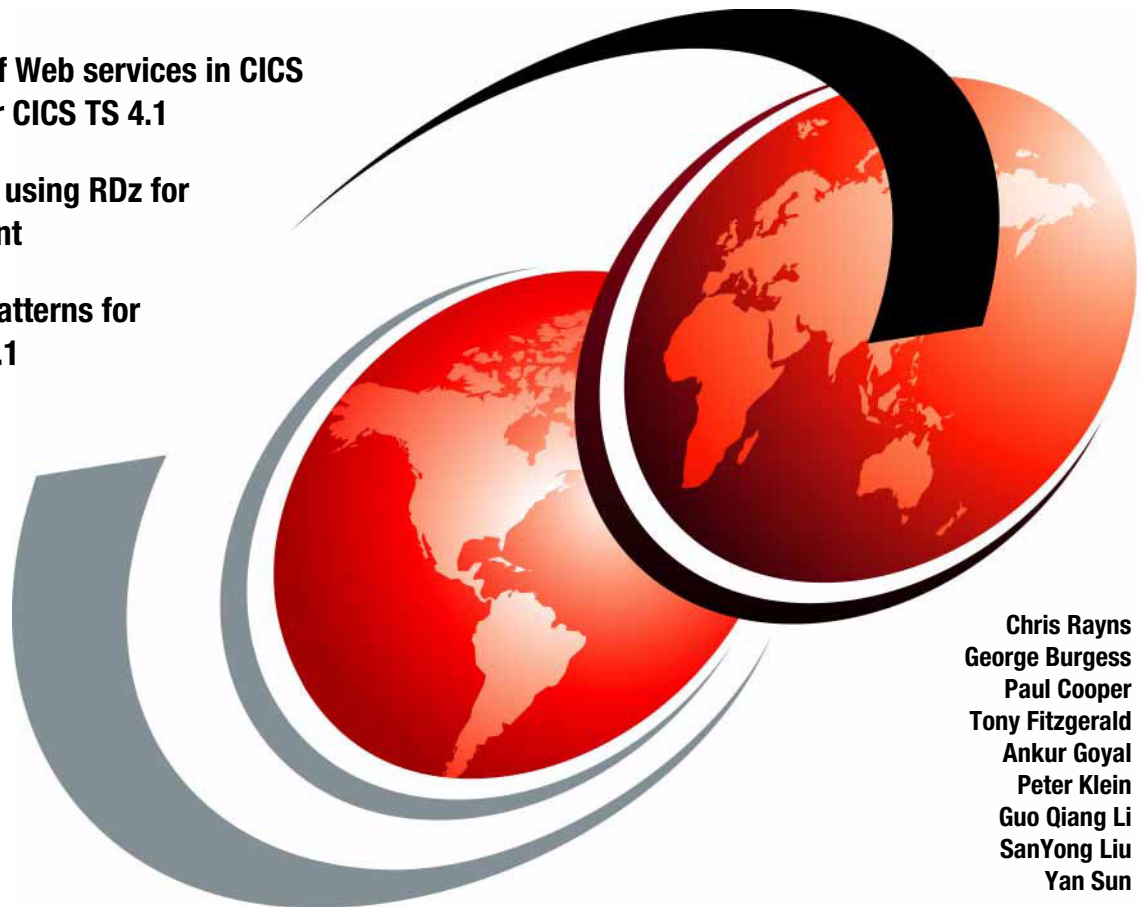


Application Development for CICS Web Services

Overview of Web services in CICS
updated for CICS TS 4.1

Experience using RDz for
development

New SOA patterns for
CICS TS V4.1



Chris Rayns
George Burgess
Paul Cooper
Tony Fitzgerald
Ankur Goyal
Peter Klein
Guo Qiang Li
SanYong Liu
Yan Sun



International Technical Support Organization

Application Development for CICS Web Services

January 2010

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

Second Edition (January 2010)

This edition applies to Version 4, Release 1, of CICS Transaction Server.

© Copyright International Business Machines Corporation 2010. 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 who wrote this book	xi
Become a published author	xiii
Comments welcome	xiv
Summary of changes	xv
January 2010, Second Edition	xv
Chapter 1. Overview of Web services	1
1.1 Introduction	2
1.2 Service-oriented architecture	2
1.2.1 Characteristics	4
1.2.2 Web services versus SOAs	4
1.3 Web services	5
1.3.1 Properties of a Web service	5
1.3.2 Core standards	6
1.3.3 Web Services Interoperability group	9
1.3.4 Additional standards	9
1.4 IBM WebSphere Service Registry and Repository	11
1.5 SOAP	11
1.5.1 The envelope	11
1.5.2 Communication styles	16
1.5.3 Encodings	16
1.5.4 Messaging modes	17
1.6 WSDL	18
1.6.1 WSDL Document	18
1.6.2 WSDL document anatomy	19
1.6.3 WSDL definition	23
1.6.4 WSDL bindings	29
Chapter 2. CICS implementation of Web services	31
2.1 Support for Web services in CICS	32
2.1.1 Core aspects of Web services in CICS	32
2.2 Tools for application deployment	34
2.2.1 CICS Web Services Assistant	34
2.2.2 IBM Rational Developer for System z	35

2.2.3 Other Options	35
2.3 CICS as a service provider	36
2.3.1 Preparing to run a CICS application as a service provider	36
2.3.2 Processing the inbound service request	38
2.4 CICS as a service requester	40
2.4.1 Preparing to run a CICS application as a service requester	40
2.4.2 Processing the outbound service request	42
2.5 The CICS resource definitions	43
2.5.1 URIMAP	43
2.5.2 PIPELINE	45
2.5.3 WEBSERVICE	48
2.5.4 The Web service binding file (WSBind).	50
2.5.5 SOAPFAULT commands	52
2.5.6 Mapping levels	53
2.5.7 Additional enhancements with CICS TS V3.2.	55
2.5.8 Additional enhancements with CICS TS 4.1	57
2.5.9 Use of WS-Addressing in CICS TS V4.1 applications	59
2.5.10 Comparing CICS TS V3.1 with later CICS TS versions	59
Chapter 3. Development approaches	61
3.1 Introduction	62
3.2 Bottom-up approach	63
3.3 Top-down approach	65
3.4 Meet-in-the-middle approach	66
3.5 The advantages of using RDz	68
3.6 Web services versus CICS TCP/IP connectivity	70
3.7 Conclusions.	71
Chapter 4. CICS catalog manager example application	73
4.1 Samples for use with CICS Web Services	74
4.2 Introduction to the catalog manager application	74
4.3 Installation and set up of the base application	75
4.3.1 Creating the VSAM data sets	76
4.3.2 Defining the base application to CICS	76
4.3.3 Configuring the example application.	77
4.3.4 Configuring code page support.	80
4.4 Web service support for the example application	81
4.4.1 The Web client front end	81
4.4.2 The CICS Web service client front end	83
4.4.3 Order dispatch Web services endpoints	84
4.4.4 Alternative Web service provider configuration.	84
4.5 Web services setup.	85
4.5.1 Creating the zFS directories	86

4.5.2	Creating the PIPELINE definition	86
4.5.3	Creating a TCPIP SERVICE	88
4.5.4	Dynamically installing WEBSERVICE and URIMAP resources	89
4.5.5	Creating the WEBSERVICE resources with RDO	92
4.5.6	Creating the URIMAP resources with RDO	93
4.5.7	Completing the installation	94
4.6	Installing the client application	94
4.6.1	FTP the client application	95
4.6.2	Install the client	95
4.6.3	Start the client	98
4.6.4	Testing the client	98
Chapter 5. Rational Developer for System z (RDz)		103
5.1	What is Rational Developer for System z?	104
5.2	RDz and CICS application development	104
5.3	Components of RDz	104
5.3.1	Workspace	104
5.3.2	Workbench	105
5.3.3	Perspective	106
5.3.4	View	108
5.3.5	Editor	109
5.3.6	Projects and subprojects	110
5.4	Writing your first COBOL Program with RD/z	111
5.4.1	Property groups	114
5.4.2	Compiler options	115
5.4.3	SQL options	116
5.4.4	CICS options	116
5.4.5	Property Group Manager view	119
5.4.6	Property Group editor	121
5.5	Writing your first Java program with RD/z	123
5.6	Overview of Debugging with RDz	126
5.6.1	Supported languages and environments	126
5.6.2	Local and remote debug	127
5.6.3	Basic debugging features and tools	127
5.7	Establishing Connection to remote Websphere Application Server	129
5.8	Import and Export EAR/WAR files	132
5.9	Summary	136
Chapter 6. Exposing the Catalog Sample CICS application as a Web service		137
6.1	Introduction	138
6.2	Install the provider mode resources	140
6.3	Create the provider mode deployment artifacts	141

6.3.1	Using the CICS Web Services Assistant	142
6.3.2	Use Rational Developer for System z	148
6.4	Testing the Web service	156
6.4.1	The Web Services Explorer	156
6.4.2	Generate a client.	160
6.5	Publishing WSDL to WebSphere Service Registry and Repository	164
6.5.1	Changes to DFHLS2WS for WebSphere Service Registry and Repository in CICS TS V4.1.	165
6.5.2	Changes to DFHWS2LS for WSRR in CICS TS V4.1.	167
6.5.3	New parameters to support SSL encryption in CICS TS V4.1	168
6.6	Writing applications that process the XML	169
6.6.1	Creating a custom application handler	169
6.6.2	Creating an XML-ONLY WEBSERVICE	170
Chapter 7. Create a CICS Web service requester application using the catalog sample		171
7.1	Introduction	172
7.2	Create a Web service requester using the CICS Web Services Assistant	174
7.2.1	Generate the required artifacts	174
7.2.2	Set up the CICS infrastructure	178
7.2.3	Test the requester application.	180
7.3	Creating and testing a Web service hosted in RDz.	183
7.3.1	Create a Web service skeleton with RDz	183
7.3.2	Implement the RDz based Web service	187
7.3.3	Test the Web service using RDz.	188
7.3.4	Test the Web service using the CICS sample application	192
Chapter 8. Componentization		195
8.1	CICS applications as components	196
8.2	Locally optimized Web services	197
8.3	Using WSDL to describe COBOL components	199
8.4	Further Options with CICS TS 4.1	199
8.4.1	Linking to a target PROGRAM from a requester mode PIPELINE .	200
8.4.2	Invoking a local SERVICE from a requester mode PIPELINE	201
Chapter 9. New SOA patterns for CICS TS V4.1		203
9.1	Service Component Architecture.	204
9.1.1	Introduction to SCA.	204
9.2	CICS TS V4.1: Implementation of SCA.	208
9.2.1	BUNDLE resources.	208
9.2.2	Creating services from existing CICS applications	208
9.2.3	Deploying SCA services	209
9.2.4	RDz SCA tooling.	210
9.2.5	Creating and deploying an SCA service from an existing CICS	

application.	210
Chapter 10. Hints and tips	213
10.1 Custom handlers programs for pipelines.	214
10.1.1 A simple example handler program	214
10.1.2 Handling state information	218
10.1.3 Propagating user identity tokens.	219
10.2 The SOAP fault API.	220
10.2.1 How to create a SOAP Fault in an application	220
10.2.2 Parsing SOAP Fault messages in CICS TS V4.1	221
10.3 Handling variably recurring XML elements	227
10.3.1 In-lined variably recurring data	227
10.3.2 Container based variably recurring data: inbound	229
10.3.3 Container based variably recurring data: outbound	232
10.4 Handling undefined XML (xsd:any)	233
10.5 Handling enumerated XML constructs	235
10.6 Modifying generated WSDL	237
10.6.1 Background to MTOM/XOP	237
10.6.2 Support for xsd:base64Binary and MTOM/XOP	238
10.6.3 Mapping a single field as binary data with DFHLS2WS	238
10.6.4 Handling variable length values and white space	240
10.7 WSDL types not supported by DFHWS2LS	245
10.8 Problem determination	247
10.8.1 Problems using DFHWS2LS and DFHLS2WS	247
10.8.2 Using the execution diagnostic facility to debug Web services	248
10.8.3 Debugging CICS SFR applications.	249
10.8.4 Runtime SOAP validation	251
10.9 XML parsing in CICS application.	252
10.9.1 XML Toolkit for z/OS.	253
10.9.2 COBOL High Speed XML parser	254
10.9.3 CICS API: EXEC CICS TRANSFORM	256
Chapter 11. COBOL samples	257
11.1 Introduction	258
11.2 Example 1: The <xsd:any> tag	258
11.2.1 The WSDL.	259
11.2.2 Web Services Assistant: z/OS	265
11.2.3 The COBOL program	265
11.2.4 CICS resource definitions	277
11.3 Example 2: The <choice> tag	278
11.3.1 The WSDL.	278
11.3.2 Generation of COBOL and CICS artifacts.	279
11.3.3 The COBOL program	279

11.3.4 CICS Resource Definitions	282
11.4 Example 3: minoccurs and maxoccurs	283
11.4.1 Generation of COBOL and CICS artifacts	284
11.4.2 The COBOL Program	284
11.4.3 CICS Resource Definitions	286
11.4.4 Results of calling the service.	286
Appendix A. Sample Web services	289
Preparation of your RDz environment	290
Loading an .ear file into a new or existing project	291
Description of examples A1–A3	293
The XML any passthrough Web service example.	293
The XML choice Web service example.	297
The XML occurs Web service example.	301
Appendix B. Sample COBOL programs	307
Program to call <xsd:any> example service.	308
WSDL - <xsd:any>	314
Request Language Structure - inlinI01	318
Response Language Structure - inlinO01	321
Program to call <xsd:choice> example service	324
WSDL <xsd:choice>.	330
Request Language Structure - choicI01	334
Response Language Structure - choicO01	336
Program to call minOccurs/maxOccurs example service.	338
WSDL - minOccurs/maxOccurs	344
Request Language Structure - redboI01	348
Response Language Structure - redboO01	350
Related publications	353
IBM Redbooks	353
Other publications	353
Online resources	353
How to get Redbooks	354
Help from IBM	354
Appendix C. Additional material	355
Locating the Web material	355
How to use the Web material	355
Index	359

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.

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 illustrate 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.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

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

CICS Explorer™
CICSplex®
CICS®
DataPower®
DB2®
IBM®

IMS™
Language Environment®
OMEGAMON®
Rational®
Redbooks®
Redbooks (logo) ®

System z®
Tivoli®
WebSphere®
z/OS®
zSeries®

The following terms are trademarks of other companies:

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

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

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

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

Preface

This IBM® Redbooks® publication focuses on developing Web service applications in CICS®. It takes the broad view of developing and modernizing CICS applications for XML, Web services, SOAP, and SOA support, and lays out a reference architecture for developing these kinds of applications.

We start by discussing Web services in general, then review how CICS implements Web services. We offer an overview of different development approaches: bottom-up, top-down, and meet-in-the-middle. After laying out the foundations, we review the CICS catalog manager sample application, as this is the application we used as a basis.

We then look at how you would go about exposing a CICS application (namely, the catalog manager sample application) as a Web service provider, again looking at the different approaches. The book then steps through the process of creating a CICS Web service requester.

We close out by looking at CICS application aggregation (including 3270 applications) with Rational® Application Developer for System z®. The final chapter offers hints and tips to help you when implementing this technology.

The team who wrote this book

This book was produced by a team of specialists from around the world working at the China Development Lab, Beijing.

Chris Rayns is an IT Specialist and the CICS project leader at the International Technical Support Organization, Poughkeepsie Center. He writes extensively on all areas of CICS. Before joining the ITSO, Chris worked in IBM Global Services in the United Kingdom as a CICS IT Specialist.

George Burgess is currently working as the CICS Transaction Server on z/OS® Subject Matter Expert for the Peoples Republic of China and is based in Beijing. He has 24 years of experience as an Application Programmer, Systems Programmer, CICS Developer and OMEGAMON® XE for CICS Developer. His areas of expertise include Common Business Oriented Language (COBOL), CICS, DB2®, WebSphere® MQ, IMS™ DL/1, VSAM, JCL,z/OS, and OMEGAMON.

Paul Cooper has been a member of the CICS development team in IBM Hursley for over 10 years. He has spent most of that time working on the Java™ support in CICS and the Web services support in CICS. He is often involved in customer support activities around CICS Web services

Tony Fitzgerald is a Software Support Specialist in the UK. He has worked for IBM as CICS Level 2 Support for five years and has 15 years of experience working with CICS and DB2 as a Systems Programmer and an Applications Programmer. He holds a degree in Computer Science and Management Science from the University of Keele.

