it is committed.
- ▶ **Durability** - once a transaction commits, the committed changes must never be lost in the event of any failure.

Transactional quality of service refers to the level of reliability and consistency at which the transactions are executed. Transactionality is essential for maintaining the integrity for the enterprise.

There are various approaches to providing transactional quality of service ,such as two-phase commit and compensation. Two-phase commit provides a transaction coordinator, which controls the transaction based on the idea that no constituent transaction is allowed to commit unless they are all able to commit. The problem is that the transaction coordinator does not have control over all resources.

Compensation is the other approach; it is based on the idea that a transaction is always allowed to commit, but its effect and actions can be cancelled after it has committed.

Compensation can be used in business processes to enable updates to be committed in several related transactions before the process has completed. If the process does not complete successfully, compensation is used to automatically perform actions that compensate for updates that have been committed. The compensation actions can roll back committed updates or can take alternative actions.

These approaches are important to guarantee that a complex request is executed either in its entirety or not at all.

11.5.1 WebSphere

This section discusses the transaction services on both platforms.

Transaction support

The way that applications use transactions depends on the type of application component involved.

- ▶ A session bean can either use container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (where the bean manages transactions itself).
- ▶ Entity beans use container-managed transactions.
- ▶ Web components (servlets) use bean-managed transactions.

WebSphere applications can be configured to interact with databases, JMS queues, and JCA connectors through their local transaction support when distributed transaction coordination is not required. The coordination of resource managers is also supported.

Resource managers that offer transaction support can be categorized into those that support two-phase coordination (by offering an XAResource interface) and those that support only one-phase coordination (for example, through a LocalTransaction interface). The WebSphere Application Server transaction support provides coordination, within a transaction, for any number of two-phase capable resource managers. It also enables a single one-phase capable resource manager to be used within a transaction in the absence of any other resource managers, although a WebSphere transaction is not necessary in this case.

With the Last Participant Support of WebSphere Application Server Enterprise, you can coordinate the use of a single one-phase commit (1PC) capable resource with any number of two-phase commit (2PC) capable resources in the same global transaction. At transaction commit, the two-phase commit resources are prepared, first using the two-phase commit protocol; if this is successful, the one-phase commit-resource is then called to commit (one-phase). The two-phase commit resources are then committed or rolled back, depending on the response of the one-phase commit resource.

The ActivitySession service of WebSphere Application Server Enterprise provides an alternative unit-of-work (UOW) scope to that provided by global transaction contexts. It is a distributed context which can be used to coordinate multiple one-phase resource managers. The WebSphere EJB container and deployment tooling support ActivitySessions as an extension to the J2EE programming model. EJBs can be deployed with life cycles that are influenced by the ActivitySession context as an alternative to the transaction context. An application can then interact with a resource manager through its LocalTransaction interface for the period of a client-scoped ActivitySession rather than just the duration of an EJB method.

WebSphere Application Server also provides support for compensation through the process choreographer, which we can define as a tool for executing complex business processes. It can be used for the choreography of all kinds of business processes or flows.

11.5.2 .NET

As J2EE, .NET also provides libraries for supporting transaction management. Microsoft offers transaction management support in MTS, COM+ and Common Language runtime technologies.

Microsoft Windows 2000 includes COM+ V1.0 with services such as transactions, and role-based security, among others. COM+ was upgraded from V1.0 to V1.5 and has been rebranded as Enterprise Services under .NET and Windows 2003 Server provides Enterprise Services. In this section, we will be talking about Enterprise Services, which are designed to support transactions and distributed processing in heterogeneous environments.

The Microsoft .NET Framework supports two transaction models for objects registered with Enterprise Services:

- ▶ **Manual transactions**

The developer is responsible for handling the transaction tasks such as begin, commit or abort, end and so on. The manual transaction model enables total control over the behavior of a transaction, which is a useful feature in cases such as managing nested transaction and linked transactions.

- ▶ **Automatic transactions**

The developer defines an object's transactional behavior by setting a transaction attribute value on an ASP.NET page, a Web Service method, or a class. Once an object is selected to participate in a transaction, it will automatically execute within the transaction's scope. An automatic transaction model is not suitable for handling nested transactions.

Transaction management

Microsoft Component Services provide an infrastructure for transaction management. In Windows 2000, the Component Services are comprised of two services:

- ▶ **COM+ (Enterprise Services)**

These allow you to configure and administer COM components and COM+ applications. Enterprise Services deals with transactions. As we already discussed, with transactions, the developer can ask for a transaction without worrying about how many resource managers will be involved, or how the components involved in the transaction interact. The transactions ensure the integrity of the application data by locking particular data records for a certain amount of time. This locking is based on the isolation level. At a higher isolation level, with more locking, there are minor chances of getting incorrect data.

In Windows 2000, COM+ services defaults to the highest isolation level that guarantees all work is completely isolated during a transaction. You can change the isolation level after the transaction begins with an SQL command.

In Windows 2003, Enterprise Services allow the specification of the isolation level that you want to use with the transaction by specifying the Isolation

property when you declare the transaction attribute. This allows you to define the isolation level that works best with the application.

► **Microsoft Distributed Transaction Coordinator (DTC)**

This allows you to administrate the distributed transactions and provides transaction services to ensure successful and complete transactions, even with system failures, process failures, and communication failures.

It coordinates transactions that update two or more transaction-protected resources such as databases, message queues, file systems, and so on. These transaction-protected resources may be contained on a single system or distributed across a network of systems.

The transaction manager is a component of the DTC and it is the process responsible for coordinating the MS DTC transactions.

A two-phase commit is also supported; in the algorithm for MS DTC, phase one involves the transaction manager requesting each enlisted component to prepare to commit and in phase two, if all components have successfully prepared, the transaction manager broadcasts the commit decision.

The performance of MS DTC transactions can be monitored using either the System Performance Monitor or the Component Services administrative tool.

11.6 Manageability

The use of either WebSphere or .NET has an impact on the way that the infrastructure is managed. There are operational concerns such as initial deployment of an application, maintenance, tuning, etc. Factors related to quality of service such as availability, security, performance can also be identified, reported and corrected using techniques and features provided with application servers to manage the infrastructure as well as the applications. They provide tools to monitor the applications in a preventive and corrective way.

Manageability is also related to administrative tasks for local and remote installation, configuration changes, applying fixes, application updates, etc.

In this section, we will focus on JMX used by WebSphere to provide management of the J2EE applications and WMI (Windows Management Instrumentation) for the .NET side.

11.6.1 WebSphere

Java Management Extensions (JMX) is a framework which provides a standard way of exposing Java resources (application servers, for example) to a system management infrastructure. The JMX framework allows a provider to implement

functions, such as listing the configuration settings, and allows users to edit the settings. It also includes a notification layer which can be used by management applications to monitor events such as the startup of an application server.

JMX allows the WebSphere users to use third-party management tools. JMX is a Java specification (JSR-003) that is part of J2SE 1.4.

The JMX architecture is divided into three levels:

- ▶ Instrumentation level
Dictates how resources can be wrapped within special Java beans, called Managed Beans (MBeans). The instrumentation of a resource allows it to be manageable through the agent level, which we will explain below.
- ▶ Agent level
Provides a specification for implementing agents. Management agents directly control the resources and make them available to remote management applications.
- ▶ Distributed services level
Provides the interfaces for implementing JMX managers. The distributed services level defines management interfaces and components that can operate on agents or hierarchies of agents.

Below are some key features of the implementation of JMX on WebSphere Application Server V5:

- ▶ All processes run the JMX agent.
- ▶ All runtime administration is performed through JMX operations.
- ▶ Connectors are used to connect a JMX agent to a remote JMX-enabled management application. The following connectors are currently supported:
 - SOAP JMX Connector
 - RMI/IIOP JMX Connector
- ▶ Protocol adapters provide a management view of the JMX agent through a given protocol. Management applications that connect to a protocol adapter are usually specific to a given protocol.
- ▶ A runtime object's configuration settings can be queried and updated.
- ▶ Application components and resources can be loaded, initialized, changed and monitored in the runtime.

Some of the more common JMX usage scenarios you will encounter are as follows:

► Internal product usage:

All the WebSphere V5.0 administration clients use JMX:

- WebSphere administrative console
- wsadmin scripting client
- Admin client Java AP

► External programmatic administration

In general, most external users will not be exposed to the use of JMX. Instead, they will access administration functions through the standard WebSphere admin clients.

However, external users will need to access JMX in the following scenarios:

- External programs written to control the Network Deployment runtime and its WebSphere resources by programmatically accessing the JMX API.
- Third-party applications that include custom JMX MBeans as part of their deployed code, allowing the applications components and resources to be managed via the JMX API.

Find more details in the redbook *IBM WebSphere Application Server V5.0 System Management and Configuration*, SG24-6195.

System management tools

The IBM WebSphere Application Server administration tools are used for system management for the entire distributed topology:

- WebSphere Administrative Console.
- Command-line operational tools.
- WebSphere scripting: IBM WebSphere Application Server includes a new scripting tool called wsadmin.
- Java APIs: the Java-based JMX APIs can be accessed directly by custom Java applications.

Although any of these interfaces can be used to configure system management functions, the use of the administrative console is preferred because it validates any changes made to the configuration.

Please refer to the IBM Redbook *IBM WebSphere Application Server V5.0 System Management and Configuration*, SG24-6195 for more information on this subject.

11.6.2 .NET

Microsoft provides a set of features built into the Microsoft Windows 2000 or 2003 operating system for building management information into the applications and for performing ongoing management activities.

Windows Management Instrumentation (WMI)

WMI is a key component of Microsoft Windows 2000 management services. It is also available for Windows 95, Windows 98, Windows NT 4.0, and Windows 2003. It is an object-oriented technology for programs to store information in a central repository, so it is a management infrastructure, a part of the operating system that provides application information and common management process for both local and remote application and operating system components. One example of its usage: WMI collects application failures, making it possible to determine the availability percentage of the applications.

Other programs or scripts can access the information in the central repository and call class methods to interact with the program and control its operation.

The management agent is a feature which is required by application instrumentation and used to monitor the local resources and publish data about the resource's current state and performance. Management agents also provide local configuration services as a way to make remote management possible.

There are many WMI management agents that help to collect application information, including agents for the following sources:

- ▶ Event Log
- ▶ Active Directory
- ▶ Registry
- ▶ Operating services
- ▶ Performance counters
- ▶ Application services
- ▶ HTTP requests, among others sources.

With WMI, it is possible to create these agents, collect and store information, view configuration, status, and operational data about the application and all of its supporting resources. New applications can also be developed using instrumentation. For more information on WMI, see *Managing Windows with WMI* at:

<http://msdn.microsoft.com/library/techart/mngwmi.htm>

Microsoft Operations Manager (MOM)

Manager provides event-driven management for the Windows platform and applications that implement Windows Management Instrumentation (WMI).

MOM data can be accessed by using any scripting or programming language supported by Windows Management Instrumentation (WMI) or Microsoft SQL Server. MOM exposes internal information about events, alerts, and computers through its WMI providers.

The MOM architecture includes various components and also includes user interfaces (consoles). These components are deployed in configuration groups which basically contain the following :

- ▶ MOM Database - a Microsoft SQL Server database that stores configuration and monitoring data.
- ▶ MOM DCAM - this is the central MOM server on which the MOM Administrator console is installed. The DCAM includes three components:
 - Data Access Server (DAS): controls the flow of data between the database, the Consolidator, the MOM Administrator console, and the Web console.
 - Consolidator: collects monitoring data from agents on managed computers and forwards this data to the DAS. The Consolidator also sends configuration changes to agent-managed computers.
 - Agent Manager: discovers, installs, and uninstalls agents on managed computers. The Agent Manager also passes configuration information to managed computers based on the processing rules for the configuration group.
- ▶ Agents - an agent is installed on each computer that is monitored by MOM. Agents collect monitoring data on a remote computer, apply processing rules to the collected data, and then send the resulting data to the Consolidator.
- ▶ MOM Administrator console - a MOM user interface that is used to monitor, configure, and deploy in the enterprise environment.
- ▶ MOM Reporting - an optional component used to generate reports based on the monitoring data that MOM collects. It generates reports with information on capacity planning, performance analysis, and application-specific reports that you can use to monitor resources, traffic, and availability.

Another tool which is available is the Systems Management Server, which is a Windows-based product designed to manage, support, and maintain a distributed network of computer resources and to help automate change and configuration management.

For more information, see the Systems Management Server Web page at:

<http://www.microsoft.com/smsgmt/default.asp>

11.7 Maintainability

This term refers to the maintenance of the environment that the application is running in both hardware and software perspectives. How easy is it to repair a machine that runs a critical application ? How do you modify a software system or a component to correct faults, improve performance or adapt to a changed environment without stopping the running application ? These are the common questions that come up when we talk about maintainability.

Maintainability is related to some other topics that we covered in this chapter, such as availability. Application maintainability refers to deployment of new applications and updates of the existing ones.

11.7.1 WebSphere

WebSphere Workload Management provides features such as clustering, which allows servers to be maintained and upgraded in a way that is transparent to the users without having to stop the application.

In a distributed environment, where we physically separate some components, due to the components' independence, the server components can be reconfigured, or even replaced, without affecting the installation of other components on separate machines.

Some considerations about vertical and horizontal scaling come into play:

- ▶ It is easier to administer the member application servers in a vertically scaled topology because all the code is running on the same machine, which is therefore a single point of administration. This topology can also easily be combined with other topologies. We can implement vertical scaling on more than one machine in the configuration; this requires IBM WebSphere Application Server Network Deployment to be installed.
- ▶ Horizontal scaling requires code migration to multiples nodes; there is more installation and maintenance associated with the additional machines.

One of the advantages of horizontal scaling is that if you have to upgrade the hardware, the application will still run on the other machine.

For application maintainability, WebSphere Application Server provides support for dynamic deployment and hot deployment.

- ▶ Dynamic deployment is the ability to deploy new code and/or change an existing component without restarting the server or stopping the application in order for the change to take effect.

- ▶ Hot deployment is the process of adding new components (such as WAR files, EJB jar files, enterprise Java beans, servlets, and JSP files) to a running server without having to stop the application server process and start it again.

A best practice for maintenance is to keep the EAR files in sync. It is likely that you will use a tool such as Rational ClearCase to manage the code. Basically, you should always be able to retrieve from your configuration management tool the exact configuration which runs in production or testing.

11.7.2 .NET

Within the .NET Framework, you can use some tools to deploy applications and manage the code. In this section, we will briefly describe the available tools that can be used to manage .NET code.

Source code management

Source code management in the .NET Framework is possible by using Microsoft Visual Source Safe (VSS), which is a repository used to manage project files. Visual Source Safe is integrated with Visual Studio .NET.

The user can see the latest version of any file, make changes, and save a new version in the repository. By performing a check-in or check-out, the user can get or save the files, respectively.

XCOPY deployment

.NET Framework applications do not need registry entries and the assemblies (*.dll, *.exe) are self-describing via their meta-data; they can be simply deployed using an XCOPY command or FTP.

11.8 Portability

There are two flavors of portability: platform-neutral and application portability.

Platform-neutral refers to the ability to move a code base from one operating system to another without having to change the code itself. This is a possibility with most implementations of J2EE. A key difference between J2EE and .NET is that J2EE is platform-neutral, running on a variety of hardware and operating systems, such as Win32, UNIX, and mainframe systems. This is possible because the Java Runtime Environment (JRE) is available on any platform.

Application portability is a key feature of the J2EE platform; it means that a J2EE application can run on any J2EE-compatible application server.

Many J2EE implementations such as IBM WebSphere Application Server provide and support vendor-specific extensions to the J2EE. These extensions are intended to provide functionality that is not part of the J2EE specification to the applications. Many extensions delivered with WebSphere Application Server become part of the J2EE specification in its further releases.

11.8.1 WebSphere

WebSphere Application Server runs under several platforms, as listed below; this means that you can deploy the same Java application in any WebSphere Application Server running on these platforms without having to make any change in the code.

- ▶ AIX 4.3.3 4330-10 Maintenance level
- ▶ AIX 5.1 5100-02 or 5100-03 Maintenance level
- ▶ AIX 5.2
- ▶ Windows 2000 Advanced Server Service Pack 3
- ▶ Windows 2000 Server Service Pack 3
- ▶ Windows 2003 Server, Enterprise
- ▶ Windows 2003 Server, Standard
- ▶ Windows NT 4.0 Service Pack 6.0a
- ▶ Red Hat Linux 8
- ▶ Red Hat Enterprise Linux WS/ES/AS for Intel 2
- ▶ SuSE Linux 7.3 for Intel
- ▶ SuSE SLES 7 2.4
- ▶ UnitedLinux 1.0
- ▶ OS/400 5.1
- ▶ OS/400 5.2
- ▶ Red Hat Linux 7.2 for s/390
- ▶ SuSE SLES 7 for zSeries
- ▶ HP-UX 11iv1
- ▶ Solaris 8
- ▶ Solaris 9

For the most current information on the supported software releases, operating systems, and maintenance levels, see:

<http://www.ibm.com/software/webservers/appserv/doc/latest/prereq.html>

11.8.2 .NET

The J2EE specification supports only Java as a programming language and runs on any platform. .NET supports multiple programming languages and .NET applications only run on the Microsoft platform.

For server-side applications, the following versions of Windows are supported:

- ▶ Microsoft Windows 2000 Professional with Service Pack 2.0
- ▶ Microsoft Windows 2000 Server family with Service Pack 2.0
- ▶ Microsoft Windows XP Professional
- ▶ Microsoft Windows Server 2003 family

11.9 Web Services

A typical Web Services application will execute in a runtime environment which, in turn, executes on an operating system. Hence, this Web Services application would inherit quality of service already provided by its underlying environments. If the Web Services application is running in WebSphere Application Server, this Web Services application can take advantage of scalability, availability, reliability and other quality of service provided by WebSphere Application Server.

Since both WebSphere and .NET solutions provide similar support for the open standards for the Web Services technology, such as SOAP, WSDL, and UDDI, quality of service will become an important selling and differentiating point of these services.

In this section, we will look at some Web Service QoS requirements, such as performance, security, transactionality, and reliability.

Performance considerations

We can consider some aspects to measure the performance of a Web Service, such as latency and throughput, which were covered previously, as well as execution time (which is the time that a Web Service takes to process a sequence of activities) and transaction time (which is the period of time that the Web Service takes to complete a transaction).

There are many factors that can influence the performance of a Web Service, such as network, application logic, encoding styles and also the underlying messaging and transport protocols used. Of course, there are some factors that are outside the control of the Web Service, such as Web server response time and availability, original application execution time in the Web application server, and back-end database or legacy system performance. The performance of these can be improved using techniques such as caching, load balancing on both the Web server and Web application server sides.

The *encoding style* determines the scalability and performance of a Web Service. Some performance tests between SOAP RPC and Document style encodings were performed using an open-source utility called TestMaker; they provided some information regarding scalability and performance, as we will see.

Figure 11-3 shows that as the payload size increases, the number of transactions per second decreases when using SOAP RPC encoding. The payload size is measured in bytes.

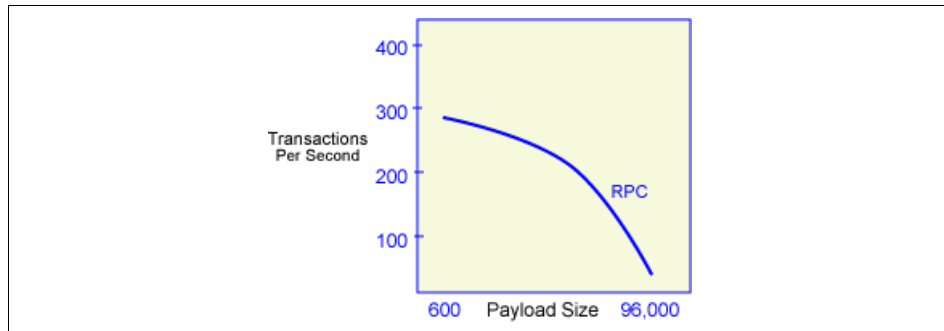


Figure 11-3 Performance when using SOAP RPC encoding

The next test that was performed used SOAP Document style encoding. The performance stays relatively stable when the payload size is increased.

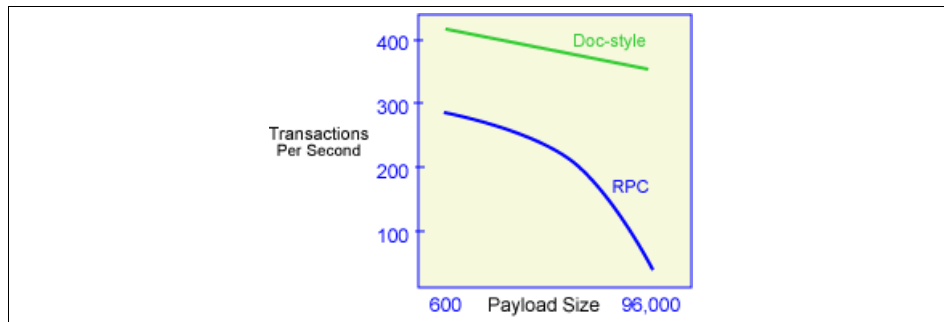


Figure 11-4 Performance when using Document style encoding

The test using RPC literal encoding shows that it provides the performance benefits of SOAP Document style encoding with a little more work due to the parsing of the XML data.

In terms of underlying messaging and transport protocols, let's talk about SOAP. SOAP uses XML to encode messages; it is easy to process messages at every step of the invocation process. At a glance, it seems that an XML-based scheme would be intrinsically slower than that of a binary based model, but it is not as straightforward as that. First, when SOAP is used for sending messages across the Internet, the time to encode/decode the messages at each endpoint is very short compared to the time to transfer the bytes between endpoints, so using XML in this case is not significant.

Secondly, in an intranet environment, it is very likely that the endpoints of a SOAP message are running the same implementation of SOAP parsers.

The SOAP protocol uses various steps to process a complete communication style and the whole process requires various levels of XML parsing. The parser is also important in SOAP performance. The SAX-based SOAP implementations can be used to increase throughput, reduce memory overhead and improve scalability.

References

Learn about SOAP encoding's impact on Web Service performance at:

<http://www-106.ibm.com/developerworks/webservices/library/ws-soapenc/>

Understanding quality of service for Web Services by referencing:

<http://www-106.ibm.com/developerworks/java/library/ws-quality.html>

Security

Quality of service is required to provide security, reliable messaging, and management for each layer of the Web Services stack. To secure Web Services, you must consider a broad set of security requirements, such as:

- ▶ Authentication
- ▶ Authorization
- ▶ Privacy
- ▶ Trust
- ▶ Integrity
- ▶ Confidentiality
- ▶ Secure communications channels
- ▶ Federation
- ▶ Delegation
- ▶ Auditing across a spectrum of application and business topologies

One of the key requirements for the security model in today's business environment is the ability to interoperate between formerly incompatible security technologies (such as public key infrastructure, Kerberos, etc.) in heterogeneous environments (such as .NET and J2EE). The complete Web Services security protocol stack and technology roadmap is described in *Security in a Web Services World: A Proposed Architecture and Roadmap*, which is available at the following address:

<http://www.ibm.com/developerworks/webservices/library/ws-secmap/>

Web Services security is a SOAP message-level security specification used to support security token propagation, message integrity, and message

confidentiality. One intent of the specification is to address interoperability between different implementations of Web Services security.