Ankur Goyal is an IT Specialist with IBM with the Rational Software Technical Sales team in India. He consults and supports customer solutions on IBM middleware, open source, open standards, and emerging technologies. Prior to this role he was part of the IBM Academic Initiative-Ecosystem group, where his team was responsible for building an ecosystem around IBM middleware in the emerging market of India. His areas of interest include software development best practices, Eclipse, and integrating Web technologies with the help of open standards. He joined IBM in 2004 after gaining expertise in application development as a Software Engineer. He holds a Bachelor's degree in information technology from National Institute of Technology, Durgapur, and is also IBM certified for J2EE, DB2, WebSphere, Rational, XML, and SOA.

Peter Klein is a CICS Team Leader at the IBM Germany Customer Support Center. He has 18 years of experience working as a Technical Support Specialist with IBM software products. His expertise includes WebSphere MQ, CICSplex® System Manager, and distributed transaction systems. Peter has contributed to several other IBM Redbooks publications and ITSO projects sponsored by IBM Learning Services.

Guo Qiang Li is a Software Engineer with the China CICS Team, which is the first team in CDL working on CICS Transaction Server. He focuses on CICS Dynamic LIBRARY management testing and CICS Web Service support testing. His experiences on CICS include CPSM and Web services. He graduated from Tianjin University with a Master's degree and joined the China CICS Team in 2006.

SanYong Liu is an Advisory IT Specialist with Technical Sales Support for the IBM software products in China. His expertise includes CICS Transaction Server, WebSphere Application Server, WebSphere MQ, and WebSphere software portfolio for SOA. He has several years of customer experience for consulting and supporting large banks in China with IBM mainframe solutions. He holds a Bachelor's degree in Software Engineering from XiDian University.

Yan Sun is a CICS Team Leader at Data Center(Beijing), Industrial and Commercial Bank of China. She has seven years of experience working as a Systems Programmer. Her areas of expertise include CICS Transaction Server, WebSphere MQ, and Encryption Facility for z/OS. She holds a Master's degree in Information Management from Beijing Jiaotong University.

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

Richard Conway
International Technical Support Organization, Poughkeepsie Center

Mark Pocock, CICS 390 Change Team
IBM Hursley

Thanks to the authors of the previous editions of this book.

► Authors of the first edition, *Application Development for CICS Web Services*, published in May 2006, were:

Isabel Arnold, Chris Backhouse, Leigh Compton, David Evans, Jim Hollingsworth, William Yates

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will 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 books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

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

ibm.com/redbooks

- ▶ Send your comments in an e-mail to:

redbooks@us.ibm.com

- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition might also include minor corrections and editorial changes that are not identified.

January 2010, Second Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- ▶ Chapter 5 Rational Developer for System z (RD/z)
- ▶ Chapter 9 Componenization
- ▶ Chapter 10 SCA
- ▶ Chapter 12 COBOL Samples
- ▶ Appendix A Sample Web services
- ▶ Appendix B Sample COBOL programs

Changed information

- ▶ All chapters have been updated to reflect the changes in CICS TS 4.1



Overview of Web services

In this chapter, we focus on some of the architectural concepts that have to be considered in a Web services project. We define and discuss service-oriented architecture (SOA) and the relationship between SOAs and Web services.

We then take a closer look at Web services, a technology that enables you to invoke applications using Internet protocols and standards. The technology is called Web services because it integrates services (applications) using Web technologies (the Internet and its standards).

1.1 Introduction

There is a strong trend for companies to integrate existing systems to implement IT support for business processes that cover the entire business cycle. Today, interactions already exist using a variety of schemes that range from rigid point-to-point electronic data interchange (EDI) interactions to open Web auctions. Many companies have already made some of their IT systems available to all of their divisions and departments, or even their customers or partners on the Web. However, techniques for collaboration vary from one case to another and are thus proprietary solutions. Systems often collaborate without any vision or architecture.

Thus, there is an increasing demand for technologies that support the connecting or sharing of resources and data in a flexible and standardized manner. Because technologies and implementations vary across companies and even within divisions or departments, unified business processes cannot be smoothly supported by technology. Integration has been developed only between units that are already aware of each other and that use the same static applications.

Furthermore, there is a requirement to further structure large applications into building blocks to use well-defined components within different business processes. A shift towards a service-oriented approach (SOA) not only can standardize interaction, but also allows for more flexibility in the process. The complete value chain within a company is divided into small modular functional units, or services. A SOA thus has to focus on how services are described and organized to support their dynamic, automated discovery and use.

Companies and their sub-units should be able to provide services easily. Other business units can use these services to implement their business processes. This integration can ideally be performed during the runtime of the system, not just at the design time.

1.2 Service-oriented architecture

This section is a short introduction to the key concepts of a SOA. The architecture makes no statements about the infrastructure or protocols it uses. Therefore, you can implement a SOA using technologies other than Web technologies.

As shown in Figure 1-1, a SOA contains three basic components:

- ▶ Service provider

The service provider creates a Web service and possibly publishes to the service broker the information necessary to access and interface with the Web service.

- ▶ Service broker

The service broker (also known as a service registry) makes the Web service access and interface information available to any potential service requester.

- ▶ Service requester

The service requester binds to the service provider to invoke one of its Web services, having optionally placed entries in the broker registry using various find operations.

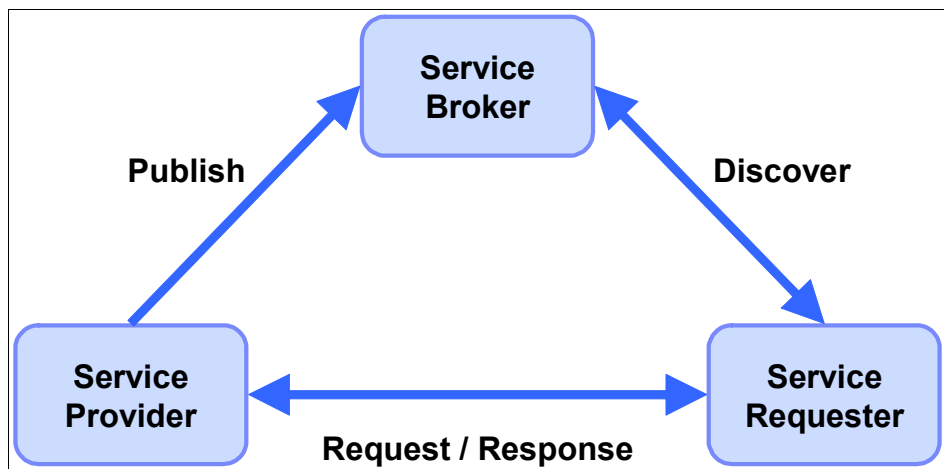


Figure 1-1 Web services components and operations

Each component can also act as one of the two other components. For example, if a service provider requires information that it can only acquire from some other service, it acts as a service requester while still serving the original request.

- ▶ The service provider creates a Web service and possibly publishes its interface and access information to the service broker.
- ▶ The service broker (also known as service registry) is responsible for making the Web service interface and implementation access information available to any potential service requester.
- ▶ The service requester binds to the service provider to invoke one of its Web services, having optionally placed entries in the broker registry using various find operations.

1.2.1 Characteristics

The SOA uses a loose coupling between the participants. Such a loose coupling provides greater flexibility as follows:

- ▶ Old and new functional blocks are encapsulated into components that work as services.
- ▶ Functional components and their interfaces are separated. Therefore, new interfaces can be plugged in more easily.
- ▶ Within complex applications, the control of business processes can be isolated. A business rule engine can be incorporated to control the workflow of a defined business process. Depending on the state of the workflow, the engine calls the respective services.

1.2.2 Web services versus SOAs

SOAs have been used under various guises for many years. It can be, and has been, implemented using a number of different distributed computing technologies, such as CORBA or messaging middleware. The effectiveness of SOAs in the past has always been limited by the ability of the underlying technology to interoperate across the enterprise.

Web services technology is an ideal technology choice for implementing a SOA:

- ▶ Web services are standards-based. Interoperability is a key business advantage within the enterprise and is crucial in B2B scenarios.
- ▶ Web services are widely supported across the industry. For the first time, all major vendors are recognizing and providing support for Web services.
- ▶ Web services are platform and language neutral. There is no bias for or against a particular hardware or software platform. Web services can be implemented in any programming language or toolset. This is important because continued industry support exists for the development of standards and interoperability between vendor implementations.
- ▶ This technology provides a migration path to enable existing business functions gradually, as Web services are required.
- ▶ This technology supports synchronous and asynchronous, RPC-based, and complex message-oriented exchange patterns.

Conversely, there are many Web services implementations that are not a SOA. For example, the use of Web services to connect two heterogeneous systems directly together is not a SOA. These uses of Web services solve real problems and provide significant value on their own. They can form the starting point of a SOA.

In general, a SOA has to be implemented at an enterprise or organizational level to harvest many of the benefits.

For more information about the relationship between Web services and SOAs, or the application of IBM Patterns for e-business to a Web services project, refer to *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303.

1.3 Web services

Web services perform encapsulated business functions, ranging from simple request-reply to full business process interactions. These services can be new applications or just wrapped around existing business functions to make them network-enabled. Services can rely on other services to achieve their goals.

It is important to note from this definition that a Web service is not constrained to using SOAP over HTTP/S as the transport mechanism. Web services are equally at home in the messaging world.

1.3.1 Properties of a Web service

All Web services share the following properties:

- ▶ Self-contained
On the client side, no additional software is required. A programming language with XML and HTTP client support is enough to get you started. On the server side, an HTTP server and a SOAP server are required.
- ▶ Self-describing
Using Web Services Description Language (WSDL), all the information required to implement a Web service as a provider, or to invoke a Web service as a requester, is provided.
- ▶ Published, located, and invoked across the Web
This technology uses established lightweight Internet standards such as HTTP. It makes use of the existing infrastructure.
- ▶ Modular
Simple Web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer Web services from a Web service implementation. Web services can be chained together to perform higher-level business functions. This shortens development time and enables best-of-breed implementations.

- ▶ Language-independent and interoperable
The client and server can be implemented in different environments. Theoretically, any language can be used to implement Web service clients and servers.
- ▶ Inherently open and standard-based
XML and HTTP are the major technical foundations for Web services. A large part of the Web service technology has been built using open-source projects. Vendor independence and interoperability are realistic goals.
- ▶ Loosely coupled
Traditionally, application design has depended on tight interconnections at both ends. Web services require a simpler level of coordination that allows a more flexible reconfiguration for an integration of the services in question.
- ▶ Programmatic access
The approach provides no graphical user interface. It operates at the code level. Service consumers have to know the interfaces to Web services, but do not have to know the implementation details of services.
- ▶ Wraps existing applications
Already existing stand-alone applications can easily be integrated into the SOA by implementing a Web service as an interface.

1.3.2 Core standards

Web services are built upon four core standards, as explained in the following sections.

Extensible Markup Language (XML)

XML is the foundation of Web services. However, because much information has already been written about XML, we do not describe it in this document. You can find information about XML at the following Web page:

<http://www.w3.org/XML/>

SOAP

Originally proposed by Microsoft®, SOAP was designed to be a simple and extensible specification for the exchange of structured, XML-based information in a decentralized, distributed environment. As such, it represents the main means of communication between the three actors in an SOA:

- ▶ Service provider
- ▶ Service requester
- ▶ Service broker

A group of companies, including IBM, submitted SOAP to the W3C for consideration by its XML Protocol Working Group. There are currently two versions of SOAP: Version 1.1 and Version 1.2.

The SOAP 1.1 specification contains three parts:

- ▶ An envelope that defines a framework for describing message content and processing instructions. Each SOAP message consists of an envelope that contains an arbitrary number of headers and one body that carries the payload. SOAP messages might contain faults. Faults report failures or unexpected conditions.
- ▶ A set of encoding rules for expressing instances of application-defined data types.
- ▶ A convention for representing remote procedure calls and responses.

A SOAP message is, in principle, independent of the transport protocol that is used, and can, therefore, potentially be used with a variety of protocols, such as HTTP, JMS, SMTP, or FTP. Right now, the most common way of exchanging SOAP messages is through HTTP.

The way SOAP applications communicate when exchanging messages is often referred to as the message exchange pattern (MEP). The communication can be either one-way messaging, where the SOAP message only goes in one direction, or two-way messaging, where the receiver is expected to send back a reply.

Due to the characteristics of SOAP, it does not matter what technology is used to implement the client, as long as the client can issue XML messages. Similarly, the service can be implemented in any language, as long as it can process XML messages.

Note: The authors of the SOAP 1.1 specification declared that the acronym SOAP stands for Simple Object Access Protocol. The authors of the SOAP 1.2 specification decided not to give any meaning to the acronym SOAP.

Web Services Description Language (WSDL)

This standard describes Web services as abstract service end points that operate on messages. Both the operations and the messages are defined in an abstract manner, while the actual protocol used to carry the message and the end point's address are concrete.

WSDL is not bound to any particular protocol or network service. It can be extended to support many different message formats and network protocols. However, because Web services are mainly implemented using SOAP and HTTP, the corresponding bindings are part of this standard.

The WSDL 1.1 specification only defines bindings that describe how to use WSDL in conjunction with SOAP 1.1, HTTP GET and POST, and MIME. The specification for WSDL 1.1 can be found at the following Web page:

<http://www.w3.org/TR/wsd1>

WSDL 2.0 provides a model as well as an XML format for describing Web services. It enables you to separate the description of the abstract functionality offered by a service from the concrete details of a service description. It also describes extensions for MEPs, SOAP modules, and a language for describing such concrete details for SOAP1.2 and HTTP.

There are eight MEPs defined. CICS TS V4.1 supports four of them:

▶ In-Only

A request message is sent to the Web service provider, but the provider is not allowed to send any type of response to the Web service requester.

▶ In-Out

A request message is sent to the Web service provider, and a response message is returned. The response message can be a normal SOAP message or a SOAP fault.

▶ In-Optional-Out

A request message is sent to the Web service provider, and a response message is optionally returned to the requester. The response message can be a normal SOAP message or a SOAP fault.

▶ Robust-In-Only

A request message is sent to the Web service provider, and no response message is returned to the requester unless an error occurs. In this case, a SOAP fault message is sent to the requester.

The four MEPs that CICS TS V4.1 does not support are:

▶ Out-Only

▶ Robust-Out-Only

▶ Out-In

▶ Out-Optional-In

The specification for WSDL 2.0 can be found at the following Web page:

<http://www.w3.org/TR/wsd120>

Universal Description, Discovery, and Integration standard

The Universal Description, Discovery, and Integration (UDDI) standard defines a means to publish and to discover Web services. As of this writing, UDDI Version 3.0 has been finalized, but UDDI Version 2.0 is still more commonly used. For more information, refer to the following Web pages:

<http://www.uddi.org/>

<http://www.oasis-open.org/specs/index.php#wssv1.0>

1.3.3 Web Services Interoperability group

Web services can be used to connect computer systems together across organizational boundaries. Therefore, a set of open non-proprietary standards that all Web services adhere to maximizes the ability to connect disparate systems together.

The Web Service Interoperability (WS-I) group is an organization that promotes open interoperability between Web services regardless of platform, operating systems, and programming languages. To promote this cause, the WS-I group has released a basic profile that outlines a set of specifications to which WSDL documents and Web services traffic (SOAP over HTTP transport) must adhere to be WS-I compliant. The full list of specifications can be found at the WS-I Web site:

<http://www.ws-i.org/>

IBM is a member of the WS-I community, and CICS support for Web services is fully compliant with the WS-I basic profile 1.0.

1.3.4 Additional standards

There are other Web services specifications that are now supported by CICS. For a list of the limitations of CICS support, refer to *CICS Web Services Guide*, SC34-6838.

Web Services Atomic Transaction

This specification, commonly known as WS-Atomic Transaction, defines the atomic transaction coordination type for transactions of a short duration. Together with the Web Services Coordination specification, it defines protocols for short-term transactions that enable transaction processing systems to interoperate in a Web services environment. Transactions that use WS-Atomic Transaction have the properties of atomicity, consistency, isolation, and durability (ACID).

Web Services Security: SOAP Message Security

This specification is a set of enhancements to SOAP messaging that provides message integrity and confidentiality. The specification provides three main mechanisms that can be used independently or together:

- ▶ The ability to send security tokens as part of a message, and for associating the security tokens with message content
- ▶ The ability to protect the contents of a message from unauthorized and undetected modification (message integrity)
- ▶ The ability to protect the contents of a message from unauthorized disclosure (message confidentiality)

Web Services Trust Language

This specification, commonly known as WS-Trust, defines extensions that build on Web Services Security to provide a framework for requesting and issuing security tokens, and broker trust relationships.

SOAP Message Transmission Optimization Mechanism (MTOM)

This specification is one of a related pair of specifications that define how to optimize the transmission and format of a SOAP message. MTOM defines:

- ▶ How to optimize the transmission of base64 binary data in SOAP messages.
- ▶ How to implement optimized MIME multipart serialization of SOAP messages in a binding, independent way.
- ▶ The implementation of MTOM relies on the related XML-binary Optimized Packaging (XOP) specification. As these two specifications are so closely linked, they are normally referred to as MTOM/XOP.

Web Services Addressing

This specification, usually referred to as WS-Addressing, provides a standard way to address Web services and to provide addressing information in SOAP messages. The WS-Addressing specification introduces two primary concepts: endpoint references, and message addressing properties.

- ▶ Endpoint reference

This is a way to encapsulate information about specific Web service endpoints. Endpoint references can be propagated to other parties and then used to target the Web service endpoint that they represent.

- ▶ Message addressing properties

MAP are a set of defined WS-Addressing properties that can be represented as elements in SOAP headers and provide a standard way of conveying information, such as the endpoint to which message replies should be directed, or information about the relationship that the message has with other messages.

1.4 IBM WebSphere Service Registry and Repository

IBM provides an enterprise strength solution that enables governance of SOA artifacts, most of which are related to Web services. The IBM WebSphere Service Registry and Repository product is such a solution.

The product provides an integrated service metadata repository to govern services and manage the service life cycle, promoting visibility and consistency, and reducing redundancy in your organization. You can seamlessly publish and find capabilities across all phases of SOA, enriching connectivity with dynamic and efficient interactions between services at runtime.

1.5 SOAP

In this section we focus mainly on SOAP 1.1.

1.5.1 The envelope

A SOAP message is an envelope containing zero or more headers and one body:

- ▶ The envelope is the root element of the XML document, providing a container for control information, the addressee of a message, and the message itself.
- ▶ Headers contain control information, such as quality of service attributes.
- ▶ The body contains the message identification and its parameters.
- ▶ Both the headers and the body are child elements of the envelope element.

Figure 1-2 shows a simple SOAP request message.

- ▶ The header tells who must deal with the message and how to deal with it. When the actor is next or when the actor is omitted, the receiver of the message must do what the body says. Furthermore, the receiver must understand and process the application-defined <TranID> element.
- ▶ The body tells what has to be done: Dispatch an order for quantityRequired 1 of itemRefNumber 0010 to customerID CB1 in chargeDepartment ITSO.

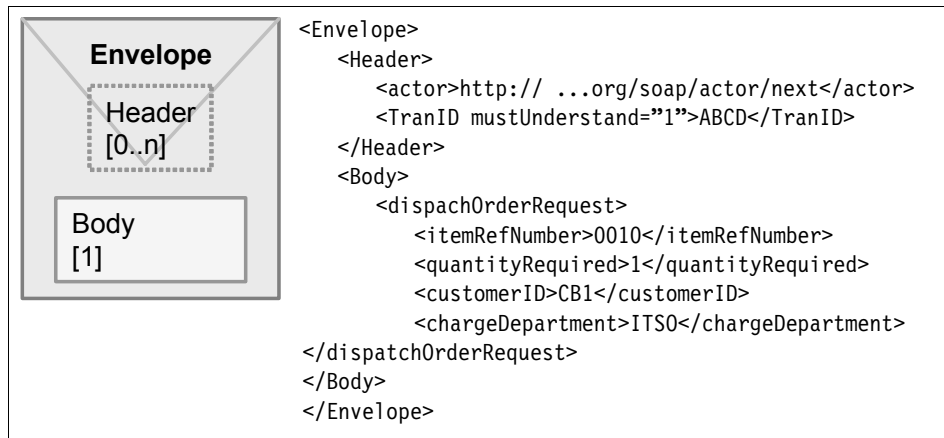


Figure 1-2 Example of a simple SOAP message

Namespaces

Namespaces play an important role in SOAP messages. A namespace is simply a way of adding a qualifier to an element name to ensure that it is unique.

For example, we might have a message that contains an element <customer>. Customers are fairly common, so it is likely that many Web services have customer elements. To ensure that we know what customer we are talking about, we declare a namespace for it, for example, as follows:

```
xmlns:itso="http://itso.ibm.com/CICS/catalogApplication"
```

This identifies the prefix itso with the declared namespace. Then, whenever we reference the element <customer> we prefix it with the namespace as follows: <itso:customer>. This identifies it uniquely as a customer type for our application. Namespaces can be defined as any unique string. They are often defined as URLs because URLs are generally globally unique, and they have to be in URL format. These URLs do not have to physically exist though.

The WS-I Basic Profile 1.0 requires that all application-specific elements in the body must be namespace-qualified to avoid collisions between names.

Table 1-1 shows the namespaces of SOAP and WS-I Basic Profile 1.0 used in this book.

Table 1-1 SOAP namespaces

Namespace URI	Explanation
http://schemas.xmlsoap.org/soap/envelope/	SOAP 1.1 envelope namespace
http://schemas.xmlsoap.org/soap/encoding/	SOAP 1.1 encoding namespace
http://www.w3.org/2001/XMLSchema-instance	Schema instance namespace
http://www.w3.org/2001/XMLSchema	XML Schema namespace
http://schemas.xmlsoap.org/wsdl	WSDL namespace for WSDL framework
http://schemas.xmlsoap.org/wsdl/soap	WSDL namespace for WSDL SOAP binding
http://ws-i.org/schemas/conformanceClaim/	WS-I Basic Profile

SOAP envelope

The Envelope is the root element of the XML document representing the message. It has the following structure:

```
<SOAP-ENV:Envelope .... >
  <SOAP-ENV:Header>
    <SOAP-ENV:HeaderEntry.... />
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    [message payload]
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In general, a SOAP message is a (possibly empty) set of headers plus one body. The SOAP envelope also defines the namespace for structuring messages. The entire SOAP message (headers and body) is wrapped in this envelope.

Headers

Headers are a generic and flexible mechanism for extending a SOAP message in a decentralized and modular way without prior agreement between the parties involved. They allow control information to pass to the receiving SOAP server and also provide extensibility for message structures.

Headers are optional elements in the envelope. If present, the Header element must be the first immediate child element of a SOAP Envelope element. All immediate child elements of the Header element are called *header entries*.

There is a predefined header attribute called SOAP-ENV:mustUnderstand. The value of the mustUnderstand attribute is either 1 or 0. The absence of the SOAP mustUnderstand attribute is semantically equivalent to the value 0.

If the mustUnderstand attribute is present in a header entry and set to 1, the service provider must implement the semantics defined by the element:

```
<Header>
  <thens:TranID mustUnderstand="1">ABCD</thens:TranID>
</Header>
```

In the example, the header entry specifies that a service invocation must fail if the service provider does not support the ability to process the TranID header.

A SOAP intermediary is an application that is capable of both receiving and forwarding SOAP messages on their way to the final destination. In realistic situations, not all parts of a SOAP message might be intended for the ultimate destination of the SOAP message, but, instead, might be intended for one or more of the intermediaries on the message path. Therefore, a second predefined header attribute, SOAP-ENV:actor, is used to identify the recipient of the header information. In SOAP 1.2 the actor attribute is renamed SOAP-ENV:role. The value of the SOAP actor attribute is the URI of the mediator, which is also the final destination of the particular header element (the mediator does not forward the header).

If the actor is omitted or set to the predefined default value, the header is for the actual recipient and the actual recipient is also the final destination of the message (body). The predefined value is:

```
http://schemas.xmlsoap.org/soap/actor/next
```

If a node on the message path does not recognize a mustUnderstand header and the node plays the role specified by the actor attribute, the node must generate a SOAP MustUnderstand fault. Whether the fault is sent back to the sender depends on the message exchange pattern in use. For request/response, the WS-I BP 1.0 requires the fault to be sent back to the sender. Also, according to WS-I BP 1.0, the receiver node must discontinue normal processing of the SOAP message after generating the fault message.

Headers can carry authentication data, digital signatures, encryption information, and transactional settings. They can also carry client-specific or project-specific controls and extensions to the protocol. The definition of headers is not just up to standards bodies.

Note: The header must not include service instructions (that would be used by the service implementation).

Body

The SOAP Body element provides a mechanism for exchanging information intended for the ultimate recipient of the message. The Body element is encoded as an immediate child element of the SOAP Envelope element. If a Header element is present, the Body element must immediately follow the Header element. Otherwise, it must be the first immediate child element of the Envelope element.

All immediate child elements of the Body element are called *body entries*. Each body entry is encoded as an independent element within the SOAP Body element. In the most simple case, the body of a basic SOAP message consists of an XML message as defined by the schema in the types section of the WSDL document. It is legal to have any valid XML as the body of the SOAP message, but WS-I conformance requires that the elements be namespace qualified.

Error handling

One area where there are significant differences between the SOAP 1.1 and 1.2 specifications is in the handling of errors. Here we focus on the SOAP 1.1 specification for error handling.

SOAP itself predefines one body element, the fault element, which is used for reporting errors. If present, the fault element must appear as a body entry and must not appear more than once. The children of the fault element are defined as follows:

- ▶ `faultcode` is a code that indicates the type of the fault. SOAP defines a small set of SOAP fault codes covering basic SOAP faults:
 - `soapenv:Client`
This code indicates that the client sent an incorrectly formatted message
 - `soapenv:Server`,
This code is for delivery problems
 - `soapenv:VersionMismatch`,
This code can report any invalid namespaces specified on the Envelope element
 - `soapenv:MustUnderstand`
This code is for errors regarding the processing of header content
- ▶ `faultstring` is a human-readable description of the fault. It must be present in a fault element.

- ▶ `faultactor` is an optional field that indicates the URI of the source of the fault. The value of the `faultactor` attribute is a URI identifying the source that caused the error. Applications that do not act as the ultimate destination of the SOAP message must include the `faultactor` element in a SOAP fault element.
- ▶ `detail` is an application-specific field that contains detailed information about the fault. It must not be used to carry information about errors belonging to header entries. The absence of the `detail` element in the fault element indicates that the fault is not related to the processing of the body element (the actual message).

For example, a `soapenv:Server` fault message is returned if the service implementation throws a SOAP exception. The exception text is transmitted in the `faultstring` field.

Although SOAP 1.1 permits the use of custom-defined `faultcodes`, the WS-I Basic Profile only permits the use of the four codes defined in SOAP 1.1.

1.5.2 Communication styles

SOAP supports two different communication styles:

- ▶ Document

Also known as *message-oriented style*, this is a flexible communication style that provides the best interoperability. The message body is any legal XML as defined in the `types` section of the WSDL document.

- ▶ Remote procedure call (RPC)

The remote procedure call is a synchronous invocation of an operation which returns a result; it is conceptually similar to other RPCs.

1.5.3 Encodings

In distributed computing environments, encodings define how data values defined in the application can be translated to and from a protocol format. We refer to these translation steps as *serialization* and *deserialization*, or, synonymously, *marshalling* and *unmarshalling*.

When implementing a Web service, we have to choose one of the tools and programming or scripting languages that support the Web services model. However, the protocol format for Web services is XML, which is independent of the programming language. Thus, 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.

The following encodings are defined:

- ▶ SOAP encoding

SOAP encoding enables marshalling/unmarshalling of values of data types from the SOAP data model. This encoding is defined in the SOAP 1.1 standard.

- ▶ Literal

The literal encoding is a simple XML message that does not carry encoding information. Usually, an XML Schema describes the format and data types of the XML message.

1.5.4 Messaging modes

The two styles (RPC and document) and the two common encodings (SOAP encoding and literal) can be freely intermixed to produce what is called a SOAP messaging mode. Although SOAP supports four modes, only three of the four modes are generally used, and further, only two are preferred by the WS-I Basic Profile.

- ▶ Document/literal

Provides the best interoperability between language environments. The WS-I Basic Profile states that all Web service interactions should use the Document/literal mode.

- ▶ RPC/literal

Possible choice between certain implementations. Although RPC/literal is WS-I compliant, it is not frequently used in practice. There are a number of usability issues associated with RPC/literal.

- ▶ RPC/encoded

Early Java implementations supported this combination, but it does not provide interoperability with other implementations and is not recommended

- ▶ Document/encoded

Not used in practice.

You can find the SOAP 1.1 specification at the following Web page:

<http://www.w3.org/TR/2000/NOTE-SOAP-20000508>

The SOAP 1.2 specification is at the following Web page:

<http://www.w3.org/TR/SOAP12>

1.6 WSDL

This section introduces Web Services Description Language (WSDL) 1.1. WSDL uses XML to specify the characteristics of a Web service: what the Web service can do, where it resides, and how it is invoked. WSDL can be extended to allow descriptions of different bindings, regardless of what message formats or network protocols are used to communicate. WSDL enables 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 operations of the Web service
- ▶ Type information—Operations, parameters, and data types comprising the interface of the Web service, plus a name for this interface

1.6.1 WSDL Document

A WSDL document contains the following main elements:

- ▶ Types
This element is a container for data type definitions using some type system, usually XML Schema.
- ▶ Message
This element is an abstract, typed definition of the data being communicated. A message can have one or more typed parts.
- ▶ Port type
This element is an abstract set of one or more operations supported by one or more ports.
- ▶ Operation
This element is an abstract description of an action supported by the service that defines the input and output message and optional fault message.
- ▶ Binding
This element is 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
This element is a single endpoint, which is defined as an aggregation of a binding and a network address.
- ▶ Service
This element is a collection of related ports.

WSDL does not introduce a new type definition language. WSDL recognizes the requirement for rich type systems for describing message formats and supports the XML Schema Definition (XSD) specification.

WSDL 1.1 introduces specific binding extensions for various protocols and message formats. There is a WSDL SOAP binding, which is capable of describing SOAP over HTTP. WSDL does not define any mappings to a programming language. Rather, the bindings deal with transport protocols. This is a major difference from interface description languages, such as the CORBA Interface Definition Language (IDL), which has language bindings.

You can find the WSDL 1.1 specification at the following Web page:

<http://www.w3.org/TR/wsdl>

1.6.2 WSDL document anatomy

Figure 1-3 shows the elements comprising a WSDL document and the various relationships between them.

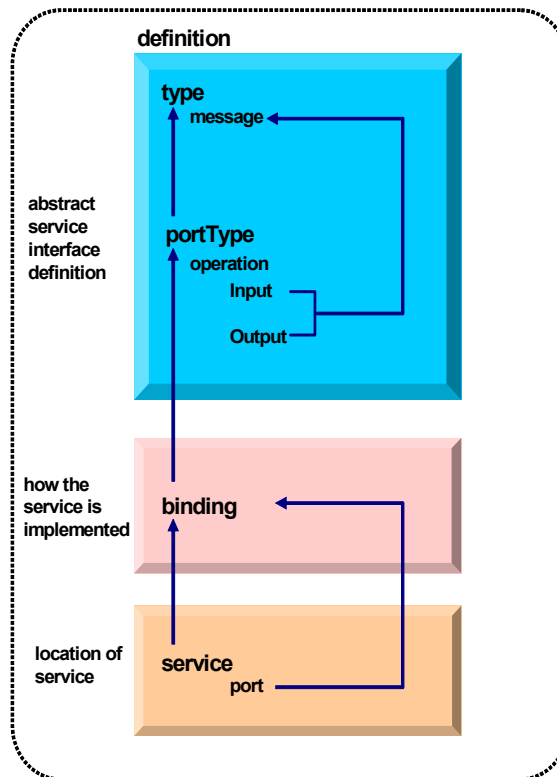


Figure 1-3 WSDL elements and relationships

The diagram should be interpreted in the following way:

- ▶ One WSDL document contains zero or more services. A service contains zero or more port definitions (service endpoints), and a port definition contains a specific protocol extension.
- ▶ The same WSDL document contains zero or more bindings. A binding is referenced by zero or more ports. The binding contains one protocol extension, where the style and transport are defined, and zero or more operations bindings. Each of these operation bindings is composed of one protocol extension, where the action and style are defined, and one to three messages bindings, where the encoding is defined.
- ▶ The same WSDL document contains zero or more port types. A port type is referenced by zero or more bindings. This port type contains zero or more operations, which are referenced by zero or more operations bindings.
- ▶ The same WSDL document contains zero or more messages. An operation usually points to an input and an output message, and optionally to some faults. A message is composed of zero or more parts.
- ▶ The same WSDL document contains zero or more types. A type can be referenced by zero or more parts.
- ▶ The same WSDL document points to zero or more XML schemas. An XML schema contains zero or more XSD types that define the different data types.

Example

Example 1-1 is an example of a simple, complete, and valid WSDL file. As Example 1-1 shows, even a simple WSDL document contains quite a few elements with various relationships to each other. Example 1-1 contains the WSDL file example. This example is analyzed in detail later in this section.

Example 1-1 Complete WSDL document

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com"
  xmlns:resns="http://www.exampleApp.dispatchOrder.Response.com"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.exampleApp.dispatchOrder.com"
  targetNamespace="http://www.exampleApp.dispatchOrder.com">
  <types>
    <xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Request.com"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      attributeFormDefault="qualified"
      elementFormDefault="qualified"
      targetNamespace="http://www.exampleApp.dispatchOrder.Request.com"
      xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com">
      <xsd:element name="dispatchOrderRequest" nillable="false">
```

```

<xsd:complexType mixed="false">
  <xsd:sequence>
    <xsd:element name="itemReferenceNumber" nillable="false">
      <xsd:simpleType>
        <xsd:restriction base="xsd:short">
          <xsd:maxInclusive value="9999"/>
          <xsd:minInclusive value="0"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="quantityRequired" nillable="false">
      <xsd:simpleType>
        <xsd:restriction base="xsd:short">
          <xsd:maxInclusive value="999"/>
          <xsd:minInclusive value="0"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
<xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Response.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  attributeFormDefault="qualified"
  elementFormDefault="qualified"
  targetNamespace="http://www.exampleApp.dispatchOrder.Response.com">
  <xsd:element name="dispatchOrderResponse" nillable="false">
    <xsd:complexType mixed="false">
      <xsd:sequence>
        <xsd:element name="confirmation" nillable="false">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="20"/>
              <xsd:whiteSpace value="preserve"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
</types>
<message name="dispatchOrderResponse">
  <part element="resns:dispatchOrderResponse" name="ResponsePart"/>
</message>
<message name="dispatchOrderRequest">
  <part element="reqns:dispatchOrderRequest" name="RequestPart"/>
</message>