WS-Security provides a general purpose mechanism for associating security tokens with messages; it is designed to be extensible (for example, to support multiple security tokens).

You can find the WS-Security specification and also some security considerations at:

<http://www.ibm.com/developerworks/library/ws-secure/>

Other mechanisms can be used to provide security to the Web Service application, such as:

- ▶ Leveraging transport layer security:
 - HTTPS - ensuring integrity and privacy
 - HTTP - basic authentication headers

The features of your reverse proxy, Web server and WebSphere application server can be fully leveraged because the SOAP server (rpcrouter) is just another servlet.

- ▶ Enhancing the SOAP message layer with security features:
 - XML signatures, addressing the integrity requirement
 - XML encryption, addressing the privacy requirement

To realize the benefits of Web Services security, it is recommended that an implementation of the specification be integrated with underlying security mechanisms.

This implementation is fully integrated with the WebSphere Application Server V5.0.2 security infrastructure. Authorization, for example, is based on the J2EE security model. When a user ID and password are embedded in a request message, authentication is performed with the user ID and password. If successful, a user identity is established in the context of execution and further resource access is authorized based on that identity. Once the user ID and password are authenticated by the Web Services security runtime, a J2EE container performs the authorization.

WebSphere Application Server V5.0.2 provides the following capabilities for Web Services security:

- ▶ Integrity of the message
- ▶ Authenticity of the message
- ▶ Confidentiality of the message
- ▶ Privacy of the message

- ▶ Transport level security: provided by Secure Sockets Layer (SSL)
- ▶ Security token propagation (pluggable)
- ▶ Identity assertion

Transactionality

This refers to the level of reliability and consistency at which the transactions are executed. It is crucial for maintaining the *integrity* of a Web Service.

The Web Services Coordination and Web Services Transaction specifications complement BPEL in that they provide a Web Services based approach to improving the dependability of automated, long-running business transactions in an extensible, interoperable way.

The WS-Coordination specification defines a framework through which those services can work from a shared "coordination context." WS-Transaction, on the other hand, provides a framework that allows the monitoring of the success or failure of each individual, coordinated activity.

WS-Transaction defines two transaction coordination types:

- ▶ Atomic transactions (AT), where the results of operations are not made visible until the completion of the unit of work. They have the following characteristics:
 - Short duration
 - Typically executed within limited trust domains

The atomic transactions are called atomic because they have an "all or nothing" property. Figure 11-5 on page 506 shows that the Web Service WSA calls Web Service WSb in the same atomic transaction. The persistent updates made by WSA and WSb must be committed or aborted together, regardless of system or network failure.

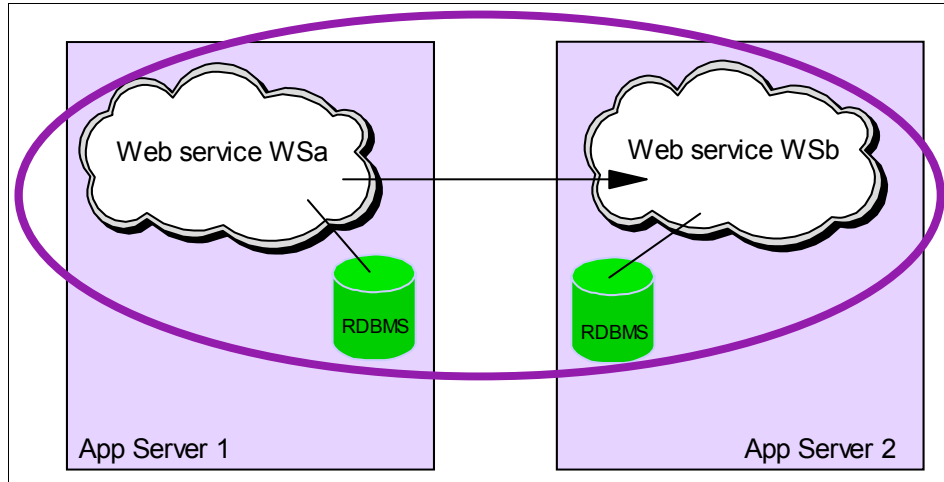


Figure 11-5 Atomic transaction

A two-phase commit atomic transaction (AT) protocol is used to coordinate the persistent updates. It is important to remember that database locks may be held for the duration of the transaction.

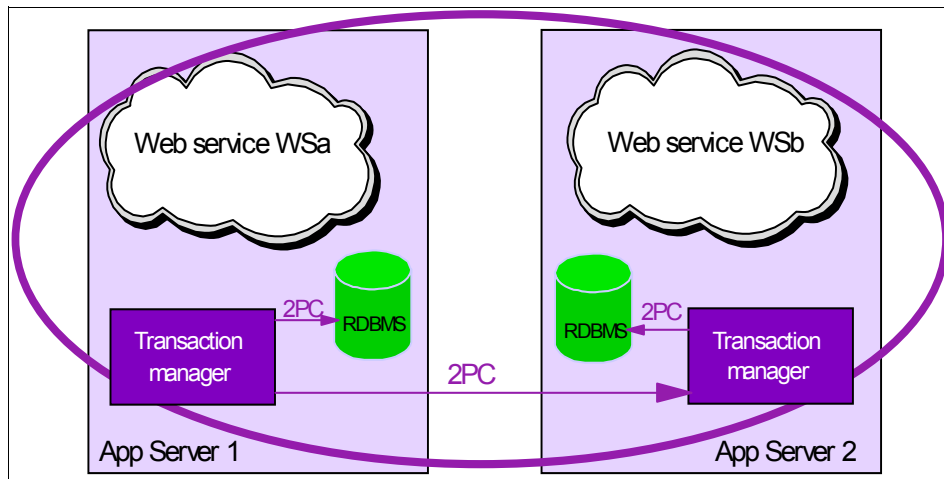


Figure 11-6 Two-phase commit in atomic transactions

- Business activities (BA), where the results of operations are made visible before the completion of the unit of work and where business logic is needed to handle unsuccessful completion, for example as part of a compensation.

These have the following characteristics:

- May have a long duration
- More appropriate for business transactions that span trust boundaries

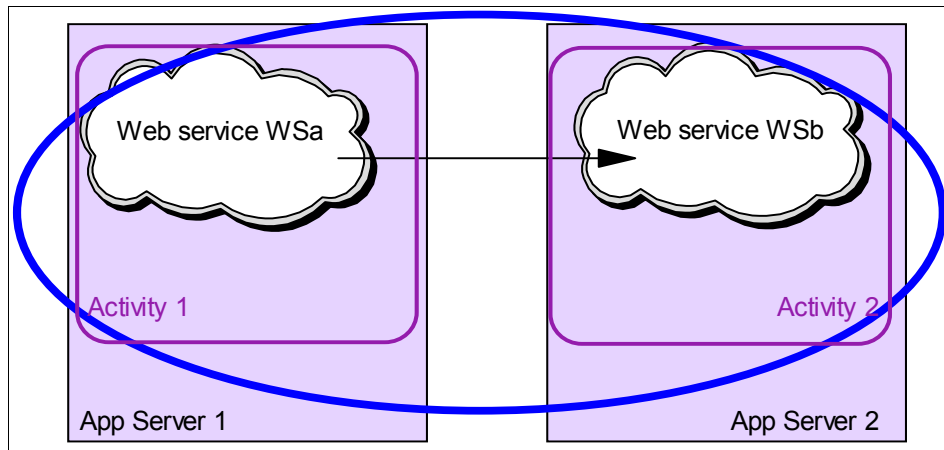


Figure 11-7 Business activities

The figure above shows the Web Service WSa calling Web Service WSb in the same long-running business transaction. They both are loosely coupled and their business activities may independently fail. Recovery tasks, such as compensation, are part of the overall business application.

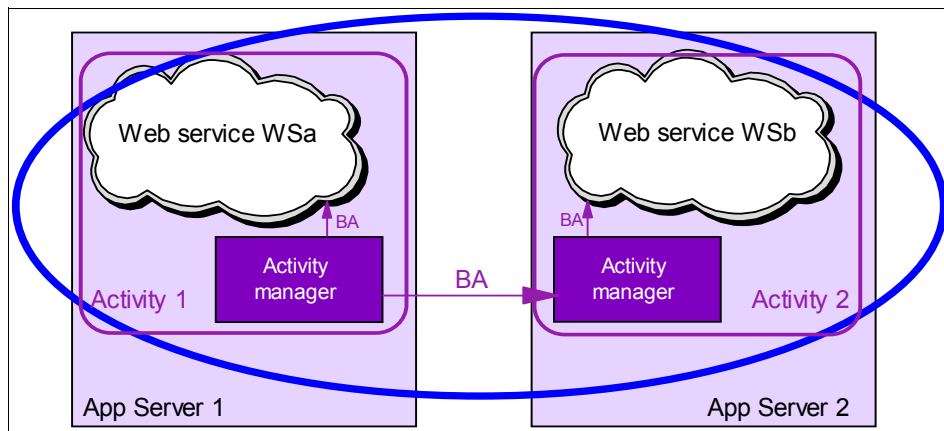


Figure 11-8 Complete business activities

Following are some considerations about the figure above:

- A business activity (BA) protocol is used to coordinate the business activities.
- Database locks may be held for short periods during transactional periods of the longer-running business activities.
- Middleware provides the infrastructure to drive compensation as required.
- Compensation logic is provided by the application.

The Atomic Transaction specification defines protocols which enable existing transaction processing systems to wrap their proprietary protocols and interoperate across different hardware and software vendors.

The specifications were published jointly by IBM, Microsoft and BEA:

<http://www.ibm.com/developerworks/webservices/library/ws-coor/>
<http://www.ibm.com/developerworks/webservices/library/ws-transpec/>

Please refer to the IBM Redbook *WebSphere MQ Solutions in a Microsoft .NET Environment*, SG24-7012 for details about Web Services transactions.

Reliability

Reliability is the quality aspect of a Web Service that represents the degree to which it is capable of maintaining the service and service quality. The number of failures per month or year represents a measure of reliability of a Web Service. In another sense, reliability refers to the assured and ordered delivery of messages being sent and received by service requestors and service providers.

To improve reliability, the applications which rely on remote Web Services can use message queuing. Applications and Web Services within an enterprise can use message queuing such as Java Messaging Service (JMS) or WebSphere MQ for Web Service invocations. Enterprise messaging provides a reliable, flexible service for the asynchronous exchange of critical data throughout an enterprise. Message queues provide two major advantages:

- ▶ They are asynchronous.
- ▶ They are reliable, since the messaging service ensures that the message is delivered only once.

WS-I

WS-Interoperability is an open industry organization chartered to promote Web Services interoperability across platforms, operating systems, and programming languages. The organization works across the industry and standards organizations to respond to customer needs by providing guidance, best practices, and resources for developing Web Services solutions.

It defines a Basic Profile specification which consists of a set of non-proprietary Web Services specifications to promote interoperability.

You can find more details at:

<http://www.ws-i.org/>



Part 4

Appendixes



Lotus Domino and .NET coexistence

Lotus Domino and Microsoft .NET technologies can be integrated using Web Services. Web Services are self-contained, self-describing, modular applications that can be published to and invoked from the Web. Unlike traditional Web-based applications, Web Services contain no user interface, which means that these applications could be remotely invoked by other applications known as *clients*. Web Services use technology standards such as XML, SOAP, WSDL, and UDDI. In addition to this, Web Services applications communicate with each other by using the HTTP protocol and SOAP messages.

Lotus Domino is ideal software for using Web Services to extend the collaborative features of Notes/Domino across the enterprise. Domino Application Server can either host or use Web Services. As we have also discussed in this redbook, Microsoft .NET platform is a collection of tools from Microsoft that let you both use and host Web Services. This appendix describes how to integrate both technologies.

A.1 Web Services integration

It is not the purpose of this redbook to introduce Web Services concepts and their architecture (you can find detailed information about these topics in the redbook *WebSphere Version 5 Web Services Handbook*, SG24-689), but to show how Web Services, developed on different languages and different tools, can interact with each other.

The principal elements involved on the interoperability of Web Services are the following:

- ▶ A service provider
- ▶ A service requester
- ▶ A way to code the data - XML language
- ▶ A way to define and describe the service - WSDL document
- ▶ A way to format remote calls - SOAP Protocol
- ▶ A network protocol - HTTP

Also in a runtime environment, we need:

- ▶ A Service Broker known also as a Service Registry.
- ▶ A way to publish and find services - UDDI

The Web Services model includes three roles: the service provider, the service broker, and the service requester and the methods and properties are associated with the service. The Web Service is published in an external or internal registry using UDDI. Once the Web Service is publicly or privately available in the appropriate UDDI registry, the service requester uses UDDI to find the Web Service and consume it. SOAP is used to invoke a Web Service, therefore binding the service requester to the service provider.

As mentioned before, Lotus Domino Application Server and Microsoft .NET platform can host or provide Web Services. This means that it is possible to use Lotus Domino Server to host Web Services as a service provider, because Domino Applications can be modified to provide SOAP Interfaces and WSDL Descriptions using XML, and use .NET clients to invoke the Lotus Domino Service as a consumer or the other way around, using Lotus Domino as a consumer (Service Requester), which involves calling or invoking the .NET service and getting the response back.

Therefore, the purpose of this section is to explain the following scenarios:

- ▶ Lotus Domino as a provider and .NET as a consumer.
- ▶ .NET as a provider and Lotus Domino as a consumer.

A.1.1 Domino provider, .NET consumer

As explained above, Lotus Domino applications can have a Web Service interface that allows it to be accessed as a Web Service by remote users or by Web server clients. The design elements needed to provide a Domino Web Service are the following:

- ▶ A *Lotus Script Web Agent*: this agent is written to accept a SOAP request, parse it, call the requested method (function), and return the result as a SOAP response to the requester.
- ▶ Any standard LotusScript function stored in a *Lotus Script library*.
- ▶ A *page containing a WSDL definition of the service*. This is required only because the database is going to be accessible by the .NET consumer.

The application components are represented in the following diagram.

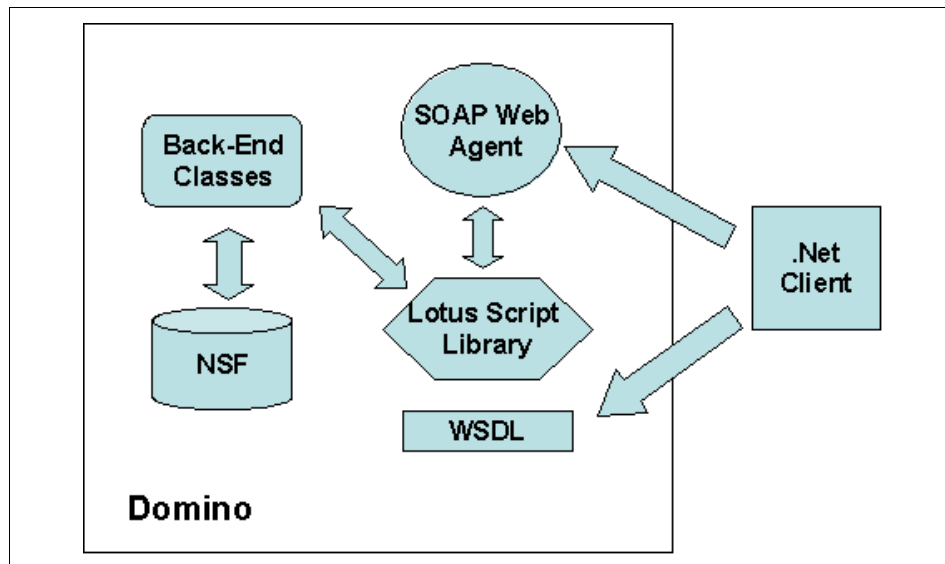


Figure A-1 Domino - .NET client interaction

To explain how to use Domino as a Web Service Provider, we have created a Notes Sample Web Services .NET Database Application (WebServiceNet.nsf). This database shows how you can easily write a 100% Lotus Script Web Service that allows .NET clients to access the details of a particular upcoming ITSO Residency by giving a Residency Code.

Note: The example of this redbook is created based on the Building Web Services using Lotus Domino 6 Tutorial which was developed by IBMdeveloperWorks and can be located in the following URL: https://www6.software.ibm.com/reg/devworks/dw-1sdom6ws-i?S_TACT=103A MW13&S_CMP=LDD. More information regarding developing Web Services in Domino can be found in Lotus Domino Designer® 6 Help database.

For the development, we used the following product versions:

- ▶ Microsoft .NET Framework Software Development kit V1.1
- ▶ Microsoft Windows 2000 Server with Service Pack 4
- ▶ Lotus Domino Server V6.0.2 CF2
- ▶ Microsoft IE V5.5 or newer
- ▶ Lotus Domino Administration Client V6.0.2.
- ▶ Lotus Domino Designer Client V6.0.2.

For more details on how to install the software products, refer to the installation manuals. Lotus Domino Administrator client and Lotus Domino Designer was installed in another machine in order to administrate the Domino Server and to design the Sample application.

Before starting, make sure that you have TCP/IP network configured (it is recommended to have a fixed IP address), that it is possible to resolve the machine host name (via Host file or DNS) and also that you have Domino Server configured. For our example, the following Domino nomenclature was selected within the configuration process, but of course it is possible to use another one.

Table A-1 Domino nomenclature

Concepts	Selected Name
Domino Domain Name	TEST
Organization Name	TEST
Server Name	Domin6/TEST
Server Title	Test Server
Notes Network Name	TCPIP Network
Notes Administrator Name	Notes Admin/TEST

Note: For more details about configuring a Domino Server, refer to the Lotus Domino Administrator 6 help database.

Once you have installed and configured the products to begin with the example, follow these steps:

1. Create a new database.
2. Create the Forms and Views for the database.
3. Create a Lotus Script Web Agent (ResidencyWS).
4. Create a Lotus Script Library (Domino).
5. Create a WSDL page to describe the Domino Web Services.
6. Create a .NET client.
7. Test the application.

A.1.1.1 Creating a new database

Although included in this redbook is the sample database inside the additional material, we are going to start from scratch by creating a new database in our server.

Open Lotus Domino Designer 6 and select **File -> Database -> New** from the menu. This opens the New Database dialog box as shown below.

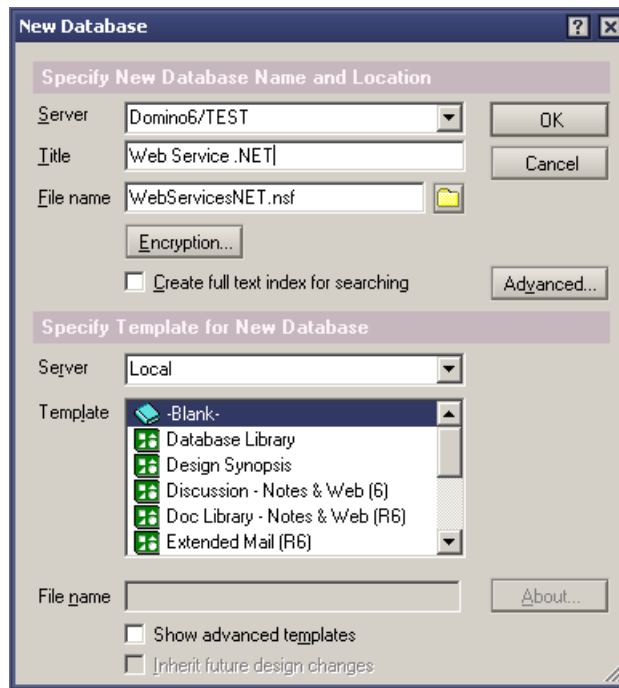


Figure A-2 Create database

In the Server field, select: **<The name of the Domino Server>** (in our case **Domino6/TEST**), as Database Title type: Web Service .NET and for File name: WebServiceNET.nsf. Select **-Blank-** as the template. Click the **OK** button.

Note: To use the sample database included in the additional material of this redbook, is necessary to sign the database using an ID with administrative privileges; follow the instructions provided in 1.2.1, “Writing a Java application using a text editor” on page 22 using Domino as a COM server, and .NET as a client in the “Create a Domino Sample Database” part. Be aware also that by default, Web agents run under the identity of the agent author (the person who saved the agent) or the agent signer so this user must have access to run Agents on the Domino Server.

A.1.1.2 Creating the Forms and Views for the database

When a new database is created from scratch, it does not contain design elements except one default view, so it is necessary to create all of the elements needed for the application. Because the sample database is going to contain all the details for the upcoming ITSO residencies, the minimum design elements to create are:

1. A new Form to provide the structure for creating and displaying ITSO Residency documents details.

Open Lotus Domino Designer 6 and select **Forms** in the Design pane, then click **New Form**; an untitled blank form is displayed. Add the necessary fields for display the residency information details such as: *Residency Name*, *Residency Code*, *Start date*, *End Date*, *Residency Contact*, *e-mail* and *Location*. Save the form with the Residencies name. The New Form could look like the figure below.

The screenshot shows a web form titled "Residency Details" within a Lotus Domino Designer 6 environment. The form is designed with a header area containing a logo and the title. Below the header, there are several input fields arranged in a table-like structure. The fields are: Residency Name (text input), Residency Code (text input), Start Date (date input), End Date (date input), Residency Contact (text input), e-mail (text input), and Location (text input). At the bottom of the form, there is a section labeled "Hidden Fields:" which contains two hidden text inputs: SDate and EDate.

Residency Name:	<input type="text" value="Name"/>
Residency Code:	<input type="text" value="Code"/>
Start Date:	<input type="text" value="StartDate"/>
End Date:	<input type="text" value="EndDate"/>
Residency Contact:	<input type="text" value="Contact"/>
e-mail	<input type="text" value="email"/>
Location	<input type="text" value="Location"/>
<p>Hidden Fields:</p> <input type="text" value="SDate"/> <input type="text" value="EDate"/>	

Figure A-3 Create Residencies Form

2. A new View for access to the list of Residency documents is created in the database.

Open Lotus Domino Designer 6 and select **Views** in the Design pane, and then click **New View** or modify the default view initially created. Add the following columns: Name, Residency Code, Start Date, Contact, email and Location. Save the View with the name Residencies. The new view could look like the figure below.

Name	Residency Code	Start Date	End Date	Contact
Creating a Demo - Enable Company Website	SM-4138-R01	06/10/2003	31/10/2003	Jon Williams
IBM WebSphere and Microsoft .NET Coexistence	SA-W324	25/08/2003	04/10/2003	Peter Kovar
WebSphere - Problem Determination Across Multiple	SA-H307-R01	10/11/2003	12/12/2003	Peter Kovar

Figure A-4 Residencies View

- Also, create a hidden View called (By Code) with the first column ordered by Residency Code. This hidden view is where the Domino Web Service is going to access for locating the residency.

A.1.1.3 Creating a Lotus Script Web Agent

After the database is created, to route the SOAP Messages Request to the appropriate function inside the Lotus Script Library, a Lotus Script Web Agent is needed. Open the Lotus Domino Designer.

- In the left pane, select **Shared Code -> Agents**. Click the **New Agent** action button. The Agent properties dialog box is displayed.

Agent	
Name	ResidencyWS
Comment	Route SOAP Messages Request
Options	<input checked="" type="radio"/> Shared <input type="radio"/> Private <input type="checkbox"/> Store search in search bar menu <input checked="" type="checkbox"/> Store highlights in document <input type="checkbox"/> Run in background client thread
	Trigger <input checked="" type="radio"/> On event <input type="radio"/> On schedule Action menu selection: [Action menu selection]
	Target: [None]
	@Commands may be used in this type of agent

Figure A-5 Web agent

Give ResidencyWS as the Domino Web Agent Name and select **Shared** as the Agent Option. Set the agent trigger to the On event and also set Action menu selection. The last thing to do is to set the Agent target to None which means that the agent is going to work on fields of the current document such as those launched from WebQueryOpen or WebQueryClose, or like a form action or hotspot that also works on fields in the current document.

First of all, declare the incoming and response SOAP Messages variables as string.

```
'Declare the response as String
Dim response As String
'Declare the incoming SOAP Message as String
Dim SOAPin As String
```

- The next step is to create a NotesSession object by declaring the variable session and setting it as New to create a new instance for that object. Then, initialize the object variable *doc* using the *DocumentContext* property of the NotesSession class. The agent can use this property to access the in-memory document. The next line of code sets the SOAPin variable equal to the content of the "Request_content" field using the *GetItemValue* method of the NotesDocument class. This is where the SOAP message resides as a result of a "Post," in the DocumentContext object.

```
Dim session As New NotesSession
Set doc = session.DocumentContext
SOAPin = doc.GetItemValue("Request_content")(0)
```

3. For debugging purposes, a Log Message Function was created, with the objective of writing a new document in the database with the incoming SOAP Message. For that purpose, a new Form and a new View were added to the database:

Messages View: the view that displays all the SOAP incoming Message documents created.

Message Form: used for creating a document with the SOAP incoming Message every time the Domino Web Services is accessed.

After the message is logged, a RemoveWhitespace function is used to remove all spaces, tabs, and new line characters as shown below:

```
LogMessage(SOAPin)
SOAPin = RemoveWhitespace(Fulltrim(SOAPin))
```

Note: For more details about the Log Message and RemoveWhitespace functions created only for debugging purposes, refer to the sample database included in the additional material.

4. A SOAP Message consists of the following parts: a SOAP Envelope which marks the beginning and end of a message and a SOAP Body inside the SOAP Envelope which includes the method signature to be executed and the method arguments. The next piece of the code manually parses the SOAP message using the Lotus Script Language string handle functions (Instr and Mid) to extract the following items.

Table A-2 SOAP Message table

Item	Variable Stored In	Definition
namespace	NameSpace	Script Library to load.
method	MethodName	Function to execute in the script library.
argument	argValue	Parameter to pass to function as the aString variable.

Later, for parsing the SOAP content more efficiently, we will use the new NotesDOMParser object. Note also the first line (On Error Goto ErrHandle), which determines how an error will be handled in this case.

```
On Error Goto Errhandle
bodyPos= Instr(1,SOAPin,|<soap:Body
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">|)+74
'parse out method signature
methodSigPos= Instr(bodyPos,SOAPin,|<|)+1
methodSigEnd=Instr(bodyPos,SOAPin,| |)
methodSignature = Mid(SOAPin,methodSigPos,(methodSigEnd-methodSigPos))
```

```

'parse out method name
methodPos= Instr(bodyPos,SOAPin,|:|)+1
methodEnd=Instr(methodPos,SOAPin,| |)
methodName = Mid(SOAPin,methodPos,(methodEnd-methodPos))
'parse out namespace
nameSpacePos= Instr(methodEnd,SOAPin,|uri:|)+4
nameSpaceEnd=Instr(nameSpacePos,SOAPin,|")|)
nameSpace=Mid(SOAPin,nameSpacePos,(nameSpaceEnd-nameSpacePos))

```

5. The next code slice maps the namespace, method, and argument from the SOAP request to the script library, function, and parameter (respectively) called by the Web agent, and captures the return value in the response variable.

```

callString = |Use | & "| & nameSpace & "| & |
response = | & methodName & |(SOAPin)|

```

Execute the callString variable that makes the specified script library run.

Execute callString

6. The next step is to build the SOAP response which includes the response and MethodName variables and store it in the strTmp variable.

```

strTmp = |<?xml version="1.0" encoding="UTF-8" standalone="no"?>| & _
|<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">| & _
|<SOAP-ENV:Body>| & _
|m:| & methodName & "Response" & | xmlns:m="uri:| & nameSpace & |"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">| &
response & _
|</m:| & methodName & |Response>| & _
|</SOAP-ENV:Body>| & _
|</SOAP-ENV:Envelope>|

```

By default, Domino translates Web agent content to HTML and, because the response is going to be populated in XML, data is necessary to specify that the content type of the agent is XML. Use the Print statement for that:

```
Print "Content-Type: text/xml"
```

7. To send the SOAP response to the requester, use the following code:

```
Print strTmp
```

To terminate execution of the current block statement, use:

```
Exit Sub
```

The last step is to include the error-handling routine that begins at the label Errhandle defined previously:

```
Errhandle:  
Exit Sub
```

In our case, we include `Exit Sub` to terminate the execution and do nothing but it is possible to include other Lotus Script code to handle the errors.

A.1.1.4 Creating a Lotus Script Library

As we defined in the SOAP incoming message skeleton, the method to be executed has to be inside a Lotus Script Library. To create a Lotus Script Library, open the database in Domino Designer 6 and select **Script Libraries**. Then click **New LotusScript Library**. This opens a new Script Library as shown in the figure.

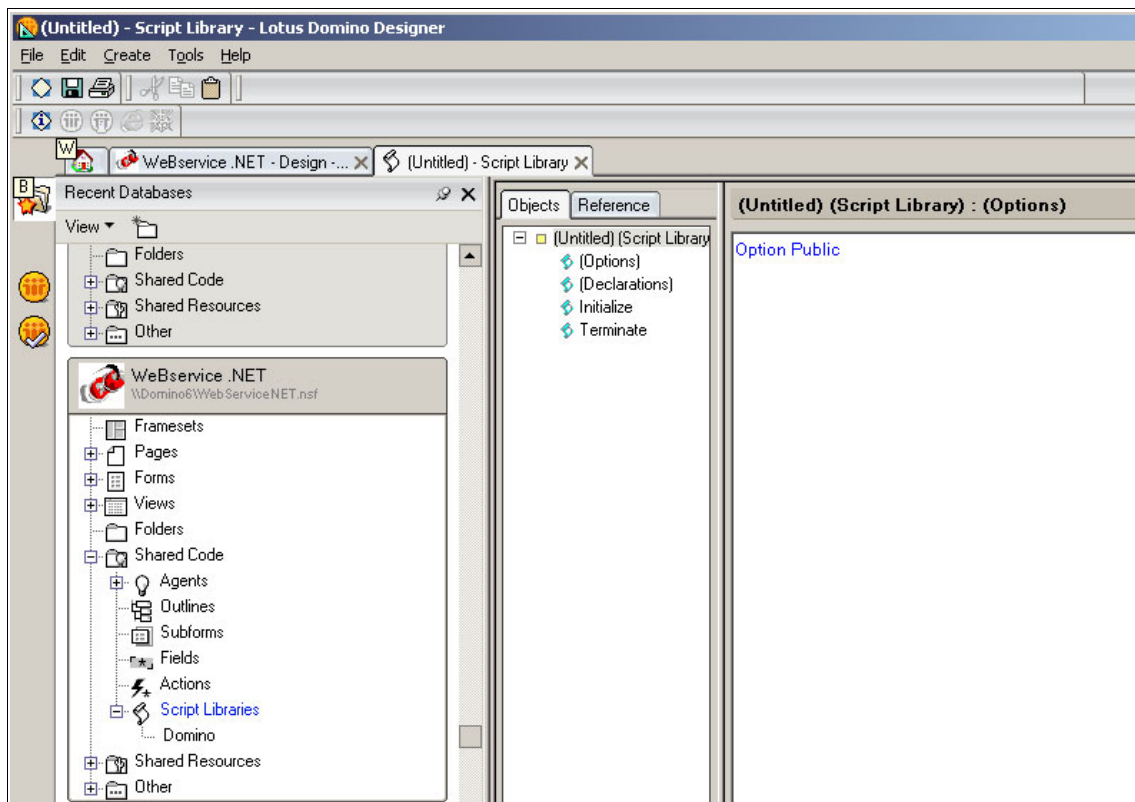


Figure A-6 New Script library

1. Before sending the details regarding a new ITSO residency to the client, it is necessary to parse the content of the SOAP incoming Message using the NotesDOMParser class. So first of all, create a NotesDOMParser object by declaring the variable domParser in the Script Library (Declaration) section.

This class is new in Lotus Domino 6 and is used to process input XML into a standard DOM (Document Object Model) tree structure.

```
Dim domParser As NotesDOMParser
```

Note: For more information about the new NotesDOM classes in Domino 6, refer to the Lotus Domino Designer 6 Help. Also, more information about DOM Document Object Model Core is found at:

<http://www.w3.org/TR/DOM-Level-3-Core/core.html#ID-1590626202>

2. Create a new function with the ResCodeSearch name. This function will be the method specified in the SOAP request.

```
Function ResCodeSearch(msg As String) As String
```

3. The next step is to create a NotesSession object by declaring the variable session and setting it as New to create a new instance for that object. Initialize the object variables.

```
Dim session As New NotesSession
Dim rootElement As NotesDOMDocumentNode
Dim nodeList As NotesDOMNodeList
Dim node As NotesDOMNode
```

4. Initialize the object variable domParser using the CreateDOMParser method of the NotesSession class and use the process method to generate the DOM tree. Set the object variable rootElement using the *Document* property of the NotesDOMParser class to access the document node. Set the object variable nodeList using the GetElementsByTagName method of the NotesDOMDocumentNode class specifying * as the tag name. This will return a NotesDOMNodeList of all the NotesDOMElementNode objects with this given tag name. The list returned is arranged in the order in which they are encountered. The last step is to locate the value of the Residency Code inside the <code> tag of the in the node list and set it to the variable Code.

```
Set domParser = session.createDOMParser(msg)
domParser.process
Set rootElement = domParser.Document
Set nodeList = rootElement.GetElementsByTagName("*")
'locate Residency Code
For i=1 To nodeList.NumberOfEntries
Set node = nodeList.GetItem(i)
If (node.NodeName="code") Then
Code = node.FirstChild.NodeValue
Exit For
End If
Next
```

5. When the Code variable is retrieved, it is necessary to locate this code inside the database. For that, set the variable db with the property CurrentDatabase

of the *NotesSession* class. Then, set the object variable view using the *GetView* method, giving it the name of a view. After that, initialize the object variable doc using the *GetDocumentByKey* method, giving it the Residency Code located before and the true parameter because we want to find an exact match.

```
Dim db As NotesDatabase
Dim view As NotesView
Dim doc As NotesDocument
Set db = session.CurrentDatabase
Set view = db.getView("By Code")
'locate Residency Code in database
Set doc = view.GetDocumentByKey(Code,True)
```

6. The last step is to create a response containing XML data for the SOAP client, and return the result to the "ResidencyWS" agent which calls our function ResCodeSearch.

```
tmp =|<Code xsi:type="xsd:string">| & doc.Code(0) &|</Code>|&_
|<Name xsi:type="xsd:string">| & doc.Name(0) &|</Name>|&_
|<StartDate xsi:type="xsd:string">| & doc.SDate(0) &|</StartDate>|&_
|<EndDate xsi:type="xsd:string">| & doc.EDate(0) &|</EndDate>|&_
|<Contact xsi:type="xsd:string">| & doc.Contact(0) &|</Contact>|&_
|<email xsi:type="xsd:string">| & doc.email(0) &|</email>|&_
|<Location xsi:type="xsd:string">| & doc.Location(0) &|</Location>|
```

```
ResCodeSearch = tmp
```

7. Save the new Lotus Script Library as Domino.

A.1.1.5 Creating a WSDL page to describe the Domino Web Services

WSDL allows a service provider to specify the following characteristics of a Web Service:

- ▶ Name of the Web Service and addressing information.
- ▶ Protocol and encoding style to be used when accessing the public operation of the Web Service.
- ▶ Type information: operations, parameters, and data types comprising the interface of the Web Service, plus a name for this interface.

The anatomy of a WSDL document is as follows:

- ▶ **Types:** A container for data type definitions using some type system, such as XML schema.
- ▶ **Message:** An abstract, typed definition of the data being communicated. A message can have one or more typed parts.

- ▶ **Port type:** An abstract set of one or more operations supported by one or more ports. Each operation defines an input and an output message as well as an optional fault message.
- ▶ **Operation:** An abstract description of an action supported by the service.
- ▶ **Binding:** A concrete protocol and data format specification for a particular port type. The binding information contains the protocol name, the invocation style, a service ID, and the encoding for each operation.
- ▶ **Port:** A single endpoint, which is defined as an aggregation of a binding and a network address.
- ▶ **Service:** A collection of related ports.

WSDL specification uses XML syntax; therefore, there is an XML schema for it.

Note: For more information about WSDL, refer to *WebSphere Version 5 Web Services Handbook*, SG24-6891.

Because the Domino Web Service created before is going to be consumed by a .NET client, a WSDL file is required. This WSDL file will be included in a Lotus Domino Page. Create a new page in Domino; follow the steps below.

1. Open the database in Lotus Domino Designer and select **Pages** on the left pane and click **New Page**; an untitled, blank page is displayed as shown in the following figure.

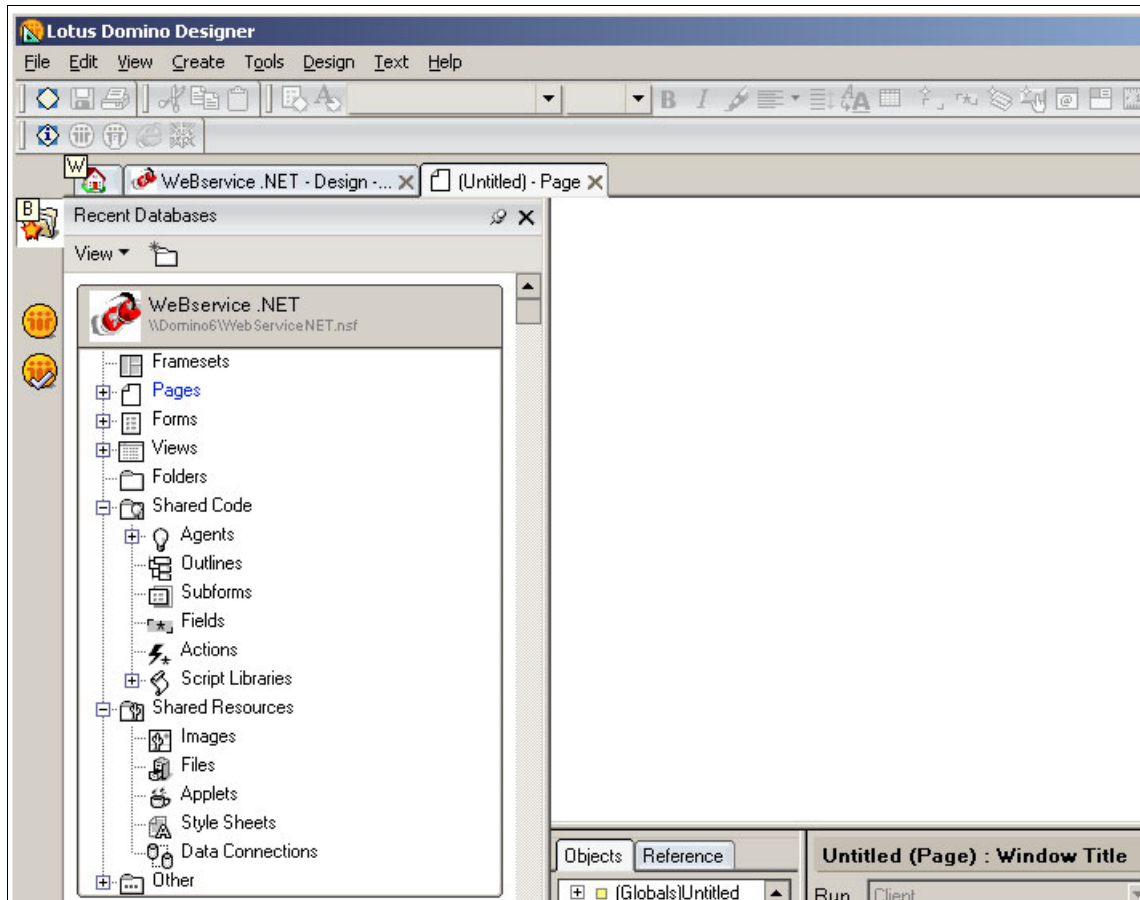


Figure A-7 Create a new Page

2. Give GetResidencyDetailSWSDL as the Page Name and select **Other** in the Web Access - Content type section of the Page Info tab. Enter text/xml in the text box as depicted in the following figure.

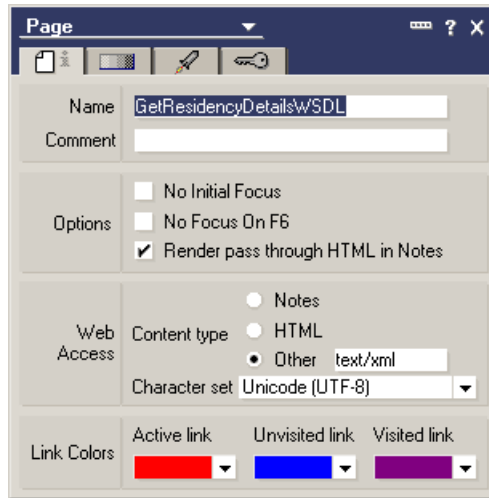


Figure A-8 Set the page content to XML

Leave the Character set to Uni code(UTF-8) and close the Properties box.

3. Create the WSDL file from scratch taking the following things into account:
 - a. The root element of the WSDL file is the `<definitions>` element, which defines the namespaces used in the file. The targetNamespace should be the name of the Domino Lotus Script Library.
 - b. The WSDL will contain two `<message>` type of interaction between the service requestor and the service provider: the first type for the request and the other for the response. The message request will contain the Residency Code as the string type and the message response will contain the data retrieved also as the string type.
 - c. The `<portType>` will be a request-response operation type which means that there will be an input message (defined in the message part of the WSDL file as the message request) followed by an output message (defined in the message part of the WSDL file as the message Response).
 - d. In the `<binding>` part, you will have to specify the following:
 - A name for the binding.
 - The connection should be SOAP HTTP; the style must be RPC.
 - The operation name must be the method name to be executed. In our case, this will be the function (*ResCodeSearch*) inside the Domino Script library.
 - A SOAP Action.

- A reference for the SOAP operation defining an input message and an output message, both to be SOAP encoded because RPC/Literal Web Service calls are not supported by Microsoft .NET. Notice that both input and output messages must contain the name of the Lotus Script Library as the namespace (*namespace="uri:Domino"*).
- e. In the *<service>* part, you will define the port that use the SOAP binding specified before and the URL for the Web Service.

Example 11-1 Web Service WSDL

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name="ResCodeSearch" targetNamespace="Domino"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="Domino"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <message name="ResCodeSearchRequest">
    <part name="code" type="s:string"/>
  </message>
  <message name="ResCodeSearchResponse">
    <part name="Code" type="s:string"/>
    <part name="Name" type="s:string"/>
    <part name="StartDate" type="s:string"/>
    <part name="EndDate" type="s:string"/>
    <part name="Contact" type="s:string"/>
    <part name="email" type="s:string"/>
    <part name="Location" type="s:string"/>
  </message>
  <portType name="ResCodeSearchPortType">
    <operation name="ResCodeSearch">
      <input message="tns:ResCodeSearchRequest" />
      <output message="tns:ResCodeSearchResponse" />
    </operation>
  </portType>
  <binding name="ResCodeSearchBinding" type="tns:ResCodeSearchPortType">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"
/>
    <operation name="ResCodeSearch">
      <soap:operation
soapAction="capeconnect:ResCodeSearch:ResCodeSearchPortType#ResCodeSearch" />
      <input>
        <soap:body use="encoded" namespace="uri:Domino"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </input>
```

```

        <output>
          <soap:body use="encoded" namespace="uri:Domino"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </output>
      </operation>
    </binding>
    <service name="FindResCode">
      <port name="ResCodeSearchPort" binding="tns:ResCodeSearchBinding">
        <soap:address location="http://127.0.0.1/WebServiceNET.nsf/ResidencyWS"
/>
      </port>
    </service>
  </definitions>

```

4. Save the GetResidencyDetailsWSDL page. It is possible to access this WSDL file from another development platform using the following URL:

<http://<someserver>/WebServiceNet.nsf/GetResidencyDetailsWSDL?OpenPage>

A.1.1.6 Creating a .NET client

For consuming the Domino Web Service, we are going to create a simple console based Microsoft .NET application using C# as the programming language and Microsoft .NET Framework Software Development Kit V1.1.

Before creating this, it is necessary to understand the way by which the clients communicate with Web Services. Web Services use HTTP and SOAP to make the business data available on the Web. A Web Service Consumer will use SOAP over HTTP to execute Remote Procedure Calls (RPC) to Web Services methods components.

For security reasons, client applications will not execute the Web Services methods on the location where Web Services reside, but will use a 'proxy object' to act on behalf of the original Web Service. The proxy object at the client side communicates with the Web Service using HTTP/SOAP protocols. The WSDL file which describes the Web Service is used to generate the proxy object. The scenario is illustrated in the following figure.

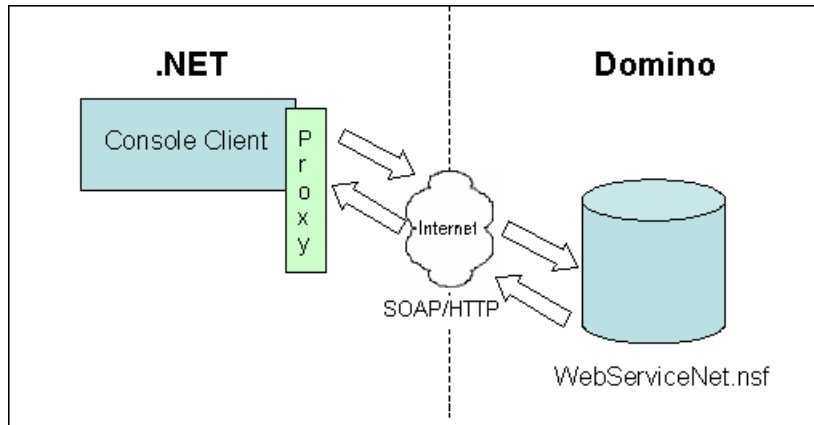


Figure A-9 Console .NET application as Web Services consumer

Therefore, the necessary steps for building a .NET client will be:

1. Create a proxy object to allow communication between the console client and the Domino Web Service.
2. Create a simple C# console based application which will invoke methods of the proxy class.

Creating a proxy object to act on behalf the Web Service

For creating a proxy object, use the Microsoft Web Services Description Language Utility (wsdl.exe) included in Microsoft .NET Framework Software Development Kit V1.1. This utility will generate code for Web Service clients from WSDL files.

Note: A full description of the utility is out of the scope of this document but more information is found at the following URL:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfwebservicedescriptionlanguagetoolwsdl.exe.asp>

Help information is also available in a command window: **wsdl /?**

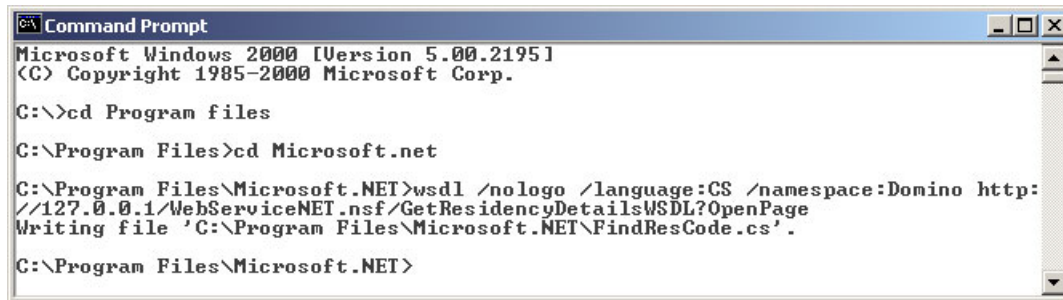
Before using the utility, make sure that the Domino server is running. After that, type this simple command to generate the proxy class:

```
wsdl /nologo /language:CS /namespace:Domino http://<Domino server
name>/WebServiceNET.nsf/GetResidencyDetailsWSDL?OpenPage
```

Where <Domino server name> is the host name for the Domino server or the IP address, http://<Domino_server_name>/WebServiceNET.nsf/GetResidencyDetailsWSDL?OpenPage

ilsWSDL?OpenPage is the URL where the WSDL is located and 'language:CS' is the programming language used to generate the proxy class (C#).

The result of the command will be a file called FindResCode.cs as shown in the following figure.