```

```

<portType name="dispatchOrderPort">
  <operation name="dispatchOrder">
    <input message="tns:dispatchOrderRequest" name="DFHOXODSRequest"/>
    <output message="tns:dispatchOrderResponse" name="DFHOXODSResponse"/>
  </operation>
</portType>
<binding name="dispatchOrderSoapBinding" type="tns:dispatchOrderPort">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="dispatchOrder">
    <soap:operation soapAction="" style="document"/>
    <input name="DFHOXODSRequest">
      <soap:body parts="RequestPart" use="literal"/>
    </input>
    <output name="DFHOXODSResponse">
      <soap:body parts="ResponsePart" use="literal"/>
    </output>
  </operation>
</binding>
<service name="dispatchOrderService">
  <port binding="tns:dispatchOrderSoapBinding" name="dispatchOrderPort">
    <soap:address
      location="http://myserver:54321/exampleApp/services/dispatchOrderPort"/>
  </port>
</service>
</definitions>

```

Namespaces

WSDL uses the XML namespaces listed in Table 1-2.

Table 1-2 WSDL namespaces

Namespace URI	Explanation
http://schemas.xmlsoap.org/wsdl/	Namespace for WSDL framework.
http://schemas.xmlsoap.org/wsdl/soap/	SOAP binding.
http://schemas.xmlsoap.org/wsdl/http/	HTTP binding.
http://www.w3.org/2000/10/XMLSchema	Schema namespace as defined by XSD.
(URL to WSDL file)	The <i>this namespace</i> (tns) prefix is used as a convention to refer to the current document. Do not confuse it with the XSD <i>target namespace</i> , which is a different concept.

The first three namespaces are defined by the WSDL specification itself. The next definition references a namespace that is defined in the SOAP and XSD standards. The last one is local to each specification.

1.6.3 WSDL definition

The WSDL definition contains types, messages, operations, port types, bindings, ports, and services.

Also, WSDL provides an optional element called `wsdl:document` as a container for human-readable documentation.

Types

The `types` element encloses data type definitions used by the exchanged messages. WSDL uses XML Schema Definition (XSD) as its canonical and built-in type system:

```
<definitions .... >
  <types>
    <xsd:schema .... /> (0 or more)
  </types>
</definitions>
```

The XSD type system can be used to define the types in a message regardless of whether or not the resulting wire format is XML. In our example we have two schema sections. One defines the message format for the input and the other defines the message format for the output.

In our example, the types definition, shown in Example 1-2 on page 24, is where we specify that there is a complex type called `dispatchOrderRequest`, which is composed of two elements:

- ▶ `itemReferenceNumber`
- ▶ `quantityRequired`

```
<types>
  <xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Request.com"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="http://www.exampleApp.dispatchOrder.Request.com"
    xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com">
    <xsd:element name="dispatchOrderRequest" nillable="false">
      <xsd:complexType mixed="false">
        <xsd:sequence>
          <xsd:element name="itemReferenceNumber" nillable="false">
            <xsd:simpleType>
              <xsd:restriction base="xsd:short">
                <xsd:maxInclusive value="9999"/>
                <xsd:minInclusive value="0"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="quantityRequired" nillable="false">
            <xsd:simpleType>
              <xsd:restriction base="xsd:short">
                <xsd:maxInclusive value="999"/>
                <xsd:minInclusive value="0"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  .
  .
</types>
```

Messages

A message represents one interaction between a service requester and a service provider. If an operation is bidirectional at least two message definitions are used to specify the transmissions to and from the service provider. A message consists of one or more logical parts.

```
<definitions .... >
  <message name="nmtoken"> (0 or more)
    <part name="nmtoken" element="qname"(0 or 1) type="qname" (0 or 1)/>
    (0 or more)
  </message>
</definitions>
```

The abstract message definitions are used by the operation element. Multiple operations can refer to the same message definition.

Operations and messages are modeled separately to support flexibility and simplify reuse of existing definitions. For example, two operations with the same parameters can share one abstract message definition.

In our example, the messages definition, shown in Example 1-3, is where we specify the different parts that compose each message. The request message `dispatchOrderRequest` is composed of an element `dispatchOrderRequest` as defined in the schema in the parts section. The response message `dispatchOrderResponse` is similarly defined by the element `dispatchOrderResponse` in the schema. There is no requirement for the names of the message and the schema-defined element to match. In our example we did this merely for convenience.

Example 1-3 Message definition in our WSDL document

```
<message name="dispatchOrderResponse">
  <part element="resns:dispatchOrderResponse" name="ResponsePart"/>
</message>
<message name="dispatchOrderRequest">
  <part element="reqns:dispatchOrderRequest" name="RequestPart"/>
</message>
```

Port types

A port type is a named set of abstract operations and the abstract messages involved:

```
<definitions .... >
  <portType name="nmtoken">
    <operation name="nmtoken" .... /> (0 or more)
  </portType>
</definitions>
```

WSDL defines four types of operations that a port can support:

- ▶ **One-way**
In this operation, the port receives a message. There is an input message only.
- ▶ **Request-response**
In this operation, the port receives a message and sends a correlated message. There is an input message followed by an output message.

- ▶ Solicit-response

In this operation, the port sends a message and receives a correlated message. There is an output message followed by an input message.

- ▶ Notification

In this operation, the port sends a message. There is an output message only. This type of operation could be used in a publish/subscribe scenario.

Each of these operation types can be supported with variations of the following syntax. Presence and order of the input, output, and fault messages determine the type of message:

```
<definitions .... >
  <portType .... > (0 or more)
    <operation name="nmtoken" parameterOrder="nmtokens">
      <input name="nmtoken" (0 or 1) message="qname"/> (0 or 1)
      <output name="nmtoken" (0 or 1) message="qname"/> (0 or 1)
      <fault name="nmtoken" message="qname"/> (0 or more)
    </operation>
  </portType >
</definitions>
```

A request-response operation is an abstract notion. A particular binding must be consulted to determine how the messages are actually sent:

- ▶ Within a single transport-level operation, such as an HTTP request/response message pair, which is the preferred option
- ▶ As two independent transport-level operations, which can be required if the transport protocol only supports one-way communication

In our example, the portType and operation definitions, shown in Example 1-4, are where we specify the port type, called dispatchOrderPort, and a set of operations. In this case, there is only one operation, called dispatchOrder. We specify the interface the Web service provides to possible clients, with the input message DFH0XODSRequest and the output message DFH0XODSResponse. Because the input element appears before the output element in the operation element, our example shows a request-response type of operation.

Example 1-4 Port type and operation definitions in our WSDL document example

```
<portType name="dispatchOrderPort">
  <operation name="dispatchOrder">
    <input message="tns:dispatchOrderRequest" name="DFH0XODSRequest"/>
    <output message="tns:dispatchOrderResponse" name="DFH0XODSResponse"/>
  </operation>
</portType>
```

Bindings

A binding contains:

- ▶ Protocol-specific general binding data, such as the underlying transport protocol and the communication style for SOAP.
- ▶ Protocol extensions for operations and their messages.

Each binding references one port type. One port type can be used in multiple bindings. All operations defined within the port type must be bound in the binding. The pseudo XSD for the binding looks like this:

```
<definitions .... >
  <binding name="nmtoken" type="qname"> (0 or more)
    <!-- extensibility element (1) --> (0 or more)
    <operation name="nmtoken"> (0 or more)
      <!-- extensibility element (2) --> (0 or more)
      <input name="nmtoken" (0 or 1) > (0 or 1)
        <!-- extensibility element (3) -->
      </input>
      <output name="nmtoken"(0 or 1) > (0 or 1)
        <!-- extensibility element (4) --> (0 or more)
      </output>
      <fault name="nmtoken"> (0 or more)
        <!-- extensibility element (5) --> (0 or more)
      </fault>
    </operation>
  </binding>
</definitions>
```

As we have already seen, a port references a binding. The port and binding are modeled as separate entities to support flexibility and location transparency. Two ports that merely differ in their network address can share the same protocol binding.

The extensibility elements <!-- extensibility element (x) --> use XML namespaces to incorporate protocol-specific information into the language- and protocol-independent WSDL specification.

In our example, the binding definition, shown in Example 1-5 on page 28, is where we specify our binding name, `dispatchOrderSoapBinding`. The connection must be SOAP HTTP, and the style must be document. We provide a reference to our operation, `dispatchOrder`, and we define the input message `DFH0XODSRequest` and the output message `DFH0XODSResponse`, both to be SOAP literal.

```
<binding name="dispatchOrderSoapBinding" type="tns:dispatchOrderPort">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="dispatchOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="DFHOXODSRequest">
        <soap:body parts="RequestPart" use="literal"/>
      </input>
      <output name="DFHOXODSResponse">
        <soap:body parts="ResponsePart" use="literal"/>
      </output>
    </operation>
  </binding>
```

Service definition

A service definition merely bundles a set of ports together under a name, as the following pseudo XSD definition of the service element shows.

```
<definitions .... >
  <service name="nmtoken"> (0 or more)
    <port .... /> (0 or more)
  </service>
</definitions>
```

Multiple service definitions can appear in a single WSDL document.

Port definition

A port definition describes an individual endpoint by specifying a single address for a binding:

```
<definitions .... >
  <service .... > (0 or more)
    <port name="nmtoken" binding="qname"> (0 or more)
      <!-- extensibility element (1) -->
    </port>
  </service>
</definitions>
```

The binding attribute is of type QName, which is a qualified name (equivalent to the one used in SOAP). It refers to a binding. A port contains exactly one network address. All other protocol-specific information is contained in the binding.

Any port in the implementation part must reference exactly one binding in the interface part.

The <!-- extensibility element (1) --> is a placeholder for additional XML elements that can hold protocol-specific information. This mechanism is required because WSDL is designed to support multiple runtime protocols.

In our example, the service and port definition, shown in Example 1-6, is where we specify our service, called `dispatchOrderService`, which contains a collection of our ports. In this case, there is only one that uses the `dispatchOrderSoapBinding` and is called `dispatchOrderPort`. In this port, we specify our connection point as, for example, `http://myserver:54321/exampleApp/services/dispatchOrderPort`.

Example 1-6 Service and port definition in our WSDL document example

```
<service name="dispatchOrderService">
  <port binding="tns:dispatchOrderSoapBinding" name="dispatchOrderPort">
    <soap:address
      location="http://myserver:54321/exampleApp/services/dispatchOrderPort"/>
    </port>
  </service>
```

1.6.4 WSDL bindings

We now investigate the WSDL extensibility elements supporting the SOAP transport binding.

SOAP binding

WSDL includes a binding for SOAP 1.1 endpoints, which supports the specification of the following protocol-specific information:

- ▶ An indication that a binding is bound to the SOAP 1.1 protocol
- ▶ A way of specifying an address for a SOAP endpoint
- ▶ The URI for the SOAPAction HTTP header for the HTTP binding of SOAP
- ▶ A list of definitions for headers that are transmitted as part of the SOAP envelope

Table 1-3 lists the corresponding extension elements.

Table 1-3 SOAP extensibility elements in WSDL

Extension and attributes		Explanation
<soap:binding ... >		Binding level; specifies defaults for all operations.
	transport="uri" (0 or 1)	Binding level; transport is the runtime transport protocol used by SOAP (HTTP, SMTP, and so on).
	style="rpc document" (0 or 1)	The style is one of the two SOAP communication styles, rpc or document.
<soap:operation ... >		Extends operation definition.
	soapAction="uri" (0 or 1)	URN.
	style="rpc document" (0 or 1)	See binding level.
<soap:body ... >		Extends operation definition; specifies how message parts appear inside the SOAP body.
	parts="nmtokens"	Optional; allows externalizing message parts.
	use="encoded literal"	literal: messages reference concrete XSD (no WSDL type) encoded: messages reference abstract WSDL type elements encodingStyle extension used.
	encodingStyle="uri-list"(0 or 1)	List of supported message encoding styles.
	namespace="uri" (0 or 1)	URN of the service.
<soap:fault ... >		Extends operation definition; contents of fault details element.
	name="nmtoken"	Relates soap:fault to wsdl:fault for operation.
	use, encodingStyle, namespace	See soap:body.
<soap:address ... >		Extends port definition.
	location="uri"	Network address of RPC router.
<soap:header ... >		Operation level; shaped after <soap:body ...>.
<soap:headerfault ... >		Operation level; shaped after <soap:body ...>.



CICS implementation of Web services

In Chapter 1, “Overview of Web services” on page 1 we learned that a Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Other systems might interact with the Web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols.

In this chapter we discuss how CICS TS V3.1, V3.2, and v4.1 implement Web services. We begin by reviewing the support for SOAP that CICS provides with CICS TS V3. We then discuss the enhancements that CICS TS V4.1 brings to CICS support for SOAP and Web services.

We continue by showing you how to prepare for running a CICS application as a service provider and what happens inside CICS when a service request arrives for a service provider application. Likewise, we discuss preparing to run a CICS application as a service requester and how CICS processes the outbound service request. This leads us to a discussion of the resource definitions that support Web services, namely, the URIMAP, PIPELINE, and WEBSERVICE definitions.

2.1 Support for Web services in CICS

The CICS implementation of Web services consists of several components. There are tools that are used by the application developer to prepare applications for use with the CICS Web services infrastructure. There are CICS resources for administering Web services, and there are optional enhancements that an administrator can enable to extend the capabilities of CICS.

In this chapter we will focus on subjects relevant to application development, but will consider other subjects that might be of interest to application developers outside the typical application development role.

2.1.1 Core aspects of Web services in CICS

Web services first became available as part of CICS in CICS TS V3.1. The core part of that infrastructure is common to all subsequent versions of CICS, and provides the following functions:

- ▶ It includes the Web Services Assistant utility.

The Web Services Assistant is the part of CICS that is most relevant for application developers. It consists of JCL-based tooling for preparing applications for Web services. It is made up of two programs, DFHWS2LS and DFHLS2WS.
- ▶ It supports several different approaches to developing your CICS applications in a Web services environment.
 - You can enable CICS to convert SOAP messages to application data

You can use either the Web Services Assistant or Rational Application Developer for System Z (RDz) to generate artifacts that can be deployed to CICS. These artifacts interface between the application and the rest of CICS and handle all of the SOAP based messaging. This approach minimizes the application programming effort.

In this scenario an artifact referred to as the Web service binding file (also known as the WSBind file) is generated and deployed to CICS.
 - You can take complete control of the processing of your data.

If you prefer to write applications that handle XML natively, or have existing in-house or commercial XML processing tools, you might use them instead of using the Web Services Assistant.

- ▶ It has the concept of a PIPELINE resource. A CICS pipeline is shared by many different Web services, and is used to configure shared qualities of service. It is usually the CICS system programmer who is responsible for configuring PIPELINE resources. This is done using a pipeline configuration file.

WSBind files are installed into a PIPELINE resource in CICS. For each WSBind file that is installed CICS creates a WEBSERVICE resource.

- ▶ It has the concept of handler programs that can be added to a pipeline by the system programmer. Handler programs enable sophisticated additional processing to be performed as part of the process of sending and receiving SOAP messages in CICS.

Several handler programs are supplied with CICS and can be used to add support for additional services such as WS-Security (for identity propagation) and WS-AT (for distributed two-phase commit).

- ▶ It has the concept of a URIMAP resource. A URIMAP associates a particular WEBSERVICE and PIPELINE combination with a specific URI. The system programmer is usually responsible for the URIMAPs, though the application developer might be responsible for selecting the URI for a Web service.

If a SOAP message is received by a CICS region then CICS begins processing it using the associated URI. CICS looks for a URIMAP resource that has been installed for that URI. If a match is found, the URIMAP indicates which PIPELINE to use for the subsequent processing. It also indicates which WEBSERVICE resource CICS should use to transform the XML into application data.

- ▶ It provides an application programming interface (API) to allow CICS applications to interact with the Web services support in CICS. The main commands for doing this are:

- SOAPFAULT ADD | CREATE | DELETE

This is used by provider mode applications that want to return application-specific diagnostics as a SOAP Fault message

- INVOKE WEBSERVICE

This is used by requester mode applications that want to invoke a remote Web service.

- ▶ It conforms to open standards including:
 - SOAP 1.1 and 1.2
 - HTTP 1.1
 - WSDL 1.1

- ▶ It ensures maximum interoperability with other Web services implementations by conforming with the Web Services Interoperability Organization (WS-I) Basic Profile 1.0. This profile is a set of non-proprietary Web services specifications, along with clarifications and amendments to those specifications, which, taken together, promote interoperability between different implementations of Web services.

2.2 Tools for application deployment

In this section, we provide a brief overview of the main application development tools for CICS Web services.

2.2.1 CICS Web Services Assistant

The CICS Web Services Assistant is a pair of JCL utilities that are used by application developers to prepare applications as Web services. It contains two utility programs:

- ▶ DFHLS2WS

DFHLS2WS is used to expose existing CICS applications as provider mode Web services. It takes as input a pair of language structures (typically COBOL copybooks) that define the commarea for the application, and generates a WSDL document that describes a SOAP interface to that same application.

- ▶ DFHWS2LS

DFHWS2LS does the reverse of DFHLS2WS. It takes as input a WSDL document, and generates language structures suitable for use in a new application program. This new application can either implement a Web service in CICS, or invoke a remote Web service from CICS.

Both DFHLS2WS and DFHWS2LS generate a file called the WSBind file. This is installed into a CICS region and contains the meta-data that CICS needs to transform SOAP messages to and from application data.

The Web Services Assistant support the following programming languages:

- ▶ COBOL
- ▶ PL/I
- ▶ C
- ▶ C++

However, the assistant does have some limitations. There are some WSDL documents that are not supported by DFHWS2LS, and some language structures that are not supported by DFHLS2WS.

2.2.2 IBM Rational Developer for System z

IBM Rational Developer for System z is an integrated development environment that helps application developers create applications for the mainframe. RDz offers a significant range of tools of interest to CICS application developers. In this book, however, we focus on those tools that are relevant for CICS Web services.

The Enterprise Services Toolkit (EST) perspective within RDz can be used to simplify the process of exposing existing CICS applications as Web services. It has wizards that guide you through the process of generating and deploying WSBind files for CICS. It also has a compiled technology that compliments the capabilities of DFHLS2WS, but goes beyond it by removing many of the limitations of DFHLS2WS.

Other capabilities that might be of interest to a CICS application developer include:

- ▶ Editors and interactive development environments (IDEs) for Java, COBOL, PL/I, and Enterprise Generation Language (EGL) components including language understanding, syntax checking, and unit testing in CICS
- ▶ Remote z/OS support including data set access, job submission, queue management, UNIX® file system management, and TSO command processing
- ▶ Interactive testing and remote debugging for applications running within CICS
- ▶ A local CICS development environment.

2.2.3 Other Options

There are scenarios where you might decide not to use the Web services Assistants or RDz. For example:

- ▶ You want to create XML aware Web services.
You might want to handle the XML from the body of a SOAP message in your applications rather than allowing CICS to do so.
- ▶ You do not want to use SOAP messages.
If you prefer to use a non-SOAP protocol for your messages, you can do so. However, your application programs will be responsible for parsing inbound messages and constructing outbound messages.
- ▶ You have an application written in an unsupported programming language.
In this case you should either write a wrapper program in a supported language, or write a program to process the XML in your preferred language.

- ▶ The CICS Web Services Assistant does not support your application's data structure.

Although the CICS Web Services Assistant supports the most common data types and structures, some exist that are not supported. That is, there might not always be a 1-1- mapping between XML data types and the data types in your language structure. In this situation, you should first consider providing a “wrapper” program that maps your application's data to a format that the assistant can support. If this is not possible, consider using Rational Application Developer for System z. As a last resort you might need to change your application's data structure.

2.3 CICS as a service provider

When CICS is acting in the role of a Web service provider, it receives SOAP messages from the network and passes them through a pipeline to a target application program. The response from the application is returned to the service requester through the same pipeline. In this section we discuss how to prepare for running a CICS application as a service provider, and how CICS processes the incoming service request.

Most of the actions described in this section are performed by a system programmer. It is helpful for the application developer to be aware of how CICS handles provider mode Web services, but typical service provider applications are unaware that they are being driven as Web services. The following information is therefore provided for background understanding.

2.3.1 Preparing to run a CICS application as a service provider

Suppose that we have an existing commarea-based CICS application that we want to expose as a Web service through HTTP. Suppose also that we want to use the Web Services Assistant to expose the application as a Web service. We go through the following steps:

1. Generate the WSBind and WSDL files (application developer).
 - a. Create an HFS directory in which to store the generated files. For example, we might create a directory named `/u/SharedProjectDirectory/MyFirstWebServiceProvider`.

- b. Run the JCL for DFHLS2WS. The input we provide includes:
 - The name of the CICS PROGRAM resource for the application
 - The names of the partitioned data set members that contain the high-level language structures used by the application program to describe the input and output commarea formats
 - The fully qualified HFS names of the WSBind file and of the file into which the Web service description is to be written (the WSDL file)
 - The URI that a client will use to access the Web service
 - How CICS should pass data to the target application program (COMMAREA or container)

Typically, an application developer would perform this step.

2. Create a TCPIP SERVICE resource definition (system programmer).

The resource definition should specify PROTOCOL(HTTP) and supply information about the port on which inbound requests are received.

Typically, a system programmer would perform this step.

3. Create a PIPELINE resource definition (system programmer).

- a. Create a service provider pipeline configuration file.

A pipeline configuration file is an XML file that describes, among other things, the message handler programs and the SOAP header processing programs that CICS invokes when it processes the pipeline.

- b. Create an HFS directory in which to store installable WSBind and WSDL files.

We call this directory the *pickup directory*, as CICS will pick up the WSBind and WSDL files from this directory and store them on a shelf directory.

- c. Create an HFS directory for CICS to store installed WSBind files.

We call this directory the *shelf directory*.

- d. Create a PIPELINE resource definition to handle the Web service request.

- i. Specify the CONFIGFILE attribute to point to the file created in step 3a.
- ii. Specify the WSDIR attribute to point to the directory created in step 3b.
- iii. Specify the SHELF attribute to point to the directory created in step 3c.

- e. Copy the WSBind and WSDL files created in step 1 on page 36 to the pickup directory created in step 3b.

4. Install the TCPIPSERVICE and PIPELINE resource definitions (system programmer).

When the CICS system programmer installs the PIPELINE definition, CICS scans the pickup directory for WSBind files. When CICS finds the WSBind file created in step 1 on page 36, CICS dynamically creates and installs a WEBSERVICE resource definition. CICS derives the name of the WEBSERVICE definition from the name of the WSBind file. The WEBSERVICE definition identifies the name of the associated PIPELINE definition and points to the location of the WSBind file in the HFS.

During the installation of the WEBSERVICE resource:

- CICS dynamically creates and installs a URIMAP resource definition. CICS bases the definition on the URI specified in the input to DFHLS2WS in step 1 on page 36 and stored by DFHLS2WS in the WSBind file.
 - CICS uses the WSBind file to create main storage control blocks to map the inbound service request (XML) to a COMMAREA or a container and to map to XML the outbound COMMAREA or container that contains the response data.
5. Publish WSDL to clients.
 - a. Customize the location attribute on the <address> element in the WSDL file so that its value specifies the TCP/IP server name of the machine hosting the service and the port number defined in step 2 on page 37.
 - b. Publish the WSDL to any parties wanting to create clients to this Web service.

2.3.2 Processing the inbound service request

Figure 2-1 on page 39 shows the processing that occurs when a service requester sends a SOAP message over HTTP to a service provider application running in a CICS TS V4.1 region.

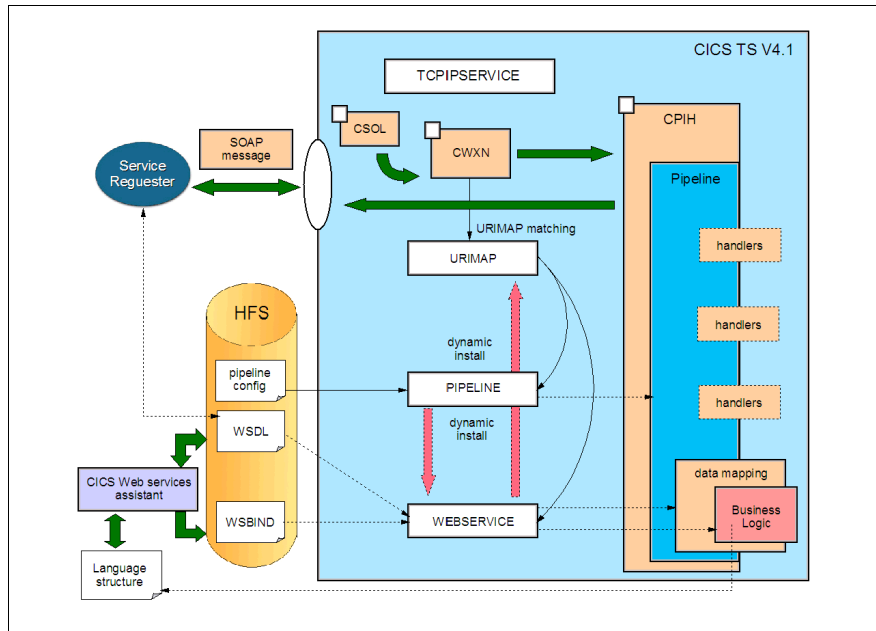


Figure 2-1 Web service runtime service provider processing

The CICS-supplied sockets listener transaction (CSOL) monitors the port specified in the TCPIP SERVICE resource definition for incoming HTTP requests. When the SOAP message arrives, CSOL attaches the transaction specified in the TRANSACTION attribute of the TCPIP SERVICE definition. Normally, this will be the CICS-supplied Web attach transaction CWXN.

CWXN finds the URI in the HTTP request and scans the URIMAP resource definitions for a URIMAP that has its USAGE attribute set to PIPELINE and its PATH attribute set to the URI found in the HTTP request. If CWXN finds such a URIMAP, it uses the PIPELINE and WEBSERVICE attributes of the URIMAP definition to get the name of the PIPELINE and WEBSERVICE definitions that it uses to process the incoming request. CWXN also uses the TRANSACTION attribute of the URIMAP definition to determine the name of the transaction that it should attach to process the pipeline. Often, this will be the CPIH transaction.

CPIH starts the pipeline processing. It uses the PIPELINE definition to find the name of the pipeline configuration file. CPIH uses the pipeline configuration file to determine which pipeline handler programs and SOAP header processing programs to invoke.

A message handler in the pipeline (typically, a CICS-supplied SOAP message handler) removes the SOAP envelope from the inbound request and passes the SOAP body to the application handler program. Usually this will be the CICS supplied application handler, DFHPITP.

DFHPITP uses the DFHWS-WEBSERVICE container to pass the name of the required WEBSERVICE definition to the data mapper. The data mapper uses the WEBSERVICE definition to locate the main storage control blocks that it needs to map the inbound service request (XML) to a COMMAREA or a container.

The data mapper links to the target service provider application program, providing input in the format that it expects. The application program is not aware that it is being executed as a Web service. The program performs its normal processing, then returns an output COMMAREA or container to the data mapper.

The output data from the CICS application program cannot just be sent back to the pipeline code. The data mapper must first convert the output from the COMMAREA or container format into a SOAP body.

The response message is passed back through the pipeline handler programs and is returned to the requester over HTTP.

2.4 CICS as a service requester

When CICS is acting in the role of a Web service requester, a CICS application program sends a SOAP message to a remote Web service through a requester mode pipeline. The response from the service provider is returned to the application program through the same pipeline. In this section we discuss how to prepare for running a CICS application as a service requester, and how CICS processes the outbound service request.

This scenario always involves the creation of a new application program, so it does involve more actions for the application developer than is typically required in provider mode.

2.4.1 Preparing to run a CICS application as a service requester

Suppose we want to write a new CICS application that invokes a remote Web service. Suppose also that we want to use the Web Services Assistant rather than taking control of the processing ourselves.

We go through the following steps:

1. Generate the WSBind file and the language structures (application developer).
 - a. Create an HFS directory in which to store the WSBind file. For example, we might create a directory named `/u/SharedProjectDirectory/MyFirstWebServiceRequester`.
 - b. Run the JCL for DFHWS2LS. The input we provide to the program includes:
 - The fully qualified HFS name of the WSDL file that describes the Web service that we want to invoke.
 - The names of the partitioned data set members into which DFHWS2LS should put the high-level language structures that it generates. The application program uses the language structures to describe the Web service request and the Web service response.
2. Create a PIPELINE resource definition (system programmer).
 - a. Create a service requester pipeline configuration file.

A pipeline configuration file is an XML file that describes, among other things, the pipeline handler programs and the SOAP header processing programs that CICS invokes when it processes the pipeline.
 - b. Create an HFS directory in which to store installable WSBind files.

We call this directory the *pickup directory* because CICS will pick up the WSBIND file from this directory and store it on a shelf directory.
 - c. Create an HFS directory for CICS to store installed WSBind files.

We call this directory the *shelf directory*.
 - d. Create a PIPELINE resource definition to handle the Web service request:
 - i. Specify the CONFIGFILE attribute to point to the file created in step 2a.
 - ii. Specify the WSDIR attribute to point to the directory created in step 2b.
 - iii. Specify the SHELF attribute to point to the directory created in step 2c.
 - e. Copy the WSBind file created in step 1 to the pickup directory from step 2b.
3. Install the PIPELINE resource definition (system programmer).

When the CICS system programmer installs the PIPELINE definition, CICS scans the pickup directory for WSBind files. When CICS finds the WSBind file created in step 1, CICS dynamically creates and installs a WEBSERVICE resource definition for it. CICS derives the name of the WEBSERVICE definition from the name of the WSBind file. The WEBSERVICE definition identifies the name of the associated PIPELINE definition and points to the location of the WSBind file in the HFS.