```
Command Prompt
Microsoft Windows [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>cd Program files
C:\Program Files>cd Microsoft.net
C:\Program Files\Microsoft.NET>wsdl /nologo /language:CS /namespace:Domino http://127.0.0.1/WebServiceNET.nsf/GetResidencyDetailsWSDL?OpenPage
Writing file 'C:\Program Files\Microsoft.NET\FindResCode.cs'.
C:\Program Files\Microsoft.NET>
```

Figure A-10 Command line wsdl utility

Note that the name of the Proxy class is the same name given to the Service part in the WSDL file.

```
<service name="FindResCode">
  <port name="ResCodeSearchPort" binding="tns:ResCodeSearchBinding">
    <soap:address
      location="http://127.0.0.1/WebServiceNET.nsf/ResidencyWS" />
    </port>
  </service>
```

The next step is to compile the file FindResCode.cs with the C# .NET compiler (csc.exe) included in the Microsoft .NET Framework SDK, by issuing the following command from the directory where the proxy class was generated:

```
csc /nologo /out:FindResCode.dll /target:library FindResCode.cs
```

The result of this command will be a Dynamic Link Library called FindResCode.dll (the proxy object).

Creating a simple C# console based application

For creating a simple console-style client application, add the following lines in any text editor, this will generate a new C# source file called NotesConsoleClient.cs.

Example 11-2 .NET client code

```
using System;
using Domino;
using System.Web.Services;
```

```

amespace NotesConsoleClient
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Client
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            FindResCode WSResCodeSearchService=new FindResCode();
            String
rName="",rStartDate="",rEndDate="",rContact="",rEmail="",rLocation="";
            String result=WSResCodeSearchService.ResCodeSearch(args[0], out
rName, out rStartDate, out rEndDate, out rContact, out rEmail, out rLocation);

            System.Console.Out.Write(
                "Name: "+rName+"\r\n"+
                "Start date: "+rStartDate+"\r\n"+
                "End date: "+rEndDate+"\r\n"+
                "Contact: "+rContact+"\r\n"+
                "e-mail: "+rEmail+"\r\n"+
                "Location: "+rLocation+"\r\n"+
                "-----\r\n");
        }
    }
}

```

The purpose of this client is to show the details of a particular ITSO residency located in the Web Services .NET Database Application for a given Residency Code. These details will be: Name, Start date, End Date, Contact, e-mail and Location.

After saving the file, build an executable by compiling this code using the C# library (FindResCode.dll) created before and by issuing the following command:

```
csc /r:FindResCode.dll NotesConsoleClient.cs
```

The result of this command will be an executable called NotesConsoleClient.exe.

A.1.1.7 Test the example

After building the *NotesConsoleClient.exe* executable (using C# library FindResCode.dll), run it from the command line window in the directory where the library is placed and after ensuring that the Domino Server is running.

The NotesConsoleClient executable needs to have a Residency Code as an input; as an example, test the Domino Web Service retrieving the information of the Residency Code SA-W324; the result of running the client is shown in the following figure.

```

C:\ Command Prompt
Volume Serial Number is 6C10-18AA

Directory of C:\Program Files\Microsoft.NET
02/12/03  03:09p                3,584 NotesConsoleClient.exe
               1 File(s)                3,584 bytes
               0 Dir(s)      2,439,149,568 bytes free

C:\Program Files\Microsoft.NET>NotesConsoleClient "SA-W324"
Name: IBM WebSphere and Microsoft .NET Coexistence
Start date: 25/08/2003
End date: 04/10/2003
Contact: Peter Kovari
e-mail: peter.kovari@us.ibm.com
Location: Raleigh NC, USA
-----
C:\Program Files\Microsoft.NET>

```

Figure A-11 Client application results

When the execution of the client had finished, it is possible to check the SOAP incoming message generated by the .NET client message that was consumed and processed by our Domino agent. Open the WebServiceNet.nsf database in a Domino Client and select the **Message view**. A new Document will appear in the view as shown below.



Figure A-12 Message View

Open the document and look at the format of the SOAP incoming message. Included within the SOAP Body is the method signature ResCodeSearch that will be executed on the Domino server. Additionally, the method signature contains the namespace where the method is located (Domino is our LotusScriptLibrary). Notice also that the Residency Code to search for is wrapped within the code argument. An example of the message is shown below with the method name, namespace (Domino Script Library), and method argument highlighted in bold:

```

<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="Domino" xmlns:types="Domino/encodedTypes"

```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<q1:ResCodeSearch xmlns:q1="uri:Domino"><code
xsi:type="xsd:string">SA-W324</code></q1:ResCodeSearch>
</soap:Body></soap:Envelope>

```

Note: Be aware that the format of the incoming SOAP message for Domino is very important and must include all the arguments highlighted in bold.

Because we cannot control the format of the SOAP incoming message form of the .NET client and because the Lotus Script Debugger in Domino is running from the Notes client, when experiencing problems with the format of the SOAP incoming message and Domino, a way to perform problem determination would be:

1. Create a new Lotus Script agent for using from a Lotus Notes Client (use the Lotus Script debugger). This agent will be the same as the ResidencyWS but with the SOAPin variable previously initialized in the way that Domino treats the SOAP incoming message as a string. For example, for the previous SOAP message, the SOAPin variable will be:

```

SOAPin=|<?xml version="1.0" encoding="utf-8"?>| &_
|<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="Domino" xmlns:types="Domino/encodedTypes"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">|&_
|<soap:Body
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">|&_
|<q1:ResCodeSearch xmlns:q1="uri:Domino">|&_
|<code xsi:type="xsd:string">|&"SA-W324"&|</code>|&_
|</q1:ResCodeSearch>|&_
|</soap:Body>|&_
|</soap:Envelope>|

```

2. Save the agent and launch the Lotus Script Debugger by selecting **Menu -> Tools -> Debug Lotus Script**.
3. Run your agent from the Actions Menu and cover all the steps to isolate the problem.

Comment: On the additional material enclosed with this redbook there is a batch file called build.bat. The purpose of this file is to generate automatically the proxy DLL, compile the NotesConsoleClient code with that DLL, and test the application with a Residency Code example. It is possible to use this file for testing the redbook example by placing it in a directory with the NotesConsoleClient.cs. Edit the file and change the paths specified for the DOTNETSDK and DOTNETFW variables using the existing ones on your machine.

A.1.2 .NET service provider, Domino service consumer

Lotus Domino can also act as a service requester and consume a .NET service, which means that you can invoke this service programmatically. Programming languages inside Domino (Lotus Script and Java) and external tools such as the Microsoft SOAP Toolkit can be used for accessing to a external .NET Web Service. The objective of this section is briefly to explain how Domino can use this tool for accessing .NET.

There are three ways to call a Web Service: HTTP GET, HTTP POST and SOAP.

Note: In Section A.2, "Using the COM interface" on page 538 we created a .NET service that can access Domino databases through COM. Running the example in a Web browser will show that the NET Framework SDK will render all the information needed for accessing a .NET Web Service programmatically.

1. HTTP Get

- Create a Lotus Script Agent for accessing a Web Service using the Microsoft XML parser included in the Internet Explorer 5.0.1 or later by setting the source using CreateObject ("Microsoft.XMLDOM"), using the load method to access the URL for the .NET Web Service, selecting the document with the documentElement method and then accessing the fields of the document with the selectSingleNode method.

- Create a Java Agent using the following classes:

- URL Class to access the .NET Web Service URL (for example: `http://localhost/webservices/sample.asmx/GetResidencyDetails?Code=SA-W324`) and the `URLConnection` `openConnection` method that returns a connection to the remote object referred to by the URL:

```
URL url = new URL
(http://localhost/webservices/sample.asmx/GetResidencyDetails?Code=SA
-W324);
URLConnection connect = url.openConnection();
```

- `BufferedReader` and `InputStreamReader` classes for reading the response from the Web Service.
2. HTTP Post

Create a Domino Form placing HTML in it by using the `Form method=postaction` tag. HTTP Post will post data to the Web Service.
 3. SOAP
 - Create a Java Agent that send a SOAP request to the Web Service and read the response with `BufferedReader` and `InputStreamReader` classes.
 - Create a Lotus Script Agent for accessing the Web Services using the Microsoft SOAP Toolkit. Download it from <http://msdn.microsoft.com>. Set the SOAP Client with `CreateObject ("MSSOAP.SoapClient")` and then initialize it with the `msoapinit` method using the WSDL file.

A.2 Using the COM interface

COM (Component Object Model) is an open software component specification developed by Microsoft. It defines a specification for developing reusable binary components across multiple software languages and platforms. COM components can be written and called by any language that can call functions through pointers, including C, C++, Delphi, Basic etc.

The COM specification provides:

- ▶ Rules for component-to-component interaction.
- ▶ A mechanism for publishing available functions to other components.
- ▶ Automatic use tracking to allow components to unload themselves when they are no longer needed.
- ▶ Efficient memory usage.
- ▶ Transparent versioning.

Since Domino Release 5.0.2b, the back-end Domino objects have a COM interface with the following benefits:

- ▶ COM requires the presence of Domino or Notes; the software can be Domino server, Domino Designer, or Notes client.
- ▶ COM provides both early-binding (custom) and late-binding (dispatch) interfaces. Early binding makes the Domino classes available as typed variables with compile-time checking. Late binding can be used where the language (for example, VBScript) precludes early binding.

- ▶ The COM interface is the same as the Lotus Script interface, with some exceptions.
- ▶ Domino can act as a COM server or a COM Client.

Note: For information on COM properties and methods and general exceptions to Lotus Script specifications, refer to the Lotus Domino Designer 6 Help database. For information about the Domino Object Model, refer to *Domino Designer 6: A Developer's Handbook*, SG24-6854.

Microsoft .NET can access Domino Objects through COM. To accomplish this, .NET client or .NET Web Services can call Domino objects via a special wrapper. This wrapper, known as Runtime-Callable Wrapper (RCW), is a piece of software that can accept commands from a component, modify them and forward them to another component. Microsoft .NET Common Language Runtime (CLR) uses the RCW to operate with the unmanaged code by making COM calls to the Domino objects as depicted in the following figure.

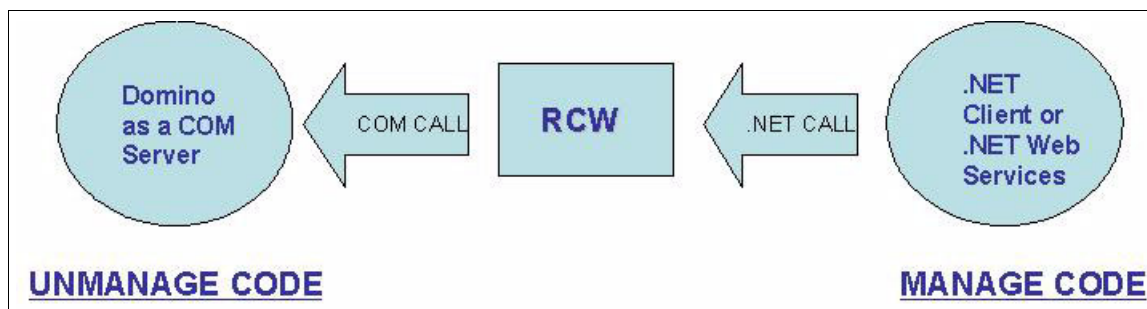


Figure A-13 .NET call through RCW

In the same way, Lotus Domino can act as COM client and provide .NET objects via a special wrapper known as a COM Callable Wrapper (CCW). The Domino COM Client uses the CCW to operate with Microsoft .NET Common Language Runtime (CLR) by making .NET calls to .NET servers, as depicted in the following figure.

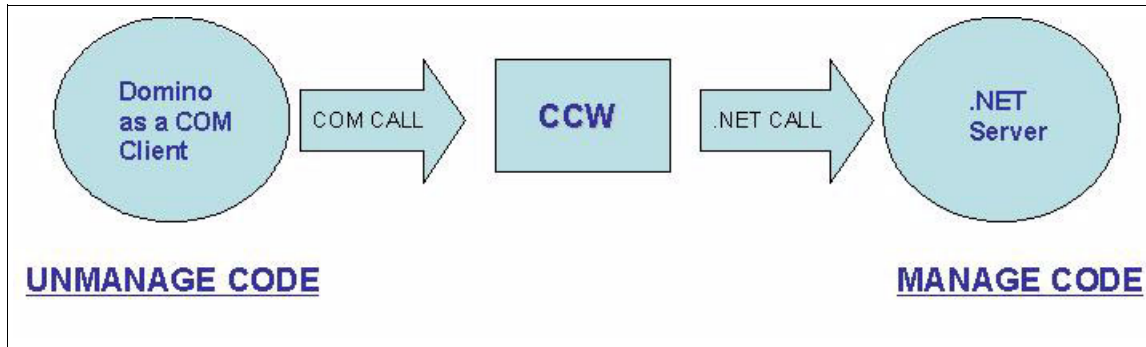


Figure A-14 Domino call through CCW

The purpose of this section is to explain how .NET clients can access Domino databases through COM. The example shows how to create a .NET Web Service that returns the details of a particular ITSO Residency located in a Notes database.

For more information on the sample database, refer to A.2.1, "Domino as a COM server, .NET as a client" on page 540, and A.2.1.2, "Creating a Domino sample database" on page 554.

A.2.1 Domino as a COM server, .NET as a client

For running this example, we used the following product versions:

- ▶ .NET Framework Software Development kit V1.1
- ▶ Microsoft Windows 2000 Server with Service Pack 4
- ▶ Lotus Domino Server V6.0.2 CF2
- ▶ Microsoft Internet Information Services V5.0
- ▶ Microsoft IE V5.5 or newer
- ▶ Lotus Domino Administration Client V6.0.2
- ▶ Lotus Domino Designer Client V6.0.2

For more details on how to install the software products, refer to the installation manuals. Internet Information Services is included with Microsoft Windows 2000 and it is possible to install it under the Add/Remove Windows Components option inside Add/Remove Programs within the Control Panel.

Lotus Domino Administrator client and Lotus Domino Designer were installed in another machine in order to administrate the Domino Server and to design the sample application.

Before starting, make sure that you have TCP/IP network configured (it is recommended that you have a fixed IP address), that it is possible to resolve the machine host name (via Host file or DNS) and that you also have Domino Server configured. For our example, the following Domino nomenclature was selected within the configuration process, but of course it is possible to use other settings.

Table A-3 Domino nomenclature

Concepts	Selected Name
Domino Domain Name	TEST
Organization Name	TEST
Server Name	Domin6/TEST
Server Title	Test Server
Notes Network Name	TCPIP Network
Notes Administrator Name	Notes Admin/TEST

Note: For more details on configuring a Domino Server, refer to the Lotus Domino Administrator 6 Help database.

Once you have installed and configured the products to begin with the example, follow these steps:

1. Set up Domino to work with IIS Server.
2. Make the Domino objects accessible to .NET and IIS.
3. Create a Domino sample database.
4. Create a .NET Web Service to access the Domino database.
5. Test the example.

A.2.1.1 Setting up Domino to work with the Internet Information Services server

For using a Microsoft IIS Server as a front-end machine with Domino, it is necessary to install the WebSphere Application Server 4.0.3 plugin for IIS on the IIS server. The plugin files are packaged with the Domino 6 server and must be copied from the Domino Server to the IIS server. After copying the plugin files, configure the plugin. The last step is configuring the Domino server to work with the plugin IIS. Note that is not necessary to install any other WebSphere components on the IIS machine.

Installing the WebSphere plugin on an IIS Server

1. First of all, create the following directory structure on the IIS machine (it is possible to use another drive):
 - ▶ C:\WebSphere\AppServer\bin
 - ▶ C:\WebSphere\AppServer\config
 - ▶ C:\WebSphere\AppServer\etc
 - ▶ C:\WebSphere\AppServer\logs
2. Then, copy the following files located in the Domino data directory to the IIS server:
 - <Domino data directory>/domino/plug-ins/plugin-cfg.xml to c:\WebSphere\AppServer\config.
 - <Domino data directory>/domino/plug-ins/w32/iisWASPlugin_http.dll to c:\WebSphere\AppServer\bin.
 - <Domino data directory>/domino/plug-ins/w32/plugin-in_common.dll to c:\WebSphere\AppServer\bin.

The directory structure is shown in the following figure.

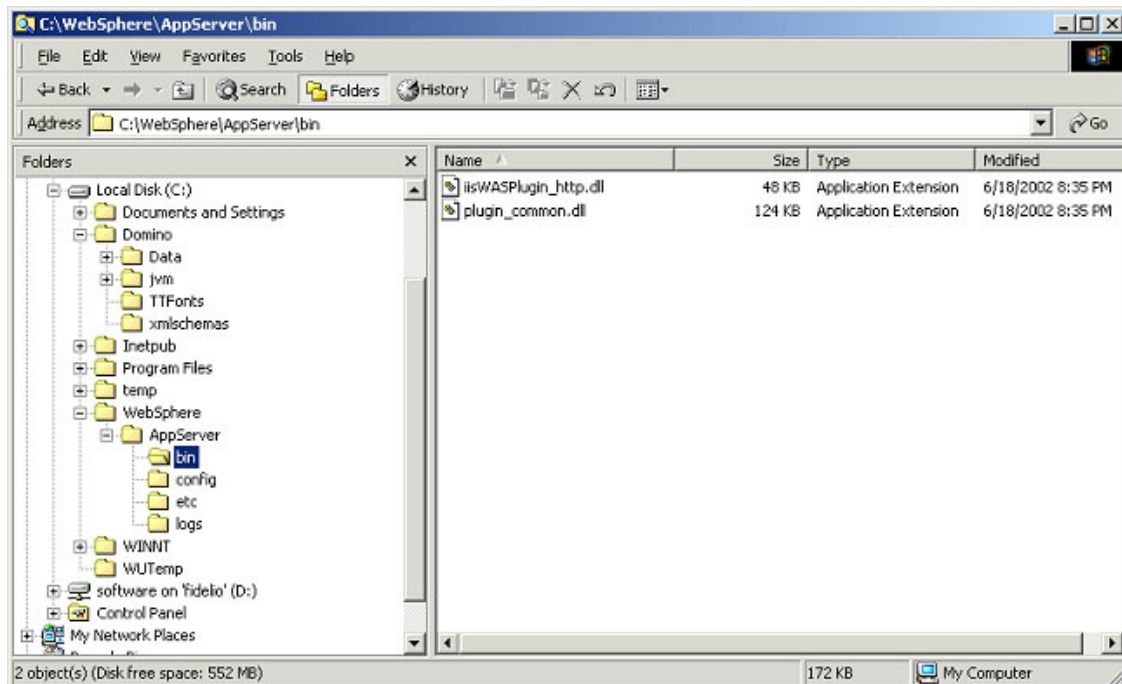


Figure A-15 Directory structure

3. Start the Internet Service Manager application by selecting **Start -> Settings -> Control Panel -> Administrative Tools -> Internet Service Manager**. Expand the Local Machine objects on the left pane to see all the services configured on it, as shown in the following figure.

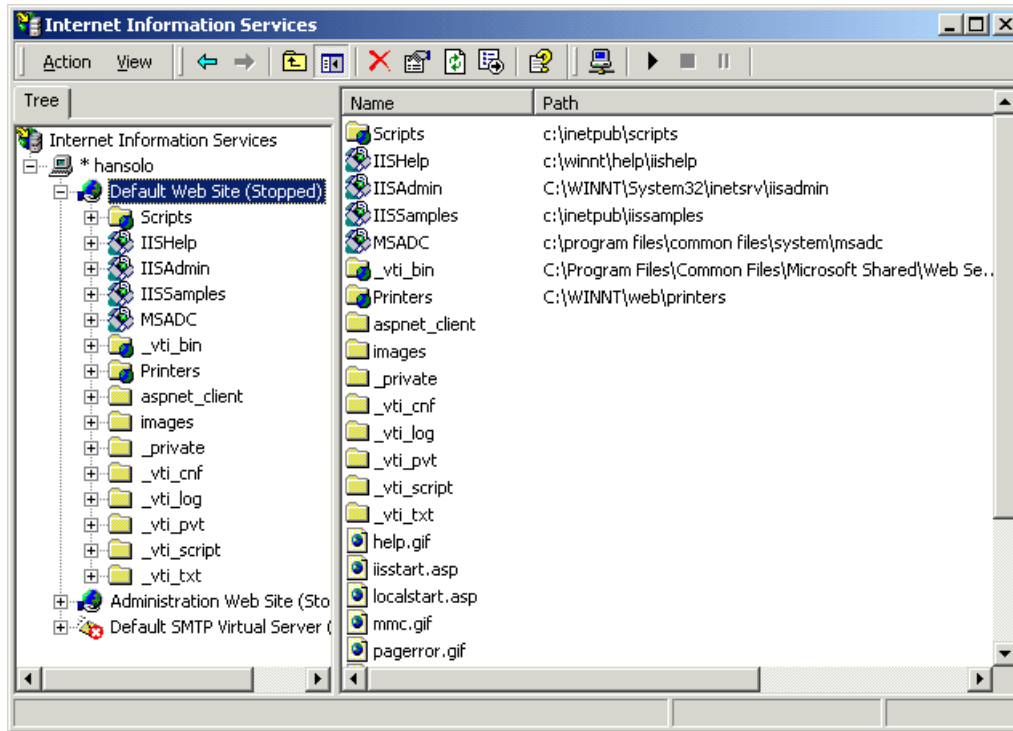


Figure A-16 IIS Services manager

4. Create a New Virtual Directory under the Default Web Site. IIS uses the virtual directories to access directories on other machines or directories outside a service's home directory. In this case, IIS uses the Virtual Directory for access to the WebSphere plugin. Right-click the **Default Web Site** and select **New -> Virtual Directory**. When the new Virtual Directory Creation Wizard is displayed, click **Next**.
5. Enter sePlugins in the alias field as illustrated below (always use this name) and click **Next**.