During the installation of the WEBSERVICE resource, CICS uses the WSBIND file to create main storage control blocks to map the outbound service request to an XML document and to map the inbound XML response document to a language structure.

4. Use the language structure generated in step 1 on page 41 to write the application program (application developer).
 - a. It issues the following command to place the outbound data into container DFHWS-DATA:

```
EXEC CICS PUT CONTAINER(DFHWS-DATA) CHANNEL(name_of_channel)  
FROM(data_area)
```

- b. It issues the following command to invoke the Web service:

```
EXEC CICS INVOKE WEBSERVICE(name_of_WEBSERVICE_definition)  
CHANNEL(name_of_channel) OPERATION(name_of_operation)
```

Note: From CICS TS 4.1 onwards, the EXEC CICS INVOKE SERVICE command should be used for all new Web service applications, rather than the INVOKE WEBSERVICE command which is a synonym

2.4.2 Processing the outbound service request

Figure 2-2 shows the processing that occurs when a service requester running in a CICS TS V4.1 region sends a SOAP message to a service provider.

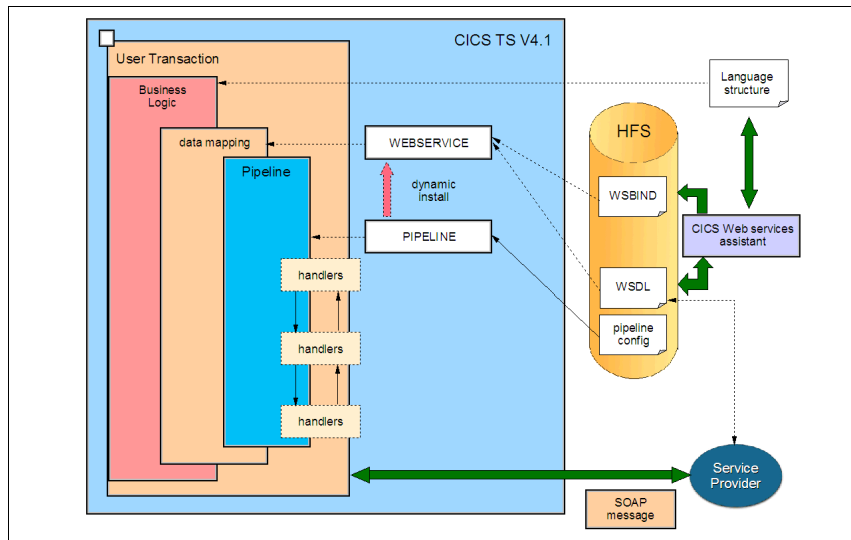


Figure 2-2 Web service requester resources

When the service requester issues the EXEC CICS INVOKE SERVICE command, CICS uses the information found in the WSBind file that is associated with the specified WEBSERVICE definition to convert the language structure into an XML document. CICS then invokes the pipeline handlers specified in the pipeline configuration file, and they convert the XML document into a SOAP message.

CICS will send the request SOAP message to the remote service provider either through HTTP or WebSphere MQ.

When the response SOAP message is received, CICS will pass it back through the pipeline. The message handlers will extract the SOAP body from the SOAP envelope, and the data mapping function will convert the XML in the SOAP body into a language structure that is passed to the application program in container DFHWS-DATA.

2.5 The CICS resource definitions

We now look in more detail at what CICS resources a systems programmer must provide to enable Web services in a CICS environment. Some of these resources are influenced by decisions made by the application developer.

2.5.1 URIMAP

URIMAP definitions are relevant in both provider and requester mode for associating CICS processing with a URI. They are also used with the EXEC CICS WEB API, but that usage scenario is not considered here.

► Provider mode

URIMAP definitions for Web service requests have a USAGE attribute of PIPELINE. These URIMAP definitions associate a URI for an inbound Web service request (that is, a request by which a client invokes a Web service in CICS) with a PIPELINE or WEBSERVICE resource that specifies the processing to be performed. They might also be used to specify:

- The name of the transaction that CICS should use to run the pipeline
- The user ID under which the pipeline transaction runs

► Requester mode

Under CICS TS V3.2 and CICS TS V4.1, URIMAP resources that specify a USAGE of CLIENT can be used when an INVOKE command is processed.

In CICS TS V3.2 a client mode URIMAP can be used to specify cryptographic information for INVOKE commands that involve SSL. CICS will look for an

appropriate client mode URIMAP as the outbound HTTP or HTTPS connection is established and will use the characteristics of the URIMAP if one is found.

In CICS TS V4.1 the application program might specify the name of a client mode URIMAP as a parameter in the INVOKE command. This provides the same benefits as in CICS TS V3.2, but it also allows the URI for the remote Web service to be defined declaratively in a CICS resource. This makes it easier for the system programmer to customize the URI between Test and Production environments.

Figure 2-3 illustrates the purpose of a URIMAP definition in provider mode.

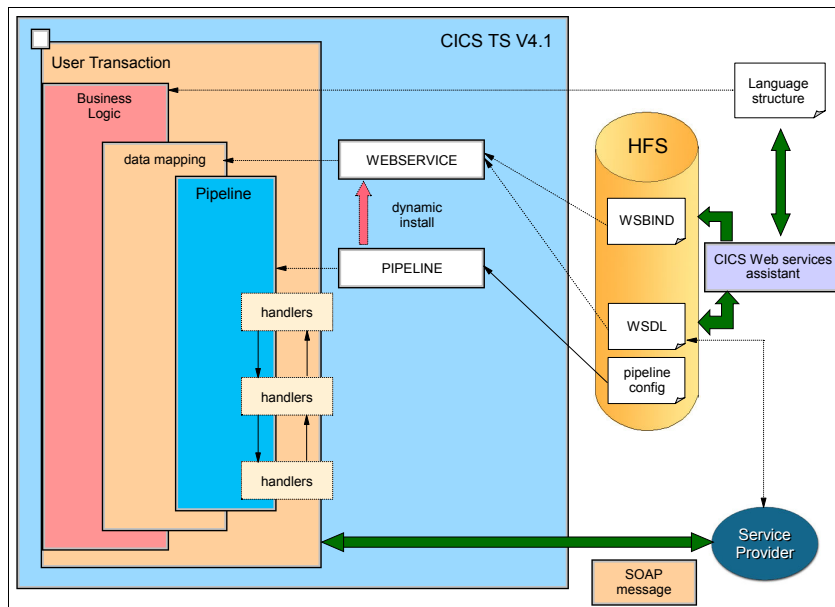


Figure 2-3 Purpose of URIMAP resource in provider mode

You can create URIMAP resource definitions in the following ways:

- ▶ Using the Web services assistant (using a PIPELINE SCAN)
- ▶ Using the CICS Explorer™
- ▶ Using the CEDA transaction
- ▶ Using the DFHCSDUP batch utility
- ▶ Using CICSplex SM Business Application Services
- ▶ Using the EXEC CICS CREATE URIMAP command

The most common mechanism for creating the provider mode URIMAP is for the application developer to decide (in consultation with the system programmer) on the URI to use, and to specify it when the Web Services Assistant is used. This will result in the URI being stored within the WSBind file.

The application developer might also indicate a default TRANSACTION to add to the URIMAP, instead of CPIH. The application developer might also indicate a default user ID under which the Web service should execute. These options are also specified using the Web Services Assistant and are stored in the WSBind file.

When you issue a PERFORM PIPELINE SCAN command (using CEMT or the CICS system programming interface), CICS scans the directory specified in the PIPELINE's WSDIR attribute (the pickup directory), and creates URIMAP and WEBSERVICE resources dynamically using the information from the WSBind files. For each Web service binding file in the directory (that is, for each file with the .wsbind suffix), CICS installs a WEBSERVICE and a URIMAP if one does not already exist. Existing resources are replaced if the information in the binding file is newer than the existing resources.

2.5.2 PIPELINE

A PIPELINE resource definition provides information about the pipeline handlers that will act on a service request and on the response. The information about the pipeline handlers is supplied indirectly. The PIPELINE definition specifies the name of an HFS file, called the pipeline configuration file, which contains an XML description of the pipeline configuration. The most important attributes of the PIPELINE resource definition are as follows:

► WSDIR

The WSDIR attribute specifies the name of the Web service binding directory (also known as the pickup directory). The Web service binding directory contains Web service binding files that are associated with the PIPELINE, and that are to be installed automatically by the CICS scanning mechanism. When the PIPELINE definition is installed, CICS scans the directory and automatically installs any Web service binding files it finds there.

If you specify a value for the WSDIR attribute, it must refer to a valid HFS directory to which the CICS region has at least read access. If this is not the case, any attempt to install the PIPELINE resource will fail.

If you do not specify a value for WSDIR, no automatic scan takes place on installation of the PIPELINE, and PERFORM PIPELINE SCAN commands will fail.

► SHELF

The SHELF attribute specifies the name of an HFS directory where CICS will copy information about installed Web services. CICS regions into which the PIPELINE definition is installed must have full permission to the shelf directory: read, write, and the ability to create subdirectories.

A single shelf directory might be shared by multiple CICS regions and by multiple PIPELINE definitions. Within a shelf directory each CICS region uses a separate subdirectory to keep its files separate from those of other CICS regions. Within each region's directory, each PIPELINE uses a separate subdirectory.

After a CICS region performs a cold or initial start, it deletes its subdirectories from the shelf before trying to use the shelf.

► CONFIGFILE

This attribute specifies the name of the PIPELINE configuration file.

Figure 2-4 illustrates the purpose of the PIPELINE resource definition.

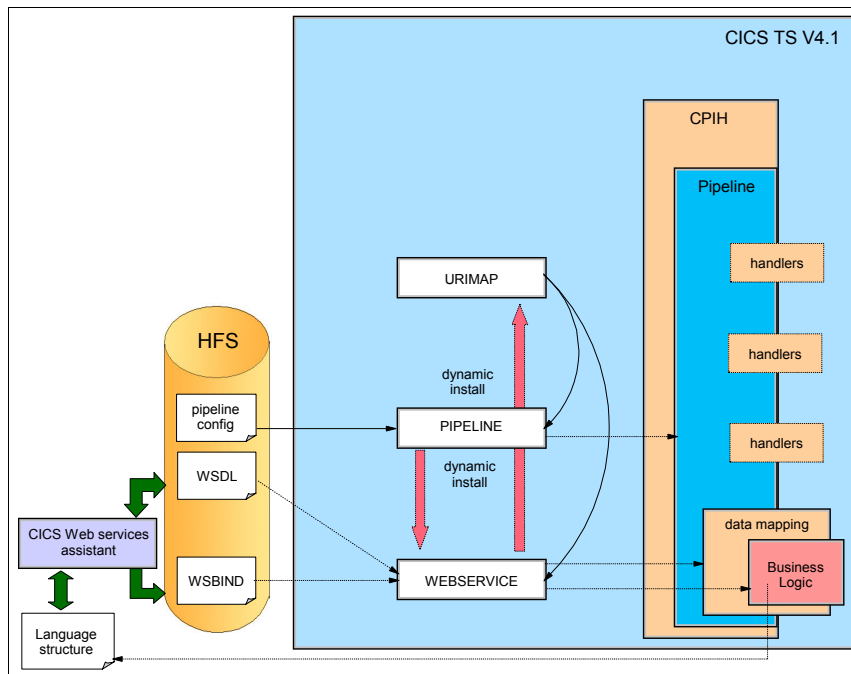


Figure 2-4 Purpose of PIPELINE resource

Pipeline configuration file

When CICS processes a Web service request, it uses a pipeline of one or more handler programs to process the request. The configuration of the pipeline is something that the application developer rarely needs to know about. The system programmer is responsible for the PIPELINE configuration and decides what handler programs are required.

The configuration of a pipeline that is used to handle a Web service request is specified in an XML document, known as a pipeline configuration file. Use a suitable XML editor or text editor to work with your pipeline configuration files.

There are two distinct types of PIPELINE, a requester mode PIPELINE, and a provider mode PIPELINE. This indicates the directionality of the communication. The WSBind files have to be installed into a PIPELINE of the appropriate type. Provider mode PIPELINES are used when exposing CICS applications as Web services. Requester mode PIPELINES are used for invoking remote Web services.

In requester mode, and from CICS TS V3.2 forwards, PIPELINES might specify a time-out value using the RESPWAIT attribute. This is the length of time that CICS will wait for a response.

It is often sufficient to use one of the example pipeline configuration files that CICS provides. These are:

- ▶ `basicsoap11provider.xml`

This file defines the pipeline configuration for a service provider that uses the SOAP 1.1 message handler supplied by CICS.

- ▶ `basicsoap11requester.xml`

This file defines the pipeline configuration for a service requester that uses the SOAP 1.1 message handler supplied by CICS.

For most deployments this is all that the application developer needs to know about the PIPELINE resources. Interested readers can find a lot more information about the content of the PIPELINE configuration files and the nature of pipeline handler programs in the CICS Information Center.

2.5.3 WEBSERVICE

This resource encapsulates a WSBind file in CICS.

Three artifacts define the execution environment that enables a CICS application program to operate as a Web service provider or a Web service requester:

- ▶ The Web service description (WSDL)
- ▶ The Web service binding file (WSBind)
- ▶ The pipeline

These three objects are defined to CICS on the following attributes of the WEBSERVICE resource definition:

- ▶ WSDLFILE
- ▶ WSBIND
- ▶ PIPELINE

The WEBSERVICE definition has a fourth attribute, `VALIDATION`, which specifies whether full validation of SOAP messages against the corresponding schema in the Web service description should be performed at run time. Validation of a SOAP message against a schema incurs considerable processing overhead. You should normally specify `VALIDATION(NO)` in a production environment. `VALIDATION(YES)` ensures that all SOAP messages that are sent and received are valid XML with respect to the WSDL. If `VALIDATION(NO)` is specified, sufficient validation is performed to ensure that the message contains well-formed XML, but more subtle errors might not be detected by CICS.

Figure 2-5 on page 49 illustrates the purpose of the WEBSERVICE resource definition.

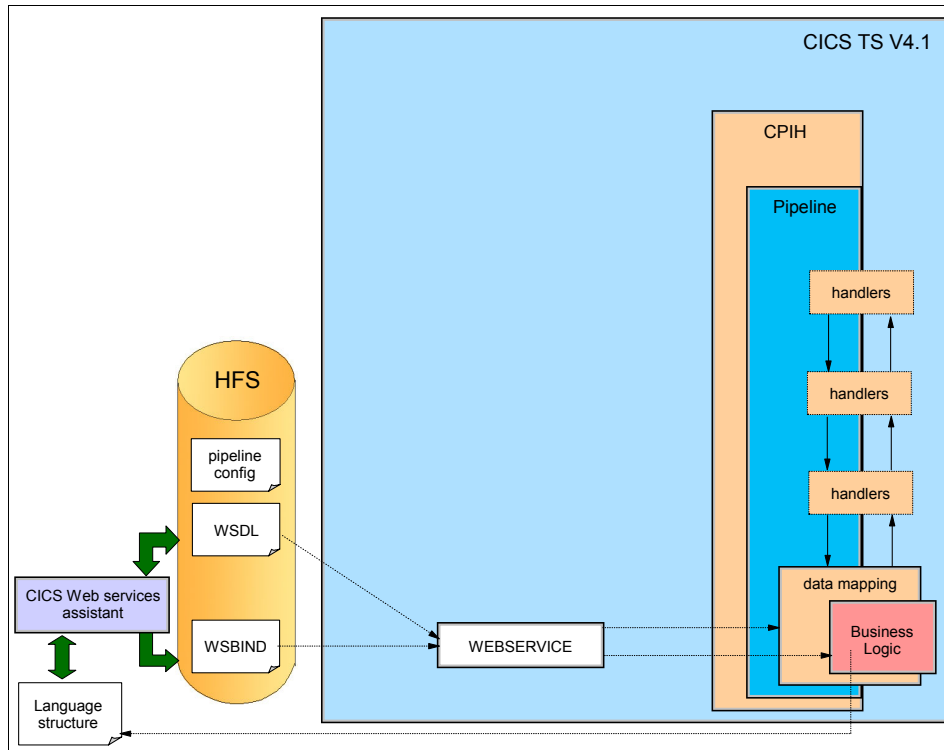


Figure 2-5 Purpose of WEBSERVICE resource

You can create WEBSERVICE resource definitions in the following ways:

- ▶ Using the Web services assistant (using a PIPELINE SCAN)
- ▶ Using the CICS Explorer
- ▶ Using the CEDA transaction
- ▶ Using the DFHCSDUP batch utility
- ▶ Using CICSplex SM Business Application Services
- ▶ Using the EXEC CICS CREATE WEBSERVICE command

When you install a PIPELINE resource, or when you issue a PERFORM PIPELINE SCAN command (using CEMT or the CICS system programming interface), CICS scans the directory specified in the PIPELINE's WSDIR attribute (the pickup directory), and creates URIMAP and WEBSERVICE resources dynamically. For each Web service binding file in the directory (that is, for each file with the .wsbind suffix), CICS installs a WEBSERVICE and a URIMAP if one does not already exist. Existing resources are replaced if the information in the binding file is newer than the existing resources.