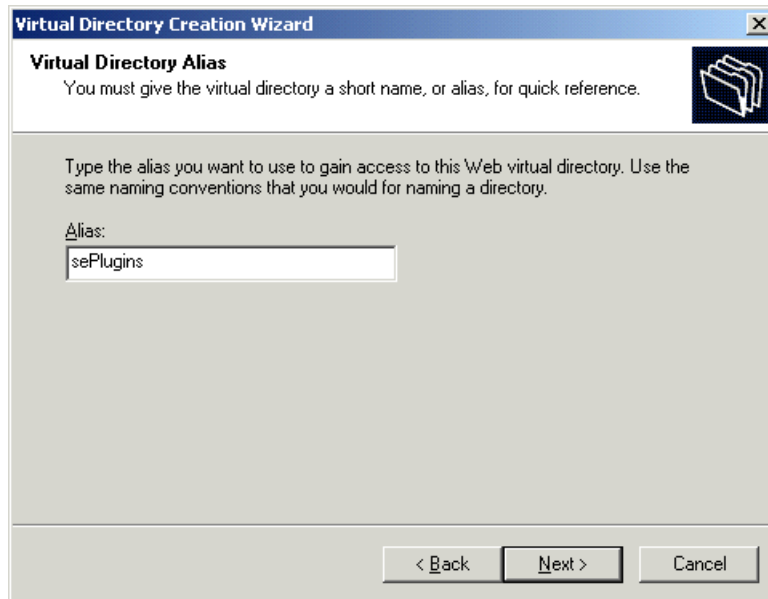


Figure A-17 Virtual Directory Alias

6. In the Directory field, browse to the WebSphere bin directory (C:\WebSphere\AppServer\bin). Click **Next**.
7. Select **Run Scripts** and **Execute** for the access permission as depicted in the following figure.

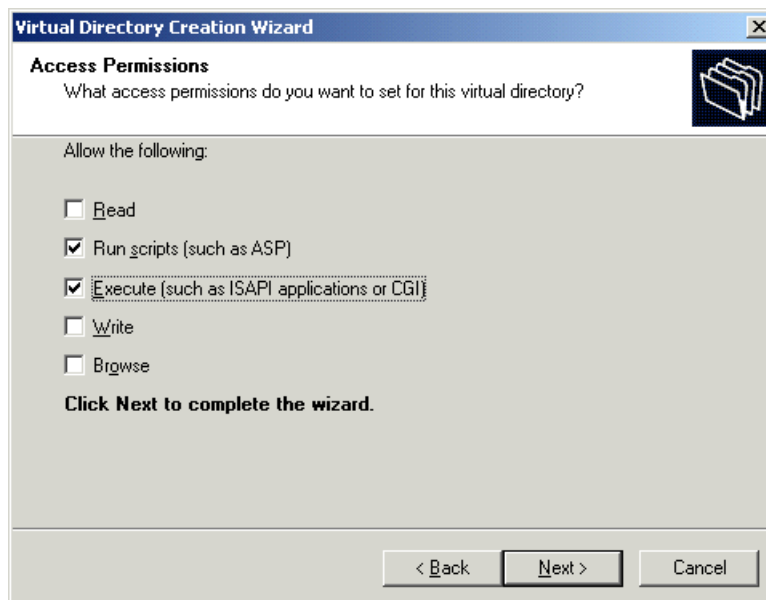


Figure A-18 Access Permissions

8. Click **Next** and then click **Finish**; the new Virtual Directory is shown on the default Web site.

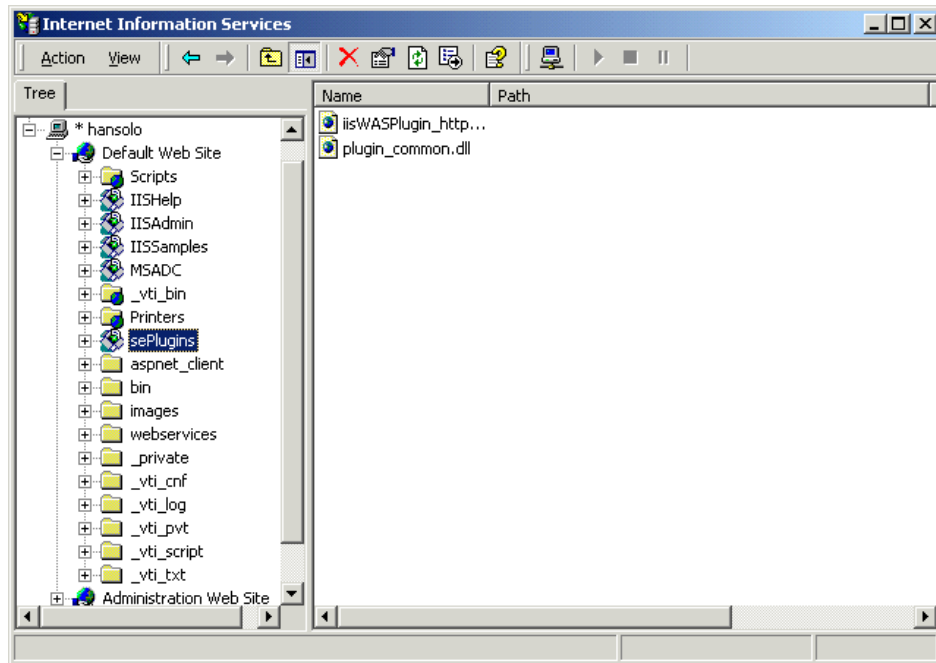


Figure A-19 Virtual Directory in IIS Manager

9. Right-click the machine name and select **Properties**. On the Internet Information Services tab, select **WWW Service** as Master Properties and edit it. The WWW Service properties dialog window will be displayed as shown in the next figure.

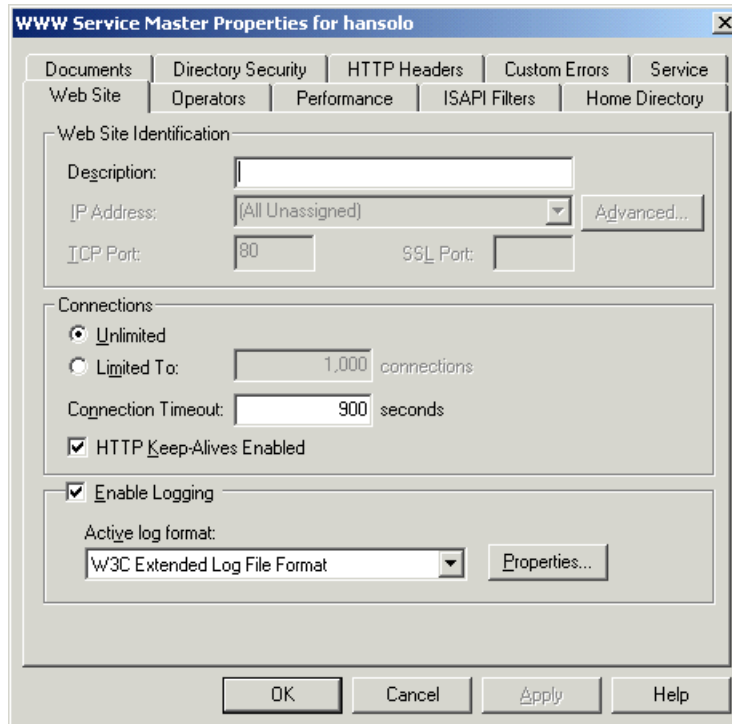


Figure A-20 WWW Service Master Properties - Web Site

10. Select the **ISAPI Filters** tab and click **Add**; the Filter Properties Dialog window will be displayed. In the Filter Name Field type **iisWASPlugin** and for the Executable field, click **Browse** and open the WebSphere bin directory. Select **iisWASPlugin_http.dll** and click **OK**. The parameters are illustrated in the following figure.

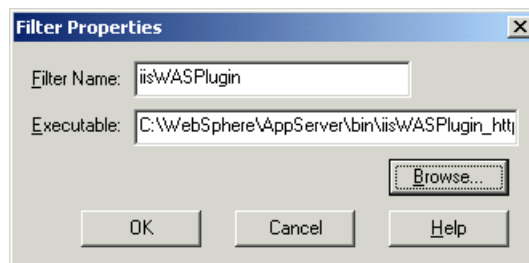


Figure A-21 Filter properties

The new ISAPI filter is displayed as depicted in the following figure.

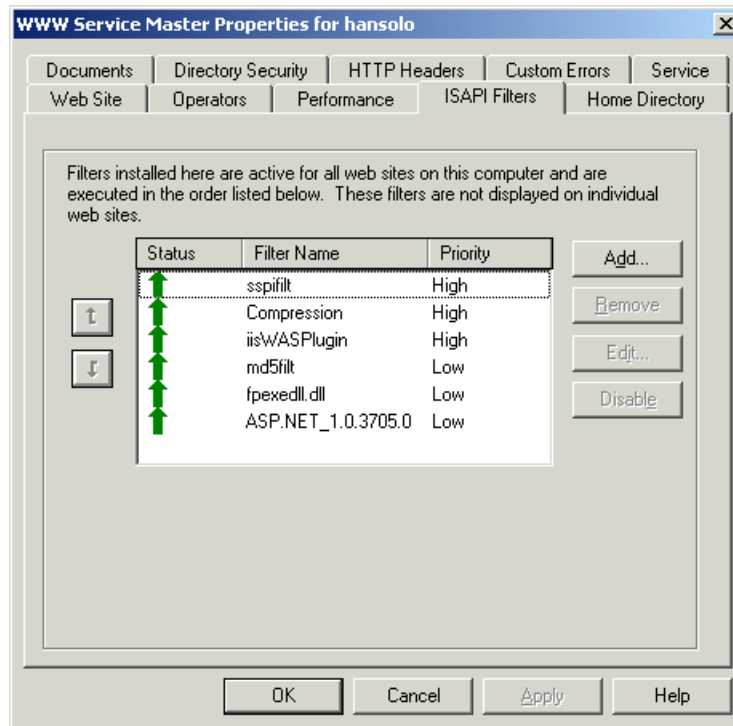


Figure A-22 WWW Service Master Properties - ISAPI Filters

11. Close all open windows.
12. Open the Windows Registry file and go to **HKEY_LOCAL_MACHINE -> Software -> IBM**. Create the key **WebSphere Application Server**. Then select **WebSphere Application Server** and create a new key, **4.0**. Then select **4.0** and create a new string value, **Plugin Config** and set the value for this variable to the location of the plugin-cfg.xml file, for example:
C:\WebSphere\AppServer\config\plugin-cfg.xml.
13. Restart your system.

Configuring the WebSphere plugin

The WebSphere configuration file plugin-cfg.xml controls the operation of the plugin. In order for the plugin to relay requests to the target Domino server, it is necessary to add directives to the file for defining a transport route to the server, and pattern rules for the URL namespaces that identify requests which are to be relayed to Domino. The plugin will only relay requests that match a namespace rule. All other requests will be handled by the front-end Web server. To configure the plugin-cfg.xml file, follow these steps.

1. Open the file, plugin-cfg.xml, with WordPad.
2. Define a group identifying the Domino Server that handles NSF requests forwarded from IIS. Server groups contain servers, and servers contain transport definitions that give the plugin the information it needs to forward requests to Domino.

```
...
<ServerGroup Name="domino_web_servers">
  <Server Name="Domino Server">
    <!-- The transport defines the hostname and port value that the web
server plugin will use to communicate with the application server. -->
    <Transport Hostname="hansolo" Port="81" Protocol="http"/>
  </Server>
</ServerGroup>
```

Note: The Transport host name and Port number are the specified Host name and Port for our example machine. Substitute the values with the Host Name and Port number needed in every case.

3. Define a URI group which specifies the strings within a URL that indicate to IIS and the plugin that the request should be forwarded to Domino.

```
...
<UriGroup Name="domino_host_URIs">
  <Uri Name="/*.nsf*"/>
  <Uri Name="*/domjava*"/>
  <Uri Name="*/icons*"/>
</UriGroup>
```

4. Define a virtual host group. Specify a Host Name and Port for the incoming requests or specify an asterisk (*) for the Host Name, Port, or both.

```
...
<VirtualHostGroup Name="domino_host">
  <VirtualHost Name="*:80"/>
  <VirtualHost Name="*:81"/>
</VirtualHostGroup>
```

5. Define a route to tie the sections together, so any request that matches the patterns listed in the domino_host_URIs group gets forwarded to the server(s) listed in the domino_web_servers group.

```
...
<Route ServerGroup="domino_web_servers" UriGroup="domino_host_URIs"
VirtualHostGroup="domino_host"/>
```

6. Stop and restart the World Wide Web Publishing Service from the Windows Services Control Panel.

Configuring Domino Server to work with Microsoft IIS

1. In the Domino Server, edit the Notes.ini file located in the Domino directory, in our case c:\Domino, and add the following line:

```
HTTPEnableConnectorHeaders=1
```

The setting enables the Domino HTTP task to process the special headers added by the plugin to requests. These headers include information about the front-end server's configuration and user authentication status. As a security measure, the HTTP task ignores these headers if the setting is not enabled. This prevents an attacker from mimicking a plugin.

2. Because the Domino Server is installed in the same machine as the IIS Server, it is necessary to change the default HTTP port for Domino (80) to an alternative number. We used the 81 port (this is why, in the plugin-cfg.xml file inside <VirtualHostGroup Name="domino_host">, both ports are specified). To change this, open the Domino Server Document from the Domino Administrator by selecting **Configuration Tab -> Server -> Current Server Document** and edit the field.
3. Select **Ports -> Internet ports -> Web** and specify the TCP/IP Port number that the Domino HTTP stack should use, as shown in the following figure.

Web (HTTP/HTTPS)	
TCP/IP port number:	81
TCP/IP port status:	Enabled
Enforce server access settings:	No
Authentication options:	
Name & password:	Yes
Anonymous:	Yes
SSL port number:	443
SSL port status:	Disabled
Authentication options:	
Client certificate:	No
Name & password:	Yes
Anonymous:	Yes

Figure A-23 Domino - Web Internet Ports administration

Note: If Domino and IIS are on separate, dedicated machines, Domino can use port 80 on its own system and no change in the Server document is needed.

4. Select the **Internet Protocols -> Domino Web Engine** tab and configure the Generating References section by selecting the appropriate protocol, Host Name, and Port specified during the configuration of the WebSphere plugin, as depicted below.

The screenshot shows the Domino Server document configuration for 'Domino6/TEST'. The 'Internet Protocols' tab is selected, and the 'Domino Web Engine' sub-tab is active. The 'HTTP Sessions' section shows 'Session authentication' set to 'Disabled'. The 'Generating References to this Server' section includes the following settings:

Generating References to this Server	
Does this server use IIS?	
Protocol:	http
Host name:	hansolo
Port number:	81

Figure A-24 Domino Web Engine administration

Note: For Domino 6, the setting "Does this server use IIS?" is not used.

Save the Domino Server Document with all the changes.

5. Make sure you have the HTTP Task running on the Domino Server. If not, add HTTP to the ServerTasks line of the Notes.ini file. This guarantees that every time the server starts, the HTTP Task is going to be loaded.

ServerTasks=Update,Replica,Router,AMgr,AdminP,CalConn,Sched,http

6. Restart the server.

Verifying the configuration

To verify the configuration, do the following:

1. Enter the URL for the Web server in your Web browser.
2. Verify that the IIS server's home page loads as shown below.

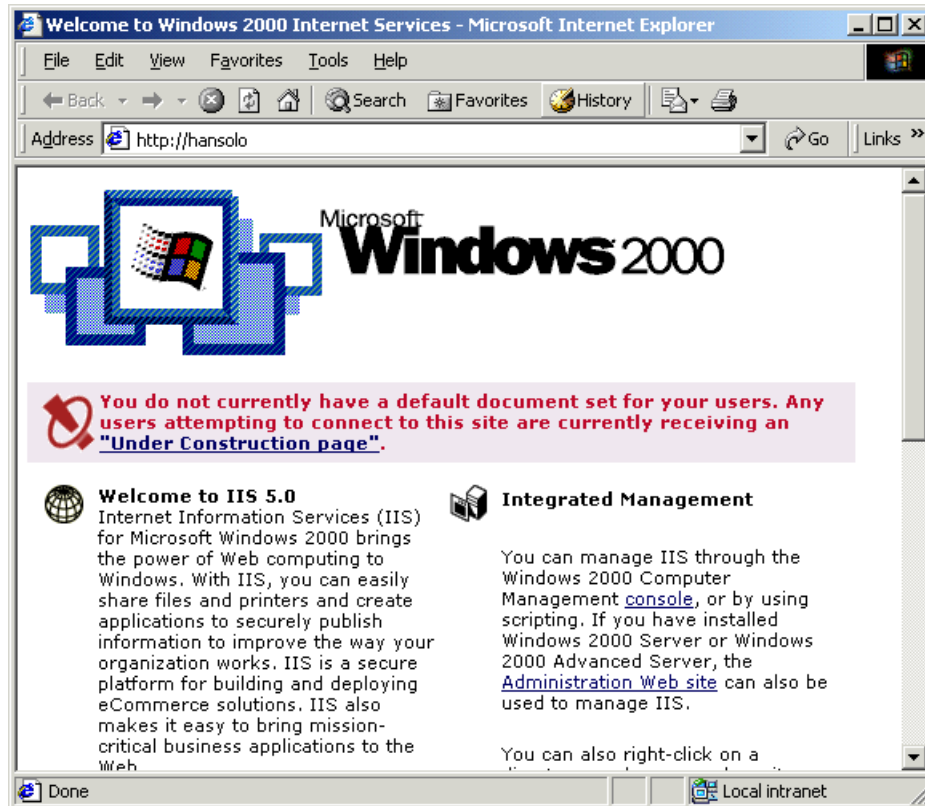


Figure A-25 Verify the plugin configuration

3. Append homepage.nsf to the URL in the address bar. If the Domino home page loads, the configuration is successful, as shown in the following figure.

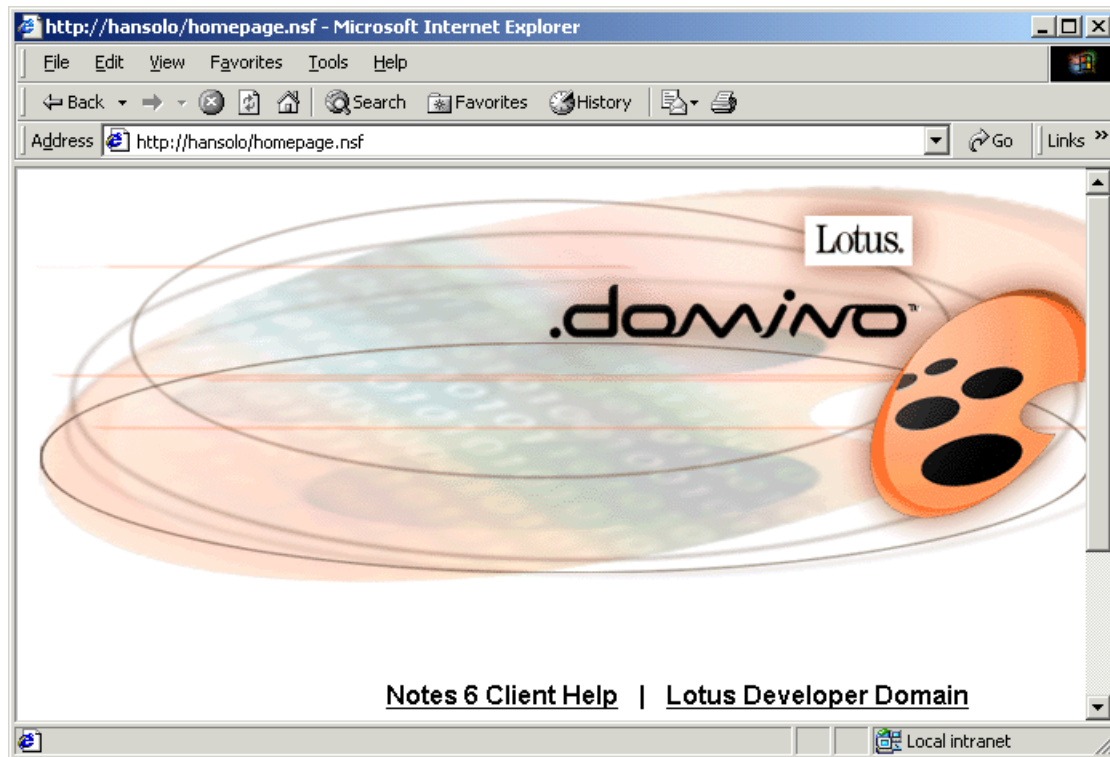


Figure A-26 Domino HTTP Server - homepage.nsf

Making the Domino Objects accessible to .NET and IIS

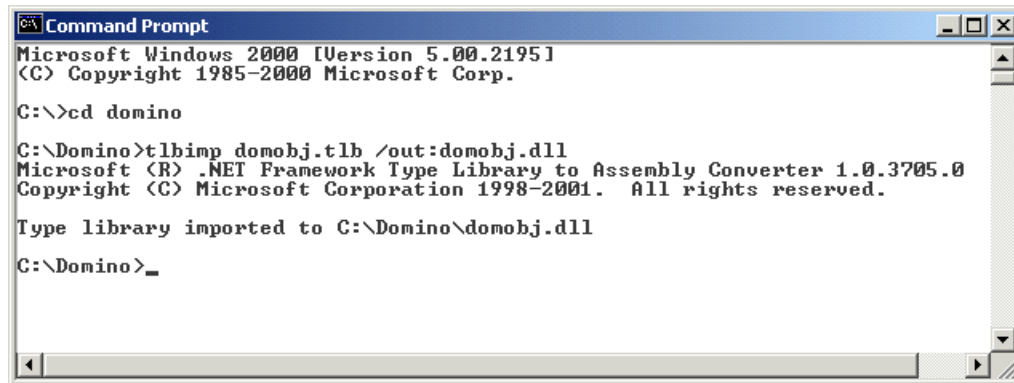
Because the Domino Objects are not included as standard within the .NET Framework Software Developer Kit (SDK), there is a tool included in the software called *Tlbimp* (type library imported) that reads the Domino COM Type Library (domobj.tlb) and creates a matching CLR assembly (domobj.dll) which will be in charge of calling the COM Components.

The Tlbimp tool is a command-line tool which will make the job of RCW easier because it is capable of converting COM metadata to .NET metadata.