The CEMT INQUIRE WEBSERVICE command is used to obtain information about a WEBSERVICE resource definition. The data that is returned depends slightly on the type of Web service. Figure 2-6 shows the types and the data returned for each.

INQUIRE WEBSERVICE command			
• depends on the type of WEBSERVICE			
Attributes	Service Provider	Service Requester to a local service	Service Requester to a remote service
WSDLFILE	Yes	Yes	Yes
WSBIND	Yes	Yes	Yes
PIPELINE	Yes	Yes	Yes
URIMAP	Yes if dynamic installed	empty	empty
BINDING	Yes	Yes	Yes
ENDPOINT	empty	empty	Yes
PROGRAM	Yes	Yes	empty
PGMINTERFACE	Yes	Yes	No
CONTAINER	Yes if Channel is used	Yes if Channel is used	No
VALIDATIONST	Yes	Yes	Yes
LASTMODTIME	Yes	Yes	Yes
STATE	Yes	Yes	Yes

Figure 2-6 CEMT INQUIRE WEBSERVICE command output

2.5.4 The Web service binding file (WSBind)

The WSBind file is a key artifact in the CICS Web services infrastructure as it bridges the gap between the application development tasks, and the CICS runtime. It contains the transformation instructions that CICS uses for transforming application data to and from XML. It also contains deployment information used for creating the WEBSERVICE and URIMAP resources.

The connection between the tooling and the runtime, apart from the language structures that the application programs use, is the WSBind file, as shown in Figure 2-7.

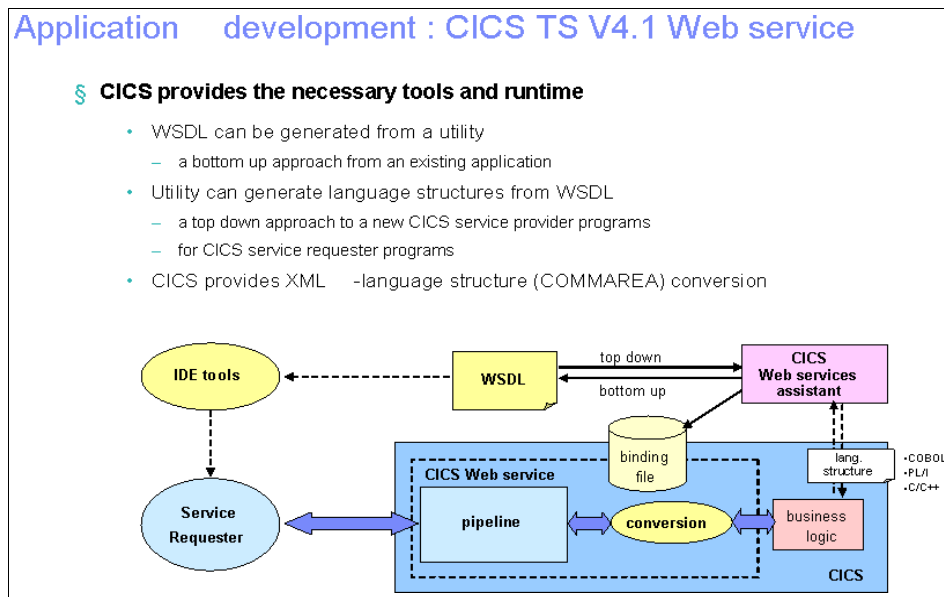


Figure 2-7 WSBind file

The application developer is usually responsible for creating the WSBind file, but it is typically the system programmer who will deploy it into the CICS region.

Figure 2-8 shows CICS usage of the WSBind file as part of the data mapping process within provider and requester mode pipelines.

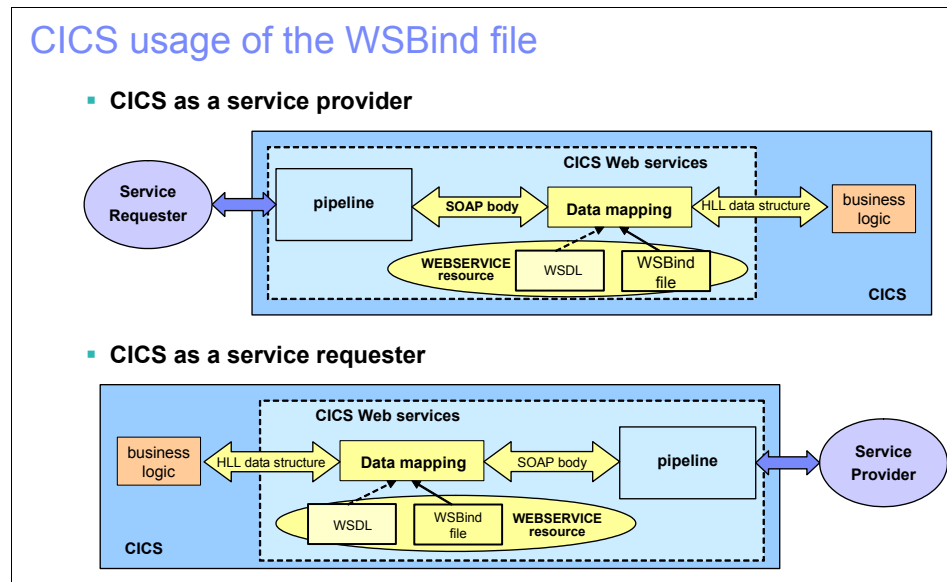


Figure 2-8 CICS usage of the WSBind file

2.5.5 SOAPFAULT commands

Provider mode Web services that are attached using a Channel have the option of sending a SOAP Fault message in response to the requester instead of sending the normal response message. There are three API commands to manage SOAP faults, though for most applications the first one is sufficient:

- ▶ EXEC CICS SOAPFAULT CREATE

Use this command to create a SOAP fault. If a SOAP fault already exists in the context of the SOAP message that is being processed by the message handler, the existing fault is overwritten. When the application returns control to CICS the SOAP Fault message will be generated and sent as a response to the requester.

▶ EXEC CICS SOAPFAULT ADD

Use this command to add either of the following items to a SOAPFAULT object that was created with an earlier SOAPFAULT CREATE command:

- A subcode
- A fault string for a particular national language

If the fault already contains a fault string for the language, then this command replaces the fault string for that language. In SOAP 1.1, only the fault string for the original language is used.

▶ EXEC CICS SOAPFAULT DELETE

Use this command to delete a SOAPFAULT object that was created with an earlier SOAPFAULT CREATE command.

These commands require information that is held in containers on the channel of the CICS-supplied SOAP message handler. To use these commands, you must have access to the channel. Only the following types of programs have this access:

- ▶ Programs that are invoked directly from a CICS-supplied SOAP message handler, including SOAP header processing programs
- ▶ Programs deployed with the Web Services Assistant that have a channel interface. Programs with a COMMAREA interface do not have access to the channel.

Many of the options on the SOAPFAULT CREATE and SOAPFAULT ADD commands apply to SOAP 1.1 and SOAP 1.2 faults, although their behavior is slightly different for each level of SOAP. Other options apply to one SOAP level or the other, but not to both, and if you specify any of them when the message uses a different level of SOAP, the command will raise an INVREQ condition. To help determine which SOAP level applies to the message, container DFHWS-SOAPLEVEL contains a binary fullword with one of the following values:

- ▶ 1: The request or response is a SOAP 1.1 message.
- ▶ 2: The request or response is a SOAP 1.2 message.
- ▶ 10: The request or response is not a SOAP message.

2.5.6 Mapping levels

The Web Services Assistant has evolved over time. Many new capabilities have been added to the Assistant beyond the original capabilities. In some cases those enhancements involve changes to the application bindings shared between CICS and the applications. Where this happens there's a version number, called the *mapping level*, that is used to allow the application developer to select precisely which version of the mapping rules to apply.

It is suggested that you use the most recent mapping level that's available to you.

The options are:

- ▶ Mapping Level 1.0
This was the initial level of capability introduced with CICS TS V3.1.
- ▶ Mapping Level 1.1
This level was introduced in APAR PK15904 for CICS TS V3.1. This APAR introduced a number of changes, including support for `xsd:list` and `xsd:union` data types in the XML schema language.
- ▶ Mapping Level 1.2
This level was introduced in APAR PK23547 for CICS TS V3.1. This APAR extended the support for COBOL data types and for supporting variable length data values.
- ▶ Mapping Level 2.0
This was the initial level of capability introduced with CICS TS V3.2. It's similar to mapping level 1.2.
- ▶ Mapping Level 2.1
This level was introduced in APAR PK59794 for CICS TS V3.2. It adds support for `xsd:any` and `xsd:anyType`.
- ▶ Mapping Level 2.2
This level was introduced in APAR PK69738 for CICS TS V3.2. It adds support for substitution groups and abstract data types.
- ▶ Mapping Level 3.0
This is the initial level of capability introduced for CICS TS V4.1. It adds support for timestamps and truncated data structures.

For a more complete list of capabilities added at each new mapping level, refer to the CICS Information Center. Some additional information is supplied in the following sections.

2.5.7 Additional enhancements with CICS TS V3.2

In this section we will briefly discuss some of the changes to the Web Services support in CICS TS V3.2.

- ▶ Web Services Assistant
 - Under CICS TS 3.2 several new mapping levels are introduced. In the previous section we briefly described some of the new capabilities. It is strongly recommended that applications are developed at the most recent mapping level possible. For CICS TS V3.2 that is currently mapping level 2.2.
 - At mapping level 2.1 DFHWS2LS adds ‘pass-through’ support for `xsd:any`, and in-line support for variably repeating data.
 - Pass-through support for `xsd:any`

This provides a mechanism at mapping level 2.1 by which WSDL documents that make use of the `xsd:any` and `xsd:anyType` constructs can be supported by DFHWS2LS. There is an example of this technique in the chapter on ‘Hints & Tips’.
 - In-line support for variably recurring data

A new parameter has been added to DFHWS2LS at mapping level 2.1 which allows simple variably recurring data to be handled in arrays. This can significantly simplify the application programming model involved. There is an example of this technique in the chapter on ‘Hints & Tips’.
 - At mapping level 2.2 DFHWS2LS adds support for enumerated content models. This includes improved support for `xsd:choice` constructs, together with support for abstract data types and substitution groups. There is an example of this technique in the chapter on ‘Hints & Tips’.
- ▶ CICS TS V3.2 support for external standards
 - WSDL 2.0 support

WSDL 2.0 is a newer, updated version of the WSDL specification. However, it is not widely implemented in Web services implementations from other vendors. DFHLS2WS can generate WSDL 2.0 documents, and DFHWS2LS can parse them, but until there is wider support for WSDL 2.0 from other vendors it is advisable to continue to use WSDL 1.1.

- Support for binary attachments (MTOM/XOP)

This is a wire-level optimization that the system programmer might implement in the PIPELINE configuration file that will result in improved efficiency in moving binary data that is embedded within SOAP messages compared to normal techniques.

Any Web service that involves xsd:base64Binary data types is eligible for this optimization. If they are enabled, the optimizations happen automatically within the pipeline without application changes.

An example of the use of binary data is included in Chapter 10, “Hints and tips” on page 213.

- CICS TS 3.2 support of WS-TRUST

This is a special purpose specification that can be used in combination with WS-Security to use exotic security tokens with CICS Web services. The security infrastructure for Web services is not exposed to the CICS applications, so this is not something most application developers will need to know about. However, the following discussion might be of interest for readers who are familiar with the details of WS-Security.

The Web Services Trust Language specification enhances Web Services Security further by providing a framework for requesting and issuing security tokens, and managing trust relationships between Web service requesters and providers.

This extension to the authentication of SOAP messages enables Web services to validate and exchange security tokens of different types using a trusted third party. This third party is called a Security Token Service (STS).

CICS support for securing Web services has been enhanced to include an implementation of the Web Services Trust Language (or WS-Trust) specification. CICS can now interoperate with a Security Token Service (STS), such as Tivoli® Federated Identity Manager, to validate and issue security tokens in Web services. This enables CICS to send and receive messages that contain a wide variety of security tokens, such as SAML assertions and Kerberos tokens, to interoperate securely with other Web services.

You can configure the CICS-supplied security handler to define how CICS should interact with an STS. The <wsse_handler> element in the pipeline configuration file includes additional elements and attributes to configure this support. CICS can either validate or exchange the first security token or the first security token of a specific type in the message header. If you want more sophisticated processing to take place, CICS provides a

separate Trust client interface that you can use in a custom message handler. You can use the Trust client instead of the security handler or in addition to it.

2.5.8 Additional enhancements with CICS TS 4.1

In this section we will briefly discuss some of the changes to the Web Services support introduced with CICS TS V4.1.

- ▶ CICS generic XML mapping

Under CICS TS V4.1 you can use the CICS XML transformation capability programmatically from within your applications. The CICS XML Assistant is provided to perform the equivalent function of the CICS Web Services Assistant. This utility helps you to create the required artifacts to transform application binary data to XML or transform XML to application binary data.

The XML assistant can create the artifacts in a bundle directory or another specified location on z/OS UNIX.

- Create the mappings using the CICS XML assistant.
- Create the resources in CICS to make the mappings available.
- Create or update an application program to use the TRANSFORM API command. The application must use a channel-based interface.
- Run the application to test that the transformation works as you intended. You can turn on validation to check that CICS converts the data correctly.

- ▶ CICS V4.1 mapping improvements

CICS TS V4.1 supports mapping level 3.0 which in turn adds support for:

- Timestamps. `xsd:dateTime` fields can be mapped as CICS ABSTIME fields.
- Truncated data structures. If an application provides less data to CICS than was expected, CICS can be configured to tolerate this.
- Bottom-up Web service enablement of sophisticated channel-based applications in DFHLS2WS. You can provide an XML channel description file that tells CICS about the set of containers expected by the application. The channel description file identifies each of the containers used by the application, together with an indication of whether they are optional or required, and whether they have text, binary, or structured content.

► CICS Web Services Addressing in CICS TS V4.1

Figure 2-9 is an example of a Web service that used WS-Addressing to route the reply message to a network address other than the one used by the requester.

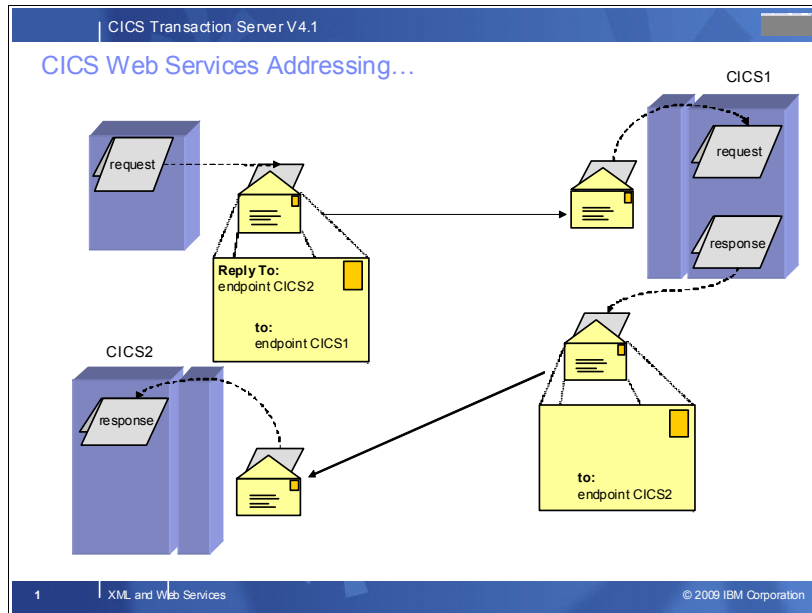


Figure 2-9 A Web service that uses WS-Addressing to route the reply message to a network address other than the one used by the requester

CICS TS V4.1 implements the Worldwide Web Consortium (W3C) Web Services Addressing (WS-Addressing) specifications. This family of specifications provides transport-neutral mechanisms to address Web services and facilitate end-to-end addressing.

WS-Addressing can be used to construct applications with loosely coupled semantics and exotic message exchange patterns.

CICS implements both the W3C WS-Addressing 1.0 Core and W3C WS-Addressing 1.0 SOAP Binding specifications that are identified by the <http://www.w3.org/2005/08/addressing> namespace.

For interoperability, CICS uses the W3C WS-Addressing Submission specification with the <http://schemas.xmlsoap.org/ws/2004/08/addressing> namespace.

2.5.9 Use of WS-Addressing in CICS TS V4.1 applications

WS-Addressing is not something most applications will need to be concerned with. Its most common use is as a middle-ware service used when servers such as WebSphere Application Server and CICS communicate with each other.

However, there are advanced scenarios in which WS-Addressing aware applications can participate in the decisions made by the middle-ware. There are new EXEC CICS API commands available in CICS TS V4.1 to facilitate these scenarios. Developers who want to know more about the EXEC CICS WSACONTEXT and EXEC CICS WSAEPR commands are advised to review the CICS Information Center for details.

2.5.10 Comparing CICS TS V3.1 with later CICS TS versions

Table 2-1 summarizes some of the differences between the support for Web services found in CICS TS V3.1 and the later versions of CICS TS.

Table 2-1 Comparison of CICS TS V3.1 Web services with newer releases

Description	CICS TS V3.1	CICS TS V3.2	CICS TS 4.1
MTOM Support	No	Yes	Yes
SOAP protocol level	SOAP 1.1 and 1.2	SOAP 1.1 and 1.2	1.1 and 1.2
Mapping level	1.0, 1.1, 1.2	1.0, 1.1, 1.2, 2.1, 2.2	1.0, 1.1, 1.2, 2.1, 2.2, 3.0
WSDL version	1.1	1.1, 2.0	1.1, 2.0

Description	CICS TS V3.1	CICS TS V3.2	CICS TS 4.1
Pipeline container names	<ul style="list-style-type: none"> ▶ DFHWS-APPHANDLER ▶ DFHWS-BODY ▶ DFHWS-DATA ▶ DFHWS-OPERATION ▶ DFHWS-PIPELINE ▶ DFHWS-SOAPACTION ▶ DFHWS-SOAPLEVEL ▶ DFHWS-TRANID ▶ DFHWS-URI ▶ DFHWS-USERID ▶ DFHWS-WEBSERVICE ▶ DFHWS-XMLNS ▶ DFHERROR ▶ DFHFUNCTION ▶ DFHHEADER ▶ DFHNORESPONSE ▶ DFHREQUEST ▶ DFHRESPONSE ▶ DFH-HANDLERPLIST ▶ DFH-SERVICEPLIST 	<p>Additional Containers</p> <ul style="list-style-type: none"> ▶ DFHWS-CID-DOMAIN ▶ DFHWS-MTOM-IN ▶ DFHWS-MTOM-OUT ▶ DFHWS-XOP-IN ▶ DFHWS-XOP-OUT ▶ DFHWS-MEP ▶ DFHWS-CTX 	<p>Additional Containers</p> <ul style="list-style-type: none"> ▶ DFH-XML-ERRORMSG
CICS resource definitions	<ul style="list-style-type: none"> ▶ PIPELINE ▶ URIMAP ▶ WEBSERVICE 	<ul style="list-style-type: none"> ▶ PIPELINE ▶ URIMAP ▶ WEBSERVICE 	<ul style="list-style-type: none"> ▶ PIPELINE ▶ URIMAP ▶ WEBSERVICE
CICS API and SPI	<ul style="list-style-type: none"> ▶ CREATE PIPELINE ▶ CREATE URIMAP ▶ CREATE WEBSERVICE ▶ INQUIRE WEBSERVICE ▶ INVOKE WEBSERVICE ▶ PERFORM PIPELINE SCAN ▶ SOAPFAULT ADD ▶ SOAPFAULT CREATE ▶ SOAPFAULT DELETE 	<ul style="list-style-type: none"> ▶ INQ PIPELINE <ul style="list-style-type: none"> – New information returned ▶ SET PIPELINE <ul style="list-style-type: none"> – RESPWAIT might be changed ▶ INQ WEBSERVICE <ul style="list-style-type: none"> – New information returned 	<ul style="list-style-type: none"> ▶ WSAEPR CREATE ▶ WSACONTEXT BUILD ▶ WSACONTEXT GET ▶ WSACONTEXT DELETE ▶ SET / INQ XMLTRANSFORM ▶ INVOKE SERVICE
XML parsing	CICS WSBInd file generated by either CICS Web service assistant or RDz	CICS WSBInd file generated by either CICS Web service assistant or RDz	CICS WSBInd file generated by either CICS Web service assistant or RDz, and CICS XMLTRANSFORM resource



Development approaches

This chapter looks at the application interface and discusses three alternative approaches to developing an application. We also consider the advantages of Rational Application Developer for System Z (RDz) for developing CICS Web services, and compare typical high-function, highly coupled approaches with the Web services style.

3.1 Introduction

There are three major application development scenarios involving CICS Web services applications:

- ▶ We have an existing application that we want to expose as a Web service.
- ▶ We want to develop a new application and make it available as a Web service.
- ▶ We want to invoke an existing Web service, probably hosted on another platform.

In all three of these scenarios we have either an existing application or a WSDL description of a Web service.

In the first scenario we want to expose an existing application as a Web service. We will usually use an approach referred to as *bottom-up* Web service enablement. (See 3.2, “Bottom-up approach” on page 63) We start with the language structures that describe the commarea for the existing application. From them, we generate the WSDL and other infrastructure elements until we have a full fledged, published Web service. This approach involves the use of either DFHLS2WS or RDz.

In the second scenario we want to host a new Web service in CICS. We will usually use an approach referred to as *top-down* Web service enablement. (See 3.3, “Top-down approach” on page 65) We start with the WSDL description of the service. From that we generate a set of language structures from which a new application can be constructed. This approach involves the use of either DFHWS2LS or RDz.

In the third scenario we want to invoke a remote Web service from CICS. We usually use the top-down approach in this scenario too, and generate language structures from the WSDL description of the remote Web service. This approach involves the use of either DFHWS2LS or RDz.

There is a special variation of the first scenario in which there is both an existing application in CICS, an existing WSDL description of a service, and a requirement to match the existing application to the existing WSDL. The approach used is referred to as *meet-in-the-middle*. (See 3.4, “Meet-in-the-middle approach” on page 66) One common scenario where this approach is used is where bottom-up enablement of an existing application is performed, followed by customization of the resultant WSDL. After which it is necessary to match the existing application to the new WSDL. This method often involves the use of a wrapper or driver program.

See Figure 3-1 on page 63 for a diagram showing the different approaches.

In some advanced scenarios it might be desirable to write applications that are XML-aware and opt-out of the CICS- and RDz-supplied Web services tooling. In this case, you can write applications that interface directly with the CICS pipeline.

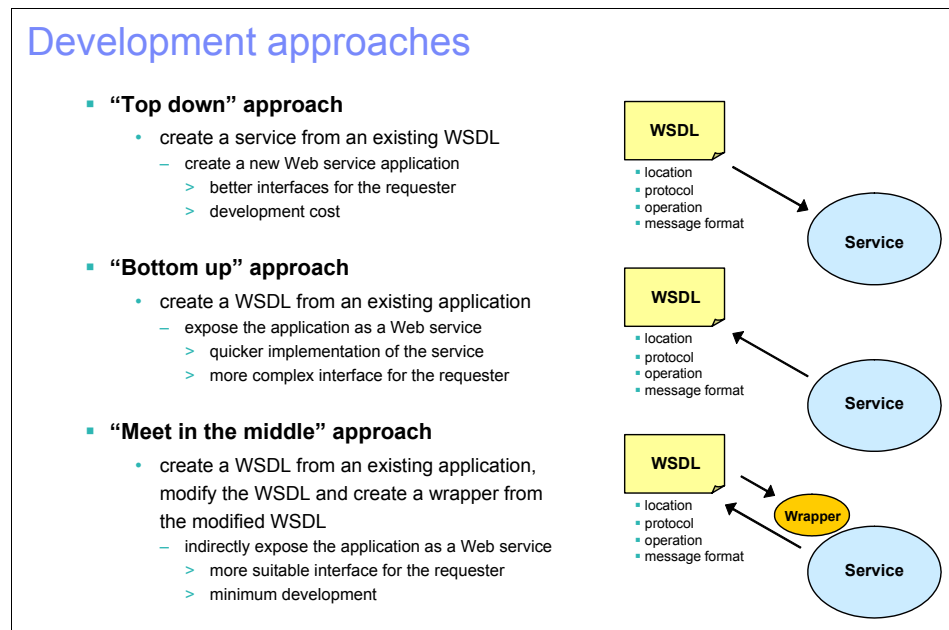


Figure 3-1 Development approaches

In summary, the three approaches can be mapped as shown in Table 3-1.

Table 3-1 The different approaches

Approach	Application	WSDL	Type
Bottom-up	Existing	New	Service provider
Top-down	New	Existing	Service provider
Top-down	New	Existing	Service requester
Meet-in-the-middle	Existing	Existing	Service provider

3.2 Bottom-up approach

This approach is usually the starting point when we have an existing CICS application that is already in production and has either a COMMAREA or Channel-based interface. See Figure 3-2 on page 64. We now want to expose this application to remote client applications using CICS Web services support.

We can either use DFHLS2WS to do this, or RDz.

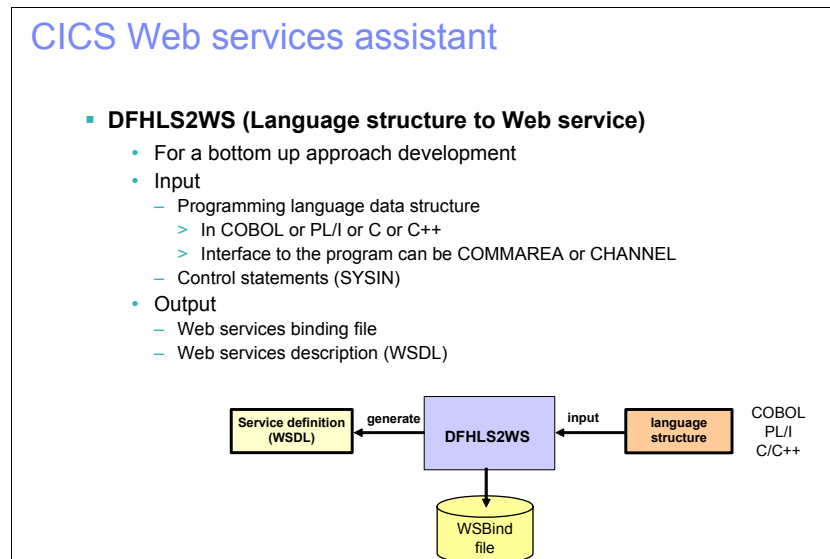


Figure 3-2 Bottom-up approach

Typically the process works this way:

1. Identify the language structures that document the input and output COMMAREA formats for the application:
 - If DFHLS2WS is used, ensure that the language structures are available as separate copybooks in a partitioned data set. Also ensure that the language structures restrict themselves to constructs that are supported by DFHLS2WS. This might involve creating simplified versions of the copybooks if the originals are complicated.
 - If using RDz rather than DFHLS2WS, there is no requirement to separate the language structures out into separate copybooks, and the range of supported language constructs is broader.
2. Generate the WSDL file and the WSBind file for the Web service using either DFHLS2WS or RDz. You must specify the name of the existing PROGRAM resource in CICS and the URI under which you want the Web service to be exposed as input parameters.
3. Identify a PIPELINE into which the WSBind file can be deployed. For unit testing purposes it is common to have a shared PIPELINE in the CICS region. The PIPELINE must be a provider mode PIPELINE. Cause the WSBind file to be placed in the WSDIR for the PIPELINE. This can be done by manually copying the file to the relevant destination on the UNIX file system, or by using RDz.

4. Cause a PIPELINE SCAN command to be issued that will install a WEBSERVICE and URIMAP resource for you. This can be done using the CEMT transaction in CICS, or by allowing RDz to install the artifacts on your behalf.

3.3 Top-down approach

This approach is usually the starting point when we have an existing WSDL document for a Web service, and we want to either implement or invoke the Web service within CICS.

We can either use DFHWS2LS to do this, or RDz, as shown in Figure 3-3.

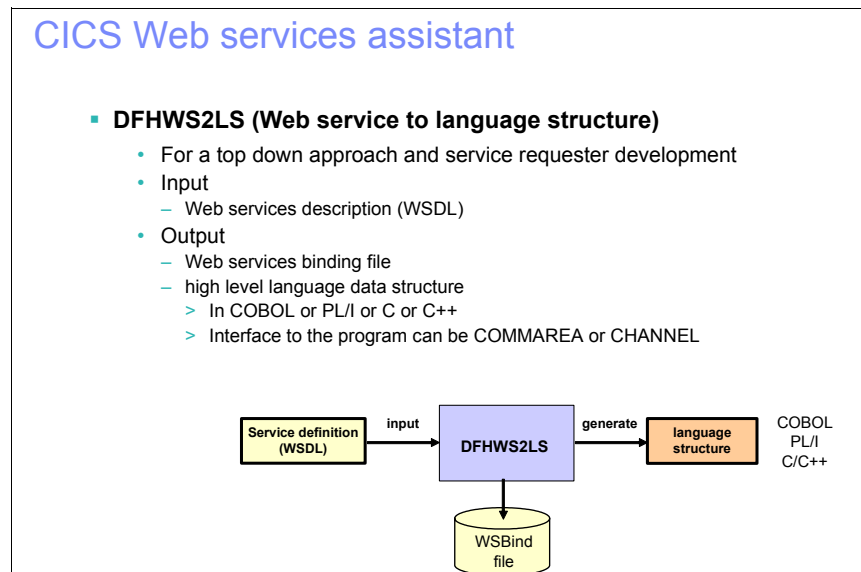


Figure 3-3 Top-down approach

Typically the process works this way:

1. Identify the WSDL that describes the Web service. If using DFHWS2LS, store a copy of the WSDL in a directory in the UNIX file system.
2. Generate the language structures and WSBind file for the new application from the WSDL using either DFHWS2LS or RDz. If the service is being implemented in CICS, you will have to specify the name of the new PROGRAM resource and the URI under which you want the Web service to be exposed as input parameters. If the service is being invoked from CICS, do not specify the name of a PROGRAM resource.

3. Identify a PIPELINE into which the WSBind file can be deployed. For unit testing purposes it is common to have a shared PIPELINE in the CICS region. The PIPELINE must be of the correct type. If the Web service is being implemented in CICS, it must be a provider mode PIPELINE. If the Web service is being invoked from CICS, it must be a requester mode PIPELINE. Cause the WSBind file to be placed in the WSDIR for the PIPELINE. This can be done by manually copying the file to the relevant destination on the UNIX file system, or by using RDz.
4. Cause a PIPELINE SCAN command to be issued, which will install a WEBSERVICE resource and, in provider mode, a URIMAP resource for you. This can be done using the CEMT transaction in CICS, or by allowing RDz to install the artifacts on your behalf.
5. Implement the new application using the generated language structures.
6. Test and deploy the Web service.

3.4 Meet-in-the-middle approach

This third approach can be used in more complicated scenarios. For example:

- ▶ If an existing application's COMMAREA interface has fields that are unsupported by DFHLS2WS
- ▶ If the programming language used is not supported by DFHLS2WS (for example an assembler program)
- ▶ If WSDL generated by DFHLS2WS is modified to make it more fully address a business requirement
- ▶ If an industry standard WSDL document is to be implemented in CICS using existing applications that are known to satisfy the requirements

This is a hybrid technique and often involves the use of a wrapper program that maps between the data format generated by DFHWS2LS (or RDz) and the desired data format used by the existing application. In some cases the existing application can be modified to use