Note: More information about Tlbimp tool can be found at the following URL:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfTypeLibraryImporterTlbimpexe.asp>

To create the domobj.dll, from the command prompt, go to the Domino Directory and type `tlbimp domobj.tlb /out:domobj.dll` as shown in the following figure.



```
C:\ Command Prompt
Microsoft Windows 2000 [Version 5.00.2195]
Copyright 1985-2000 Microsoft Corp.

C:\>cd domino

C:\Domino>tlbimp domobj.tlb /out:domobj.dll
Microsoft (R) .NET Framework Type Library to Assembly Converter 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Type library imported to C:\Domino\domobj.dll

C:\Domino>_
```

Figure A-27 Creating the Domino Object

Once you have created the domobj.dll, to make it accessible to Microsoft IIS, copy the DLL inside the bin directory included in c:\inetpub\wwwroot\.

A.2.1.2 Creating a Domino sample database

To show how Microsoft .NET can be integrated with Domino, we have created a Notes sample Web Services .NET database application (WebServiceNet.nsf) using Lotus Domino Designer 6.0.2, to serve as the repository for the upcoming ITSO residencies details.

The application includes the following design elements:

- ▶ *Residencies Form*: this is the form used to create the information details for a new upcoming residency, such as Residency Name, Residency Code, Start Date, End Date, Residency Contact, e-mail and Location.
- ▶ *Residencies View*: this is the view that shows all the residencies to the user.
- ▶ *(By Code) View*: this hidden view is ordered by the Residency Code and is where the .NET and the Domino Web Service will find access to locate the residency.
- ▶ *Message Form*: this form is used for creating a document with the SOAP incoming message every time Domino Web Services is accessed.
- ▶ *Messages View*: this is the view that displays all the SOAP Incoming Messages documents created.
- ▶ *ResidencyWS*: this is the Domino Web Agent for our Domino Web Services and is the one in charge of routing the SOAP request, parsing it, calling the requested method (function), and returning the result as a SOAP response to the requester.

- ▶ *Domino Script Library*: this contains the method (function) for the Domino Web Service.
- ▶ *GetResidencyDetailsWSDL Page*: this contains the WSDL definition of the Domino Services.

To test the integration between both technologies, download the additional material that comes with this redbook, extract the database and then follow the next steps:

1. Copy the database to the Domino server Data Directory and open Lotus Domino Administrator 6.0.2.
2. Log in as a user with administrative privileges, then open the Domino Server from the left server bookmark pane; then click the **Files** tab as shown in the following figure.

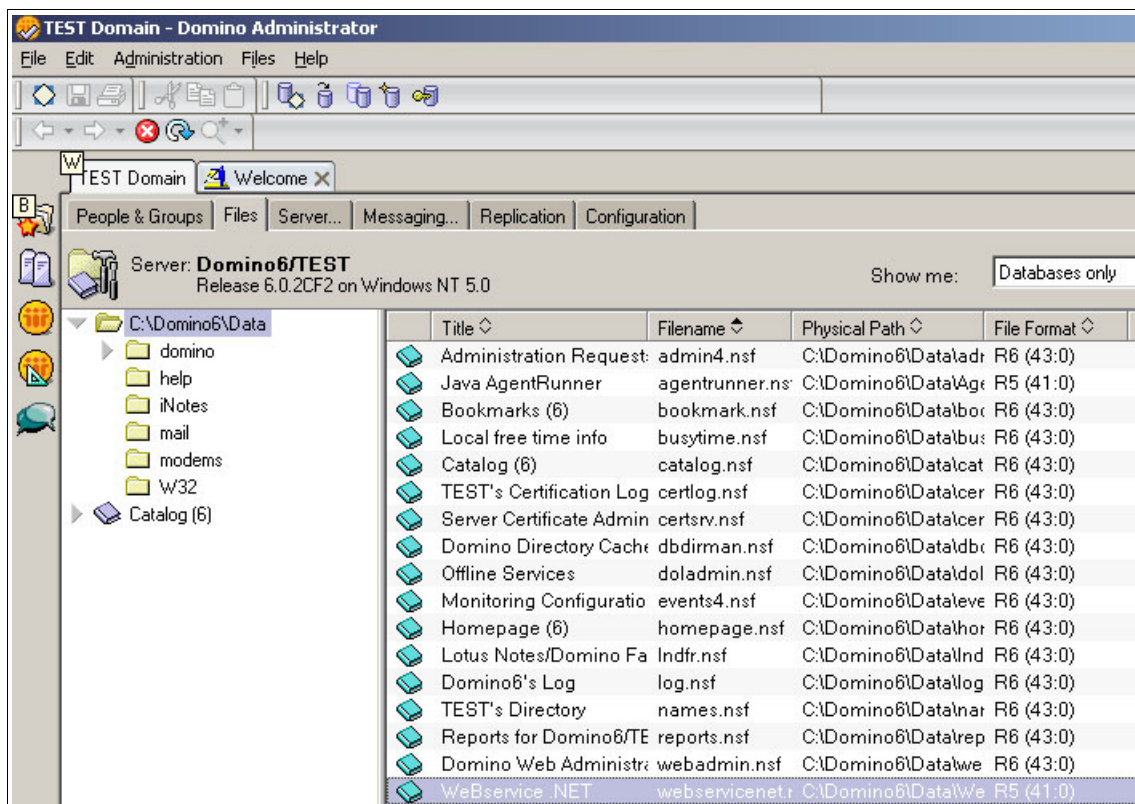


Figure A-28 Domino Administrator

3. Select the **WebService** .NET database application and open the Database toolbar located on the right side of the tools pane; select **Sign**.

4. A new dialog box appears. Leave the default parameters selected and click the **OK** button (see below).

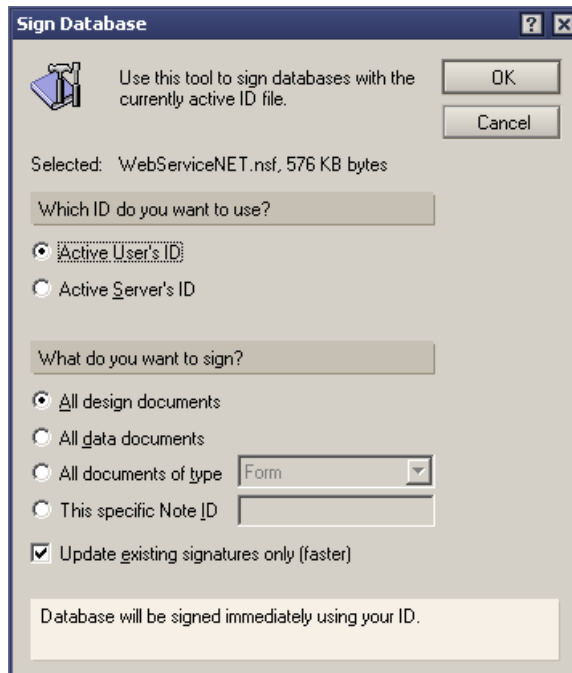


Figure A-29 Domino - Sign Database

5. A new dialog box will appear stating Your Name and Address Book does not contain a cross certificate for this organization (for example: TEST). Click **Yes** to create a new cross-certificate.
6. All the design elements of the database will be signed with the actual ID. When the process is completed, a dialog box shows the number of databases processed and the number of errors that occurred (if any).

A.2.1.3 Creating a .NET Web Service to access the Domino database

To show how you achieve can .NET access to Domino using the COM interface, we have created a Web Service file (Sample.asmx) using a standard text editor (Notepad) and C# as the developing language.

The Web Service returns the details of a particular ITSO residency located in the Web Services .NET Database Application. When the Web Service is called, perform the following operations:

1. Open the Local Web Services .NET Database (WebServiceNet.nsf).
2. Open the hidden view (By Code) which contains all the residencies ordered by Residency Code.
3. For a given Residency Code, locate the corresponding document.
4. Access the fields inside the document.
5. Return the values for this particular residency.

Let's analyze the Web Service code:

1. The first line tells the compiler to run the code in Web Service mode and the name of the C# Class:

```
<%@ WebService Language="C#" class="ResidencyDetailsWebService" %>
```

2. The next lines make references to the classes that the compiler needs to use during the compilation process. These classes are System and System.Web.Services and of course, it will be necessary to use the Domino Objects (domobj.dll) classes for accessing the database.

```
using System;
using System.Web.Services;
using domobj;
```

3. A C# program file can contain one or more namespaces; a namespace can also contain classes, structs, interfaces, etc. The following line makes a reference to a Web Service namespace which contains our ResidencyDetailsWebService Class, which inherits the functionality of the Web Service class:

```
[WebService(Namespace="http://127.0.0.1/")]
public class ResidencyDetailsWebService : WebService
```

4. The Web Service requires the user to enter a Residency Code and will return the details for this particular residency, such as Residency Name, Residency Code, Start Date, End Date, Residency Contact, e-mail and Location. To handle these data values, we used the C# "structs".

```
// The object to return residency details
public struct DocumentResult {
    public string ResCode;
    public string ResName;
    public string ResStartDate;
    public string ResEndDate;
    public string ResContact;
    public string ResLocation;
    public string Resemail;
}
```

5. This Web Service is going to be accessed through HTTP. The data that we are going to access is not sensitive and is available to the public, so we used

the [Web method] keyword. The description tag inside the keyword is used to describe the Web Service functionality.

```
[WebMethod(Description="This method will get the Residency details for the  
specified code.")]  
public DocumentResult GetResidencyDetails(string Code) .
```

6. At this time, we are going to access the Notes database using the Domino classes. First, we need to create a NotesSession object by declaring the variable session and setting it as New to create a new instance for that object. Next, we initialize (explicitly) this COM session; there are two ways to achieve this:

- *Using the Initialize Method:* this method can be used on a computer with a Notes client or Domino server and bases the session on the current user ID. If a password is specified, it must match the user ID's password.
- *InitializeUsingNotesUserName:* this method can be used only on a computer with a Domino server. If a name is specified, the InitializeUsingNotesUserName method looks it up in the local Domino Directory and permits access to the local server depending on the "Server Access" and "COM Restrictions" settings. The password must match the Internet password associated with the name. If no name is specified, access is granted if the server permits Anonymous access.

In our case, we used the second method because we used a computer with a Domino Server, specifying the user name and password.

```
// Connect to Notes and find the details for the residency.  
NotesSession session = new NotesSession();  
session.InitializeUsingNotesUserName("Notes Admin/TEST", "lotusnotes");
```

Note: Use the user name and password corresponding to your server.

7. The next step is to declare the variables db, view, doc, Name, StartDate, EndDate, Contact, email and Location. To get the values of the residencies' form fields, we need to follow the hierarchical path from the top to the lower one. In this example, we go from a NotesSession object to a NotesItem object:

NotesSession -> NotesDatabase -> NotesView -> NotesDocument -> NotesItem

We initialize the variable db with the property GetDatabase, indicating the server name (in our case "" because it is a local machine), database name (in our case WebServiceNET.nsf) and false for the [createonfail] parameter, of the higher level object (NotesSession). We set the object variable view using the GetView method, giving it the name of a view. We initialize the object variable doc using the GetDocumentByKey method, giving it the Residency Code and the true parameter because we want to find an

exact match. The last step is to set the rest of the variables using the `GetFirstItem` method which for a given a name, returns the first item of the specified name belonging to the document.

```
NotesDatabase db = session.GetDatabase("", "WebServiceNET.nsf",false);
NotesView view = db.GetView("By code");
NotesDocument doc = view.GetDocumentByKey(Code,true);
NotesItem Name = doc.GetFirstItem ("Name");
NotesItem StartDate = doc.GetFirstItem ("SDate");
NotesItem EndDate = doc.GetFirstItem ("EDate");
NotesItem Contact = doc.GetFirstItem ("Contact");
NotesItem email = doc.GetFirstItem ("email");
NotesItem Location = doc.GetFirstItem ("Location");
```

Note: For more information about these Lotus Script Classes and Methods, refer to Lotus Domino Designer 6 Help.

8. Create a new `DocumentResult` object and assign the returning values recovered from the database to the initial parameters, defined in the `DocumentResult` struct, using the `NotesItem Text` property class.

```
// Create a new DocumentResult object and return the values.
DocumentResult dr = new DocumentResult();
dr.ResCode = Code;
dr.ResName = Name.Text;
dr.ResStartDate = StartDate.Text;
dr.ResEndDate = EndDate.Text;
dr.ResContact = Contact.Text;
dr.Resemail = email.Text;
dr.ResLocation = Location.Text;
return dr;
```

9. Save the file with the `.asmx` extension.

Now, we are ready to test our Web Service; before proceeding, place the file (sample.asmx) inside the IIS Web directory path, for example `c:\inetpub\wwwroot\webservices`. If you do not have a Web Services directory, create one.

A.2.1.4 Testing the example

Open Microsoft IE Web Browser and type the URL:

`http://<hostmachine>/webservices/sample.asmx`. It will bring up a page that is created automatically by the .NET Framework, as shown in the following figure.

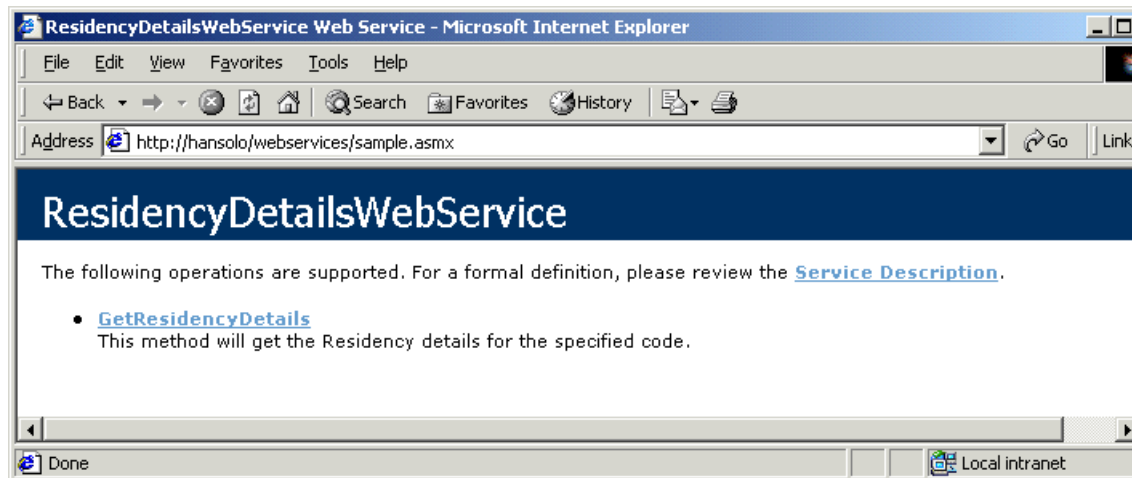


Figure A-30 Web Service test client in IIS 1.

This page has two links: one for the `GetResidencyDetails` method defined in the `ResidencyDetailsWebService` class and a link for the WSDL file which describes the Web Service also created by .NET Framework SDK.

Click the **GetResidencyDetails** link and you will see the next page, also rendered by .NET Framework.

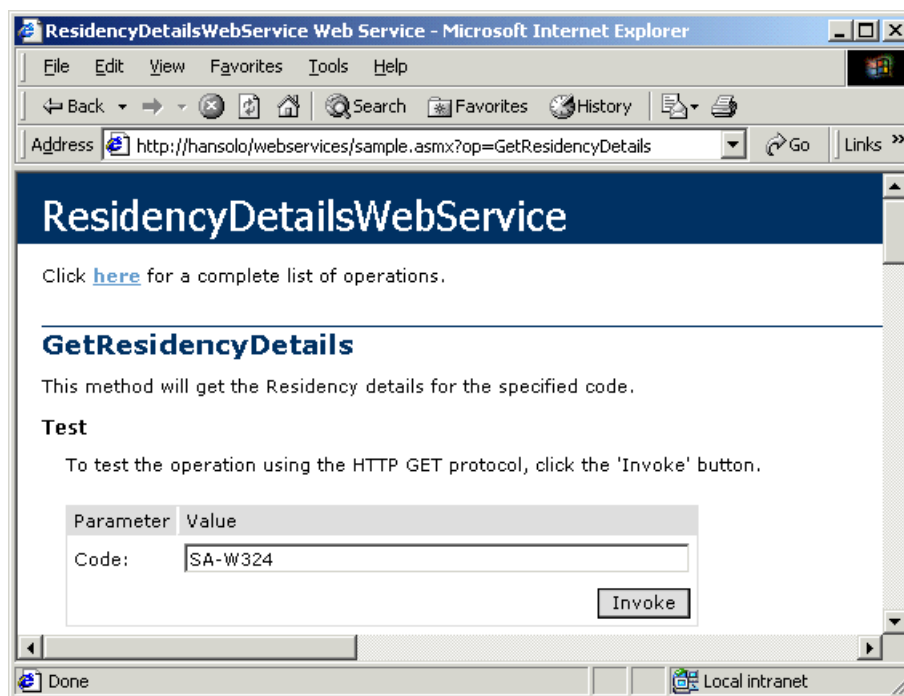


Figure A-31 Web Service test client in IIS 2.

This second page gives you the opportunity to test the Web Service and presents a good deal of useful information because the output (returned in the form of HTTP GET, HTTP POST and SOAP) provides all the hints you need for calling this Web Service programmatically, as shown in the following figures.

► HTTP GET:

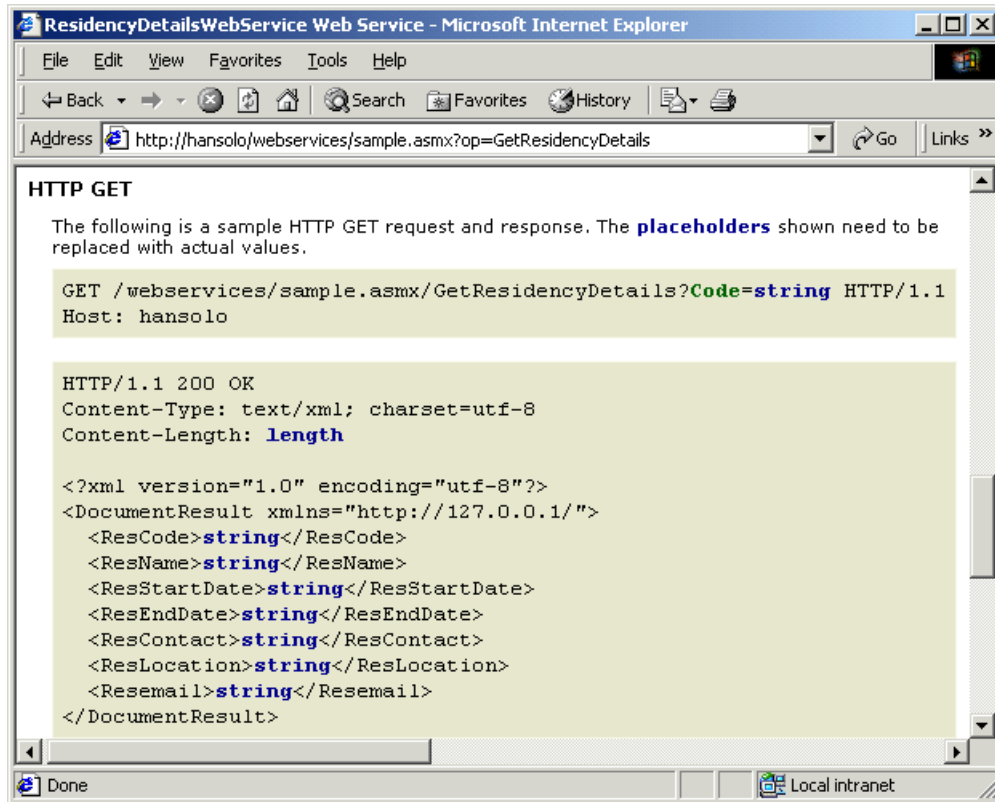


Figure A-32 Web Service request/response format - HTTP GET

► HTTP POST:

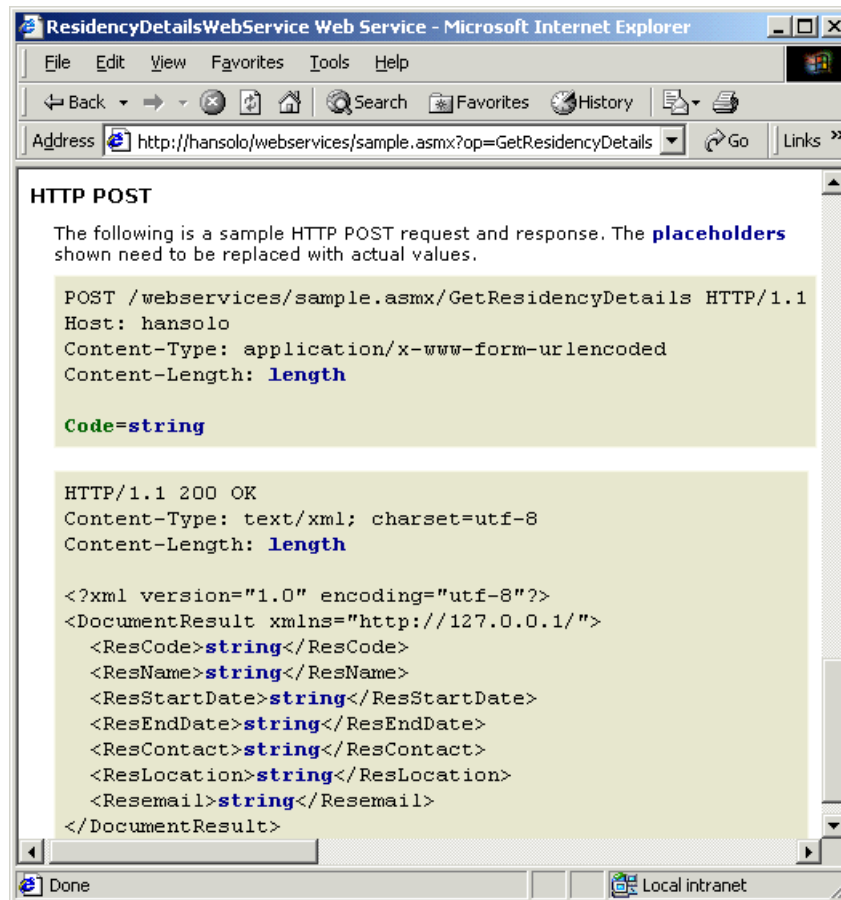


Figure A-33 Web Service request/response format - HTTP POST

► SOAP Request:

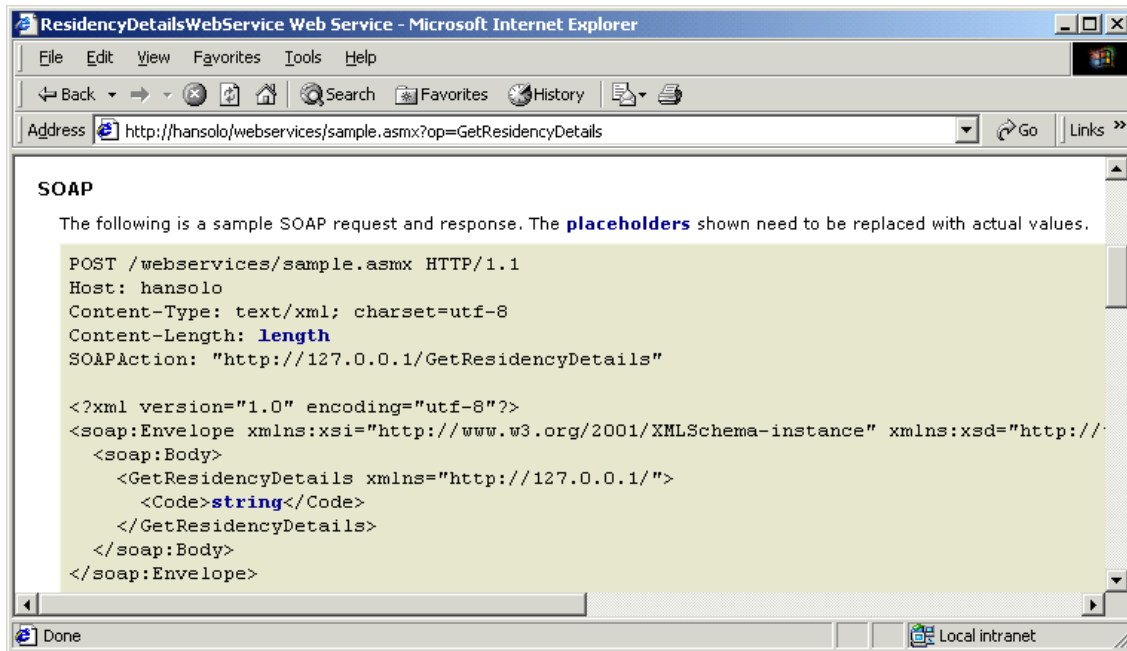


Figure A-34 Web Service request format - SOAP

► SOAP Response:

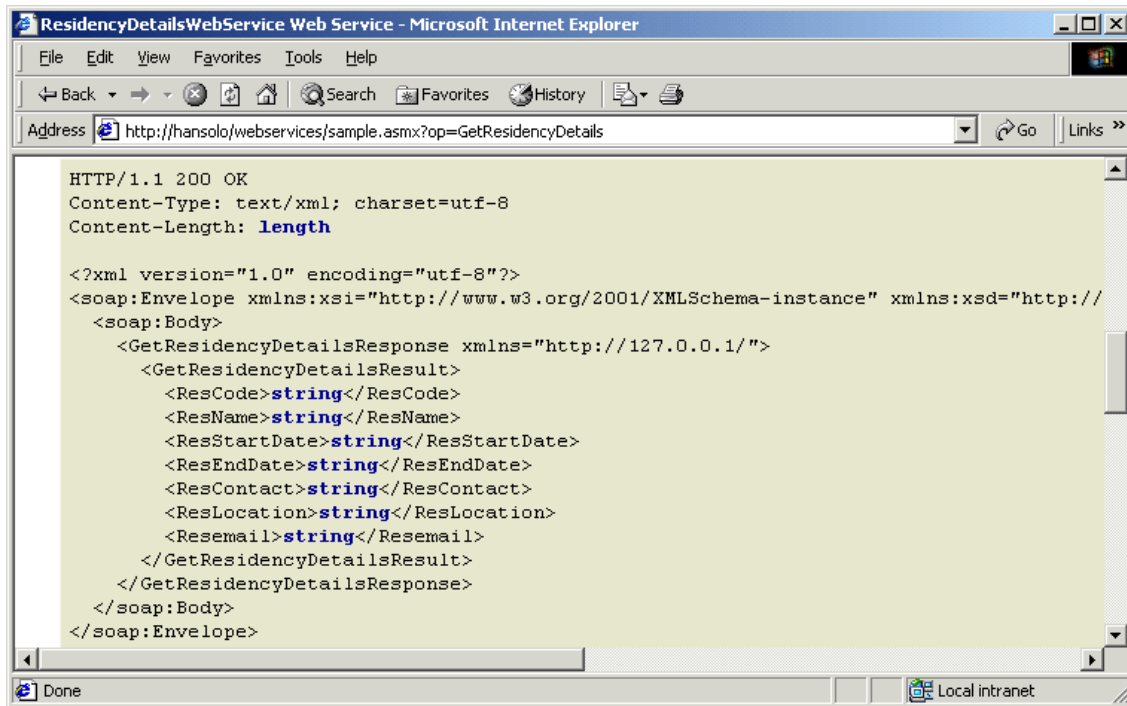


Figure A-35 Web Service response format - SOAP

Introduce a Residency Code and click the **Invoke** button. The XML result page is depicted in the next figure.

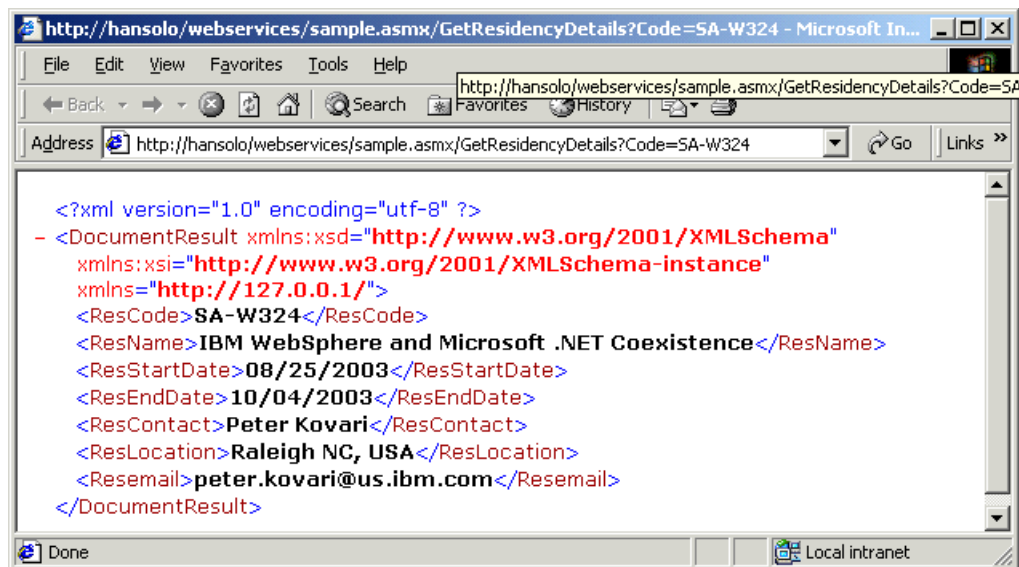


Figure A-36 Sample XML response



B

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/SG247027>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select **Additional materials** and open the directory that corresponds with the redbook form number, SG24-7027.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
SG247027.zip	Sample application in a zipped archive

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	20 MB
Operating System:	Windows 2000 Server or Professional
Processor:	1GHz or higher
Memory:	768 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, for example: C:\SG247027, and unzip the contents of the Web material zip file into this folder.

Asynchronous, stateless scenario sample

The Asynchronous, stateless scenario sample is under the AsyncStateless subdirectory.

The details for the scenario can be found in Chapter 5, "Scenario: Asynchronous" on page 229.

Synchronous, stateless scenario samples

The Synchronous, stateless scenario is split into two parts; one is WebSphere producer, .NET consumer; the other is WebSphere consumer, .NET producer.

The first part can be found under the SyncStateless-N2W subdirectory, the latter can be found under the SyncStateless-W2N directory.

Both samples have WebSphere Studio (.WSAD) and Visual Studio project (.VS) projects.

The Visual Studio project can be opened in Visual Studio; build the project and it should be ready to run.

The WebSphere Studio project needs a bit of work. Once the project is opened, it will look empty. Click **File -> Import -> Existing Project into Workspace** to import all the project directories. Once the projects are imported, click **Project ->**

Rebuild all from the menu. If you want to test the application in WebSphere Studio, then create a WebSphere V5.02 Test Environment and add the enterprise application.

The details for the scenario can be found in Chapter 7, “Scenario: Synchronous stateless (WebSphere producer and .NET consumer)” on page 297 and Chapter 8, “Scenario: Synchronous stateless (WebSphere consumer and .NET producer)” on page 329.

Abbreviations and acronyms

1PC	One Phase Commit	HIS	
2PC	Two Phase Commit	HTML	Hypertext Markup Language
AAT	Application Assembly Tool	IBM	International Business Machines Corporation
ACL	Access Control List	IDE	Integrated Development Environment
ADO	Active Data Objects	IIOP	Internet Inter-ORB Protocol
ASP	Active Server Page	IIS	Internet Information Services
BPEL	Business Process Execution Language	IL	Intermediate Language
BPEL4WS	BPEL for Web Services	ITSO	International Technical Support Organization
BSF	Bean Scripting Framework	J2EE	Java 2 Enterprise Edition
CCW	COM Callable Wrapper	J2SE	Java 2 Standard Edition
CLR	Common Language Runtime	JAR	Java Archive
CLS	Common Language Specification	JCA	Java Connector Architecture
COM	Component Object Model	JCP	Java Community Process
CORBA	Common ORB Architecture	JIT	Just-In-Time (compiler)
CTS	Common Type System	JMS	Java Message Service
DCOM	Distributed COM	JMX	Java Management Extensions
DD	Deployment Descriptor	JNDI	Java Naming and Directory Interface
DHTML	Dynamic HTML	JNI	Java Native Interface
DLL	Dynamic Link Library	JRE	Java Runtime Environment
DNA	Distributed interNet Applications	JSP	JavaServer Page
DTC	Distributed Transaction Coordinator	JSP	JavaServer Pages
EAR	Enterprise Archive	JVM	Java Virtual Machine
EIS	Enterprise Information System	LTPA	Lightweight Third Party Authentication
EJB	Enterprise Java Bean	MOM	Message-oriented Middleware
GAC	Global Assembly Cache	MOM	Microsoft Operations Manager
GRE	Generic Routing Encapsulation		
GUI	Graphical User Interface		

MSIL	Microsoft Intermediate Language
MTS	Microsoft Transaction Services
NIO	New Input/Output (Java)
ORB	Object Request Broker
OS	Operating System
PMI	Performance Monitoring Infrastructure
RAR	Resource Adapter Archive
RCW	Runtime-Callable Wrapper
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SCM	Service Control Manager
SOAP	Simple Object Access Protocol
SQL	
SSL	Secure Sockets Layer
SWAM	Simple WebSphere Authentication Mechanism
UDDI	Universal Description. Discovery and Integration
UDP	User Datagram Protocol
VBScript	Visual Basic Script
WAR	Web Archive
WLM	WebSphere Workload Management
WMI	Windows Management Instrumentation
WSDL	Web Services Description Language
WSIF	Web Services Invocation Framework
XML	

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 578. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *WebSphere Version 5 Web Services Handbook*, SG24-6891
- ▶ *IBM WebSphere V5.0 Security*, SG24-6573
- ▶ *IBM WebSphere V5.0 Performance, Scalability and High Availability*, SG24-6198
- ▶ *WebSphere Studio Application Developer Version 5 Programming Guide*, SG24-6957
- ▶ *IBM WebSphere Application Server V5.0 System Management and Configuration*, SG24-6195
- ▶ *WebSphere MQ Solutions in a Microsoft .NET Environment*, SG24-7012
- ▶ *WebSphere MQ Security in an Enterprise Environment*, SG24-6814
- ▶ *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- ▶ *Domino Designer 6: A Developer's Handbook*, SG24-6854

Other publications

These publications are also relevant as further information sources:

- ▶ *Applied Microsoft .NET Framework Programming*, Jeffrey Richter, ISBN 0-7356-1422-9
- ▶ *WebSphere MQ Using Java*, SC34-6066-01

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ Apache Jakarta project
<http://jakarta.apache.org/log4j>
- ▶ Apache Ant project
<http://ant.apache.org/>
- ▶ WebSphere InfoCenter
<http://www-3.ibm.com/software/webservers/appserv/infocenter.html>
- ▶ IBM WebSphere Application Server Web site
<http://www-3.ibm.com/software/webservers/appserv/>
- ▶ WebSphere prerequisites Web site
<http://www-3.ibm.com/software/webservers/appserv/doc/v50/prereqs/prereq502.html>
- ▶ Java Community Process
<http://jcp.org/aboutJava/communityprocess/final/jsr003/index.html>
- ▶ Microsoft Patterns
<http://www.microsoft.com/resources/practices/default.asp>
- ▶ Microsoft .NET information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcontrolexecutionlifecycle.asp>
- ▶ Microsoft .NET information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconintroductiontontserviceapplications.asp>
- ▶ Microsoft Server clustering
<http://www.microsoft.com/windows2000/technologies/clustering/default.asp>
- ▶ Microsoft load balancing
<http://www.microsoft.com/technet/treeview/default.asp?url=/TechNet/prodtechnol/windows2000serv/deploy/confeat/nlbvw.asp>
- ▶ .NET runtime
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstchart/html/vstchdeployingvsusingactivedirectory.asp
- ▶ Microsoft Operations Manager
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/momsk/html/momabout_71g1.asp

- ▶ Microsoft Systems Management Server
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/mom/sdk/html/momabout_9fvx.asp
- ▶ Microsoft Active Directory
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vstchdeployingvsusingactivedirectory.asp
- ▶ Patterns for e-business
<http://www.ibm.com/developerworks/patterns/>
- ▶ Enterprise Patterns
<http://www.enterpriseintegrationpatterns.com/>
- ▶ Interface Tool for Java (formerly known as IBM Bridge2Java) Web site
<http://www.alphaworks.ibm.com/tech/bridge2java>
- ▶ WS-Addressing
<http://www.ibm.com/developerworks/webservices/library/ws-add/>
- ▶ WS-I
<http://www.ws-i.org/>
- ▶ IBM WebSphere Integrator Web site
<http://www-306.ibm.com/software/integration/wmq/>
- ▶ IBM Redbooks Web site
<http://www.redbooks.ibm.com>
- ▶ Asynchronous Web Services Web site
<http://www-106.ibm.com/developerworks/webservices/library/ws-asynch1.html>
- ▶ Apache WSIF project
<http://ws.apache.org/wsif/>
- ▶ IIOP.NET Web site
<http://iiop-net.sourceforge.net/>
- ▶ GNU Licence information
<http://www.gnu.org/copyleft/lesser.html>
- ▶ Ja.NET Web site
<http://ja.net.intrinsyc.com/ja.net/info/>
- ▶ JNBridge Web site
<http://www.jnbridge.com>

- ▶ Janeva Web site
<http://www.borland.com/janeva/>
- ▶ SpiritWave Web site
<http://www.spirit-soft.com/products/wave/introducing.shtml>
- ▶ WebSphere MQ MA7P Spupport pack
<http://www-3.ibm.com/software/integration/support/supportpacs/individual/ma7p.html>
- ▶ Java RMI Web site
<http://java.sun.com/products/jdk/rmi/>
<http://java.sun.com/marketing/collateral/javarmi.html>
- ▶ OMG Web site
<http://www.omg.org>
- ▶ .NET remoting information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetremotingoverview.asp>
- ▶ Microsoft Web site
<http://www.microsoft.com>
- ▶ .NET information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconregisteringassemblieswithcom.asp>
- ▶ Microsoft IIS information
<http://www.microsoft.com/windows2000/en/server/iis/default.asp>
- ▶ J2EE RequestDispatcher information
<http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/RequestDispatcher.html>
- ▶ .NET security
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh19.asp>
- ▶ Microsoft SPNEGO details
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/http-sso-1.asp>
- ▶ W3C SOAP specification
<http://www.w3.org/TR/SOAP/>
- ▶ UDDI Web site
<http://www.uddi.org>

- ▶ WS-Security
<http://www.ibm.com/developerworks/library/ws-secure/>
- ▶ Web applications information
<http://www-106.ibm.com/developerworks/java/library/j-framework2/understanding.html>
- ▶ Sun JSP information
<http://java.sun.com/products/jsp/>
- ▶ Microsoft ASP.NET information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconpage.asp>
- ▶ .NET information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/gnconwebservicedirectivesyntax.asp>
- ▶ Sun EJB information
<http://java.sun.com/products/ejb/>
- ▶ Sun J2EE information
<http://java.sun.com/j2ee/>
- ▶ Microsoft clustering
<http://www.microsoft.com/windows2000/technologies/clustering/>
- ▶ IBM WebSphere Performance and Scalability best practices
http://www.ibm.com/software/webservers/appserv/ws_bestpractices.pdf
- ▶ WebSphere Application Server prerequisites
<http://www-3.ibm.com/software/webservers/appserv/doc/latest/prereq.html>
- ▶ Sun JAAS information
<http://java.sun.com/products/jaas>
- ▶ .NET security information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh06.asp>
- ▶ Microsoft WMI information
<http://msdn.microsoft.com/library/techart/mngwmi.htm>
- ▶ Microsoft Systems Management Server
<http://www.microsoft.com/smsgmt/default.asp>
- ▶ SOAP encoding performance
<http://www-106.ibm.com/developerworks/webservices/library/ws-soapenc/>

- ▶ Web Services Quality of Service
<http://www-106.ibm.com/developerworks/webservices/library/ws-soapenc/>
- ▶ Web Services security roadmap
<http://www.ibm.com/developerworks/webservices/library/ws-secmap/>
- ▶ WS-Coordination
<http://www.ibm.com/developerworks/webservices/library/ws-coor/>
- ▶ WS-Transaction
<http://www.ibm.com/developerworks/webservices/library/ws-transpec/>
- ▶ W3C DOM specification
<http://www.w3.org/TR/DOM-Level-3-Core/core.html#ID-1590626202>
- ▶ .NET information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfwebservicedescriptionlanguagetoolwsdl.exe.asp>
- ▶ Microsoft Tlbimp tool information
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfTypeLibraryImporterTlbimp.exe.asp>
- ▶ ASP.NET Web applications
<http://support.microsoft.com/default.aspx?scid=kb;EN-US;q325056>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols

- .aspx file 447
- .NET 466
 - Application State 381
 - Automatic transactions 492
 - creating a Web Service 331
 - Manual transactions 492
 - Page State 382
 - Request State 382
 - Session State 382
 - Web Service client 321
- .NET application 167, 174, 182
- .NET architecture 50, 54
- .NET client 216
- .NET client application
 - developing 317
- .NET COM interop 212
- .NET configuration 92
- .NET consumer application 265
- .NET Debug configuration 83
- .NET Enterprise Services 53
- .NET environment 52
- .NET Interop 266
- .NET key values 52
- .NET languages 60
- .NET Presentation layer 148
- .NET redistributable 59
- .NET Remoting 57, 265, 267, 454
- .NET security 420
- .NET Service Consumer 313
- .NET service provider 331, 537
- .NET service proxy 147
- .NET service stub 141
- .NET Software Development Kit 69
- .NET superclass 292
- .NET Web Service test 341

A

- AAT 34
- Abstract Window Toolkit 442
- Abstraction 18
- Access control 17
- Access Control list 419

- ACT 87
- Active Data Objects 51
- Active Directory 92, 422
- Active Server Pages 51
- Active Template Library 283
- ActiveX
 - Client 271
 - Control 441
- ActiveX Bridge 210, 271, 468
 - best practices 277
 - constraints 274
 - solution 279
- ActiveX Data Objects 458
- ActivitySession 491
- Admin service 41
- Administration 44, 90
- Administrative Console 45
- Administrative Tools 91
- ADO 458
- ADO.NET 67, 458, 460
 - authentication 72
 - Command Builders 462
 - IDataAdapter 461
 - IDataReader 461
 - IDbCommand 461
 - IDbConnection 461
 - IDbTransaction 461
 - Legacy Support 462
- amqmdnet.dll 247
- Ant 33
- Apache WSIF 223
- Applets 6
- Application Assembler 22
- Application Assembly Tool 34
- Application Center Test 87
- Application Client Module 11
- Application clients 5
- Application Component Provider 22
- Application components
 - J2EE 10
- Application configuration file 62
- Application Deployer 22
- Application life cycle management 44
- Application Logging 20, 73

- Application security 484
- Application Server 39
- application server services 40
- Application testing 82
- architecture 5
- argument by value 126
- argument paradigms 111
- Arrays 301
- ASMX Handler 450
- ASP.NET 55, 446
 - Application Resources 369
 - authentication 72
 - component trace 74
 - consuming a Web Service 322
 - Web Service client 322
- Assembly 60
 - create 332
 - life cycle 65
 - Metadata 60
 - versioning 74
- assembly resources 61
- asynchronous communication 234
- Asynchronous Interaction 117
- asynchronous interaction
 - security 235
 - transactions 236
- asynchronous interaction scenario 229
- asynchronous interoperability 237
- asynchronous message queuing 230
- asynchronous messaging
 - transport 238
- asynchronous processing 122
- asynchronous programming 230
- asynchronous service 244
- ATL 283
- Atomic transactions 505
- Atomicity 489
- attribute properties 340
- Auditing 17
- Authentication 17, 419, 485, 488
 - Externalizing 422
 - integration 421
 - mechanisms 20
 - services 72
- Authorization 17, 419, 486, 489
 - externalize 425
 - integration 425
- Availability 482
- AWT 442

- client 138
- AWT client application 145, 152

B

- bean managed persistence 453
- Bean Scripting Framework 45
- BizTalk 467
- BMP 453
- bottom interface 102
- bottom up scenario 103
- boxing 128
- BPEL 438
- BPEL4WS 438
- Bridge2Java 272, 289
- BSF 45
- Business activities 506
- business functionality 281
- business interface 155
- Business Layer 7, 56, 95
- business layer artifact 179
- business logic 452
- business logic to business logic scenario 174
- business logic to resource scenario 182
- business objects 58
- Business Process Execution Language 438
- business tier artifact 177
- bytecode 266

C

- C# 75
- C# console application 533
- Calculator example 303
 - implementation 304
- Calculator implementation 352
- Calculator.dll assembly 332
- Calculator.jar 281, 303, 348
- callback functions 231
- Capability list 419
- CCW 539
- Cell 39
- chatty interface 113
- child elements 432
- Client activation 455
- Client Certificate authentication 488
- Client container 13
- Client Layer 7, 96
- client logic to business logic scenario 152
- client logic to client logic scenario 138

- client logic to presentation logic scenario 145
- Clients
 - .NET 443
- CLR 52, 63, 265
- CLR as COM 213
- CLS 60, 63
- cluster nodes 476
- CMP 453
- Code Animation 83
- Code-based security 487
- COM 50, 272, 274
- COM ActiveX Bridge Gateway DLL 282
- COM Callable Wrapper 539
- COM components 272, 289
- COM service proxy 212
- COM Wrapper DLL 282
- Command Line
 - deployment 35
 - packaging 33
- Common Language Runtime 52, 63, 265
- Common Language Runtime features 64
- Common Language Specification 60, 63
- Common Type System 63
- Communication Protocol 136
- Compatibility testing 18
- compensation 490
- compiler 23
- Complex Data Type
 - Web services 309
- Component Load Balancing 477
- Component Object Model 50, 274
- Component Object Request Broker Architecture 452
- compound types 301
- Concentric Layered Application Model 98
- Confidentiality 17
- Configuration
 - WebSphere 11
- Configuration files 62
- Configure
 - messaging 254
- conjoined 186
- conjoined model 94
- Consistency 489
- Console Client 443
- Constructor Strings 277
- container managed persistence 453
- Containers 12
- Continuous availability 482

- Cookies 379, 384, 386
- CORBA 452
- CosNaming 14
- Create
 - message driven bean 244
- cross-layer interaction 132
- cross-network 123
- cross-process 123
- cross-technology interaction 132
- CTS 63
- Custom User registry 422
- CVS 29

D

- Daemon process 255
- Data encryption 71
- Data integrity 17
- Data layer 57
- Data Model 268
- Data Privacy 17
- Data propagation 395
 - Data Format 401
 - Data Transport 397
 - Data Types 399
 - Form-based 397
 - HTTP Request Forwarding 397
 - JavaScript URL Construction 397
 - URL Redirection 397
 - URL Redirection Solution 402
- Database 456
- Database access 456
- database connection pooling 67
- DCOM 50, 207, 452
- Deadlocks 32
- Debug Builds 82
- Debugging 82
- Decoration 157
- decorator 170, 177
- delegation 419
- delegation model 94
- delegation threads 231
- Deployment 33, 35, 345
 - By copying files 88
 - Setup 88
- deployment 88
- deployment descriptor 28
- Deployment Manager 39, 43
- Deployment project types 89

- Deployment Units 60
- Develop
 - assembly 332
- DHTML 440
- directory services 195
- DISCO 435
- disco tool 69
- distributed application 112
- Distributed COM 50
- Distributed Component Object Model 452
- Distributed components 451
- Distributed interNet Applications 51
- Distributed Object Architecture 129
- Distributed technologies 451
- Distributed Transaction Coordinator 70
- DLL 60, 266
- Document call paradigm 124, 130
- Document interface style 124, 130
- Document Model implementation 316
- Document Style Model implementation 309, 344, 351
- Document style SOAP 124
- Document Style vs RPC Style 327
- Domino Nomenclature 516
- DTC 70, 493
- Durability 490
- Dynamic deployment 498
- Dynamic Hypertext Markup Language 440
- Dynamic Link Library 266
- dynamic link library 60
- Dynamic Reloading 36
- dynamic Web pages 441
- Dynamic Web Project 309

E

- EAR 27, 305
- Edge Components 44
- editors 27
- EIS 11, 466
- EJB 6, 265, 452
- EJB bean 453
- EJB client 453
- EJB component 453
- EJB container 12, 452
- EJB deployment descriptor 453
- EJB interfaces 453
- EJB JAR 28
- EJB module 10

- EJB quality of service decorator 156
- EJB server 452
- EJB Web Service 312
- elementary integration 293
- embedded messaging server 42
- embedded Web server 40
- encoding styles 501
 - Literal 431
 - SOAP Encoded 431
- Enterprise Applications 183
- Enterprise Archive 27
- Enterprise Information Systems 11, 466
- Enterprise Java Bean 6, 265, 452, 458
- Enterprise Java Bean Module 10
- Entity beans 453
- Environment
 - J2EE 5
- Error Handling 320
- Export application 34
- Extended Solution 359
- Extensibility 18
- eXtensible Markup Language 430
- Externalizing authorization 425

F

- façade 114
- Failover 20, 72, 483
- Fat .NET Client 139
- fat client 139
- Fat Java Client 139
- Fat Java Client application 138
- File Authorization 489
- FindClass() method 272
- Flexibility 18
- Form-based authentication 488
- Form-based Propagation
 - implementation 412
 - Solution 404
- Form-based Propagation Solution 404
- Forward 164
- Forwarding 394
- Fundamental Interaction Classifications 110

G

- GAC 75
- garbage collection 13, 266
- Generic Routing Encapsulation 477
- GetArgsContainer() method 272

- Global Assembly Cache 61, 75, 241
- Global.asax 337
- graphical user interface 442
- GRE 477
- GUI 161

H

- Hardware security 484
- Head Protocol 136
- high availability 482
- Horizontal Scaling 478, 480
- Host Integration Server 467
- Hot Deployment 36, 499
- HTML 440
- HTTP 376
- HTTP Get 537
- HTTP Plugin 369
 - IIS 370
 - manual configuration 371
- HTTP Post 538
- HTTP transport 234
- HttpChannel 267
- HttpSession interface 378
- HttpSessionState object 378
- Hypertext Markup Language 440

I

- IBM ActiveX Bridge 271
- IBM Bridge2Java 272
- IBM Interface Tool for Java 203, 272, 468
- ICalculator interfaces 332
- IDE 24, 75
- IDispatch interface 271
- IIOP 15, 267
- IIOP Remoting Channels 209
- IIOP.NET 226
- IIS 50, 64, 368
- IIS HTTP Plugin 370
- impersonation 419
- in/out arguments 127
- Independence 18
- in-only arguments 125
- in-process 123
- instrumenting 85
- Integrated Debugger 29
- Integrated Development Environment 24, 75
- Integrated Security 418
- Integrating authentication 421

- Integrating Authorization 425
- integration classifications 143
- Integration Layer 8, 96
- interaction classifications 132
- interaction dynamics 111
- interface styles 111
- Interface Tool for Java 272
 - constraints 274
 - solution 289
- Intermediate Language 63
- Internet boundaries 200
- Internet Information Services 50, 64, 368
 - Caching 480
 - configuration 370
 - life cycle 66
- Internet Inter-ORB Protocol 15, 267
- Internet Services Manager 64, 92
- inter-node call 267
- inter-node synchronous stateful interaction 269
- interoperation integration layer 101, 133
- Interoperation Layer Abstraction 101
- interpreter 23
- inter-process 123
- inter-process call 267
- inter-process communication 127
- inter-process communication mechanism 206
- ISAPI Web Debug Tool 85
- Isolation 18, 185, 490

J

- J2C 466
- J2EE application client 442
- J2EE Application Resources 369
- J2EE architecture 5
- J2EE clients 441
- J2EE Connector Architecture 466
- J2EE Enterprise Application 305
- J2EE environment 5
- J2EE Product Provider 21
- J2EE, Java 2 Enterprise Edition 5
- Ja.NET 226
- JACL 45
- Janeva 226
- Java Applets 440
- Java applets 442
- Java Bean Web Service 312
- Java client
 - thin 441

- Java Client JAR 28
- Java Community Process 8
- Java Database Connectivity 459
- Java Developer's Kit 23
- Java language 9
- Java Management Extensions 46, 479, 493
- Java Messaging Service 235, 239, 463
- Java Naming and Directory Interface 14, 269
- Java Native Interface 9, 266
 - limitations 275
- Java Presentation layer 148
- Java proxy 178, 289
- Java proxy presents 155
- Java Remote Method Protocol 267
- Java Runtime Environment 9
- Java runtime environment 11
- Java runtime platforms 9
- Java Server Pages
 - JSP 444
- Java Server Pages Standard Tag Library 444
- Java service proxy 141
- Java Servlet API 443
- Java stub represents 156
- Java Virtual Machine 9, 11, 440
- JavaScript 440
- JAX-RPC 439
- JCA connector 188
- JCA container 13
- JDBC 459
 - CallableStatement 460
 - Connection 460
 - Connection pooling 460
 - DatabaseMetaData 460
 - DataSource interface 460
 - DriverManager 460
 - ResultSetMetaData 460
 - Statement 460
 - Statement pooling 460
- JDK 23
- JMS 235, 239, 463
 - .NET messaging 253
 - Header 239
 - Server 39
 - TextMessage 245
- JMX 46, 479, 493
 - Agent level 494
 - Distributed services level 494
 - Instrumentation level 494
- JMX architecture 494

- JNBridge 226
- JNDI 14, 269
- JNI 9, 266
- JRE 9
- JRMP 267
- JSP Tags 444
- JSR 101 41
- JSR 109 41
- JSTL 444
- Just-In-Time Debugging 84
- JVM 9, 11
 - logs 20

K

- Kerberos ticket 439

L

- language independence 55, 266
- Last Participant Support 491
- Layer Interaction Classifications 110
- Layered Application Model 95
- layers integration 99
- LDAP User registry 422
- Life cycle management 13, 65
- Lifetime lease 456
- Lightweight Third Party Authentication 20, 486
- listener 85
- Literal 432
- Load Balancing 20, 72
- Load Testing 85
- Logical layers
 - J2EE 6
- Long-Running Service Providers 256
- loosely coupled implementations 94
- Lotus Domino 513
 - .NET client 531
 - .NET Web Service 556
 - COM server 540
 - IIS integration 541
 - New database 517
 - new Forms and Views 518
 - new Lotus Script Library 524
 - new Lotus Script Web Agent 520
 - sample Database 554
 - Service provider 515
 - Test 534
 - Using COM interface 538
 - Web Service 526

- WSDL Page 526
- Lotus Script library 515
- Lotus Script Web Agent 515, 520
- Low throughput 32
- LTPA 20, 486

M

- MA7P 235
- Machine configuration file 62
- Maintainability 498
- Maintaining state 281
- Manageability 493
- managed code 56, 266
- Managed Heap 65
- Managed Process 39
- manifest 61
- master repository 47
- MBeans 46
- MDB 234, 243
- meet in the middle
 - interface 102
 - scenario 103
- Memory leaks 32
- message
 - request 238
 - response 238
- Message Format 238
- Message Headers 239
- message payload 239
- Message Queue Interface 464
- Message Queueing Systems 195
- message routing 122
- message styles
 - document 431
 - RPC 431
- Message Transformation 240
- message transformation 122
- Message-Driven Bean 234, 238, 243, 454
 - create 244
- Message-oriented Architecture 130
- Message-oriented middleware 230
- message-oriented solution 190
- messaging
 - configure 254
 - get in .NET 250
 - JMS in .NET 253
 - put in .NET 248
- Messaging authentication 72

- messaging middleware 118, 219, 463
- META-INF 28
- Microsoft Clustering 475
- Microsoft Distributed Transaction Coordinator 493
- Microsoft DNA 51
- Microsoft Intermediate Language 63
- Microsoft Internet Information Services 368
- Microsoft Message Queuing 70
- Microsoft MSMQ 124
- Microsoft Operations Manager 91, 496
- Microsoft patterns 55
- Microsoft technology 59
- Microsoft Transaction Services 50
- Microsoft Visual Source Safe 499
- middle interface 102
- middleware logic 313
- Model-View-Controller 56, 161, 443, 447
- model-view-controller
 - pattern 169
- MOM 496
- MQ Classes for .NET 235, 238, 247, 465
 - install 241
- MQ message 240
- MQ Transport for SOAP 234
- MQI 464
- MSIL 63
- MSMQ 70
- MTS 50
- multiple coexisting applications 98
- MVC 161, 169, 443

N

- Native log 20
- Network Deployment 38, 42
- Network Load Balancing 73, 477
- Network security 484
- NewInstance() method 272
- n-layer model 55
- Node 39
- Node Agent 39, 43
- Non-distributed transactions 70
- Non-repudiation 17
- notification 234
- NT services 67

O

- Object Linking and Embedding Database 457
- Object pooling 14, 67

- ODBC 457
- ODBC Bridge 462
- OLE/DB 457
- one to one interaction 133
- one-way messaging 234
- onMessage() method 244
- Open Database Connectivity 457
- open messaging standard 240
- Operating System level security 71
- Operating system security 484
- Operating System Services 183

P

- page directives 447
- parallel coexistence 96
- Parameter Validation 345
- Pass by Reference 127
- Pass by Value 125
- Passport authentication 488
- Performance 67, 85, 478
- performance counter 86
- Performance counters 73
- Performance Monitor 86
 - Windows 481
- Performance Monitoring Infrastructure 479
- Performance profiling 32
- Perspectives 27
- phased migration 149
- platform independence 55
- Platform Support 59
- PMI 479
- Portability 17, 499
- presentation 262
- Presentation layer 7, 55, 96
- presentation logic to business logic scenario 167
- presentation logic to presentation logic 160
- Private assembly 61
- Project types 80
- Propagation Methods 401
- proxy-stub pattern 137, 155–156, 169, 177
- pseudo Resource layer 101
- publish/subscribe pattern 130

Q

- QoS 471
- QoS Decoration 157
- Quality of Services 471
- queue manager 243

R

- RAD 77
- Rapid Action Development 77
- rapid development 24
- RAR 466
- Rational ClearCase 29
- RCW 539
- Re-authentication 419
- Redbooks Web site 578
 - Contact us xv
- Redirecting 394
- redirection 162
- redistributable runtime 90
- Reflection 271
- registry entries 295
- Relation Database Management Systems 195
- Relational Database 183
- relational database resource 185
- remote deployment 31
- Remote invocation 68
- Remote Method Invocation 14, 265, 267, 452
- Remote Object Discovery 14
- Remote Procedure Call 123
- remote queue 243
- remote WebSphere server 31
- Remoting 68, 267, 454
 - Activation 455
 - Channel 454
 - Formatter 454
 - Object Life-Cycle 456
 - Transport 454
- Remoting authentication 72
- remoting channel 270
- Remoting transport 269
- replyTo queue 245
- resource 195
- Resource Adapter Archive 466
- Resource Adapter Module 11
- Resource interfaces 201
- Resource Layer 95
- Resource layer 8, 57
- Resource permissions 484
- resource to resource scenario 195
- RMI 14, 265, 267, 452
- RMI transport 268
- Role-based security 71, 420, 487
- root element 431
- Routing SOAP request 520
- RPC 123, 265

- RPC interface style 123
- RPC Model implementation 315, 339, 350
- RPC Style vs Document Style 327
- Runtime 37, 90
- Runtime code level security 71
- Runtime-Callable Wrapper 539

S

- Sample application 303
 - implementation 304, 347, 352
- Scalability 472
- SCM 29, 66
- scripting client 42
- SDK 69
- Secure interoperability 18
- Secure Sockets Layer 71, 464
- Security 17, 70, 235, 302, 346, 484
 - Web application 418
- Security configuration file 62
- Security integration 418
- Server activation 455
- Server Clustering 72
- Server pages 443
- Server resource management 44
- server-side COM object 278
- server-side components 66
- Service Binding/Hosting 136
- Service Consumer 240, 282
- Service Consumer implementation 346
- Service Control Manager 66
- Service Discovery Mechanism 136
- service façade 118–119
- Service log 21
- Service Provider 240, 255, 281
- service proxy 118–119
- Serviced components 70
- Service-oriented Architecture 131, 299
- Servlets 443
- Session 376
- Session beans 453
- session expiration 378
- Session life cycle 394
- Session Object Mapping 385
- Session persistence 380, 384
 - Database 380
 - In process mode 384
 - Memory-to-memory replication 380
 - SQL server mode 385

- State server mode 384
- Session State 382
- Session State Interoperability 376
- Session State Life Cycle 378, 383
- Session tracking 392
- Session Tracking Mechanisms 379, 383
- Shared assembly 61
- Shared Presentation Components 368
- shared resource 183
- Shared User registry 421
- Sharing Session Data 390
 - Data serialization and deserialization 392
 - Database 391
 - Messaging point-to-point 392
 - Messaging publish/subscribe 393
 - Push 394
- Simple Object Access Protocol 433
- Simple WebSphere Authentication Mechanism 20
- SimpleDataGateway 447
- single point of failure 483
- skeleton 15
- SMS 92
- SOA Services 183
- SOAP 300, 433, 538
 - encoding 433
 - Encoding rules 434
 - Envelope 434
 - RPC 434
- SOAPAction header 450
- SoapRpcMethod attribute 339
- SoapRpcService attribute 339
- soapsuds tool 69
- solution candidates 202
- Source code management 29
- SpiritWave 226
- SPNEGO 424
- Spy++ 85
- SQL 457
- SSL 71, 464
 - Sessions 379
- Standard Support 59
- state limitations 120
- state maintenance 376
- state management 120
- State Management Objects 381
- state management objects 378
- state object 116
- state-dependent operations 121
- Stateful asynchronous 232

- Stateful Asynchronous Interaction 120
- Stateful Asynchronous solution 219
- Stateful interaction 112
- stateful interaction 232, 264
- stateful messages 121
- Stateful Remote Invocation 267
- stateful scenario
 - Java implementation 263
- stateful service provider 262
- Stateful synchronous interaction 112, 264
- Stateful Synchronous solution 203
- Stateless Asynchronous Interaction 117
- stateless communication 299
- Stateless Interaction 115
- Stateless interaction 231
- stateless services 234
- Stateless Synchronous Interaction 115
- Stateless Synchronous solution 214
- Static Content 370
- Structs 301
- Structured Query Language 457
- stub 15
- SupportPac 235
- SWAM 20
- Swing 442
- Swing client 138
- Swing client application 145, 152
- synchronization 183
- synchronous communication 299
- Synchronous Interaction 112
- Synchronous, Stateful scenario 261
- Synchronous, Stateless scenario 297, 329
- System Administrator 22
- System resource constraints 32
- Systems Management Server 92

T

- TAI 423
- Tail Protocol 136
- Tamper-proof assemblies 71
- TcpChannel 267
- Technical Solution Mapping 110, 202
- Test 82
- Thin .NET client 139
- thin client 139
- Thin Java Client 139
- Thin Java client 441
- Tivoli Access Manager WebSEAL 422, 425

- Tivoli Performance Viewer 479
- Tlbimp 553
- Tool Provider 21
- Tools
 - disco 69
 - ISAPI Web Debug 85
 - soapsuds 69
 - wsdl 69
- top down scenario 103
- Trace class 73
- Trace log 21
- Tracing 85
- Transaction Coordinator 236
- Transaction Management 16, 69
- Transactionality 489
- Transactions 236, 302
- Transparency 17
- Transport mechanism 136, 300
- transports 234
- trigger 197
- Triggering 255
- Trust Association Interceptor 423
- two-phase commit 490
- Type handling 320
- Type Library 295
- Type Metadata 60

U

- UDDI 43, 434
 - Binding template 435
 - Business entity 435
 - Business service 435
 - Publisher assertions 435
 - Taxonomy 435
 - tModel 435
- UDDI data model 435
- UDDI registry 434
- UDDI server 435
- UDDI4J 434
- UDP 477
- unified cross-technology architectural model 105
- Unit Of Work 236
- Unit of Work 189
- Unit Test Environment 30
- Unit Testing 82
- Universal Description, Discovery, and Integration 434
- Universal Test Client 31

- unmanaged code 56
- untyped binary arguments 124
- URL Authorization 489
- URL encoding 399
- URL Redirection
 - implementation 405
- URL Redirection Solution 402, 404
- URL Rewriting 379, 384, 388
- Use of cache 346
- User Datagram Protocol 477
- user experience 262
- user interface 161
- User registry 19
- User/group management 484

V

- VBScript 440
- verbose interface 113
- Versioning 74
- Vertical scaling 478, 480
- views 27
- virtualized security architecture 420
- Visual Source Safe 77, 81
- Visual Studio .NET 75, 77
 - debugger 82

W

- WAR 28
- Web application 28, 440
 - layer 161
- Web browser client 440
- Web Client 441
- Web components
 - J2EE 6
- Web container 12, 64, 147, 160, 167
- Web deployment descriptor 309
- Web forward 164
- Web Interoperability scenario 367
- Web Module 11
- Web project 306
- Web redirection 162
- Web References 443
- Web server plugin 40
- Web Service
 - .NET 331
- Web service
 - .NET client 531
 - .NET proxy 532

- Deployment 345
 - Generating proxy 325
- Web Service client
 - Configuration files 320
- Web service client
 - Generate 350
- Web service creation 307
- Web Service Description Language 430
- Web service proxy 351
- Web service wizard 307
- Web Services 215, 233, 430, 501
 - ASP.NET 449
 - Asynchronous Façade 223
- Web services 15, 41, 69
 - Complex Data Type 309
 - Data types 326
 - Deployment 311
 - Performance 501
 - Reliability 508
 - Security 503
 - Transactionality 505
 - type compatibility 320
- Web services authentication 72
- Web Services enabled resource 191
- Web Services Gateway 43, 437
- Web Services Invocation Framework 436
- Web Services security 438
- Web Services WS-Addressing 209
- Web.config 62, 337
- WEB-INF 28
- WebMethod attribute 340
- WebSEAL 422, 425
 - junction 423
 - LTPA 423
 - SPNEGO 424
 - TAI 423
- WebSphere
 - Application Security 485
 - cluster support 472
 - configure IIS 370
 - Global Security 485
 - runtime platforms 500
 - Scalability 472
 - Session Management 377
 - Transaction support 490
- WebSphere Administrative Console 36
- WebSphere application 174, 182
- WebSphere Application Server 38–39
- WebSphere Application Server Network Deploy-

- ment 38, 42
- WebSphere Client Container 138
- WebSphere configuration 11
- WebSphere Integrator 122, 240
- WebSphere MQ 219, 221, 230, 234, 464
- WebSphere MQ messaging 124
- WebSphere MQ Transport for SOAP 465
- WebSphere security 420
- WebSphere Service Provider 303
- WebSphere Studio
 - Web service 307
 - Web service client 350
- WebSphere Studio Application Developer 24–25
- WebSphere Studio Application Developer - Integration Edition 25
- WebSphere Studio Enterprise Developer 25
- WebSphere Studio Site Developer 25
- WebSphere Studiomin 35
- WebSphere workload management 473
- Windows
 - Clustering 484
 - Performance Monitor 481
- Windows .NET clients 443
- Windows applications 80
- Windows Applicaton 246
- Windows authentication 488
- Windows client application 313
- Windows event log 74
- Windows Form application 139, 146, 152
- Windows Forms 53
- Windows Management Instrumentation 91, 496
- Windows Management Interface 74
- Windows Server 2003 Clustering 475
- Windows Service 67, 256
- WLM 473
- WMI 74, 91, 496
- workbench 27
- workspace 27
- Writing C# 75
- WS-Coordination 505
- WSDL 430
 - Binding 431
 - Message 430
 - Port 431
 - PortType 430
 - Service 431
 - Types 430
- wsdl tool 69
- WSGW 437
- WS-I Basic Profile 41
- WSIF 436
- WS-Interoperability 508
- WS-Security 192, 504
- WS-Transaction 192, 505

X

- X.509 certificates 439
- XCopy Deployment 499
- XJBInit() method 272
- XML 430
- XML configuration 11
- XML digital signature 438
- XML document 124
- XML DOM 432
- XML encryption 438
- XML Schema 124, 432
- XML Schema Definition 301



Redbooks

WebSphere and .NET Coexistence

(1.0" spine)
0.875" <-> 1.498"
460 <-> 788 pages



Redbooks

WebSphere and .NET Coexistence

In-depth view of the supporting technologies

Interoperability scenarios and their implementation

Working sample code

This IBM Redbook explores the different coexistence scenarios for the WebSphere and .NET platforms. This book is a good source of information for solution designers and developers, application integrators and developers who wish to integrate solutions on the WebSphere and .NET platforms.

Part 1, “Introduction” is a quick introduction to the J2EE (WebSphere) and .NET technologies. It also depicts a basic architectural model that can be used to represent both WebSphere applications and .NET applications.

Part 2, “Scenarios” identifies several potential technical scenarios for coexistence via point-to-point integration between applications deployed in the IBM WebSphere Application Server and applications deployed in the Microsoft .NET Framework. This part provides in-depth technical details on how to implement certain scenarios using today’s existing technologies.

Part 3, “Guidelines” provides general guidelines for solution developers. A list of supporting technologies can help with the solution implementation. The Quality of Service chapter is a collection of services available on both platforms.

The Appendixes go further by using other IBM technologies and describing two integration solutions between Lotus Domino and .NET applications.

**INTERNATIONAL
TECHNICAL
SUPPORT
ORGANIZATION**

**BUILDING TECHNICAL
INFORMATION BASED ON
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks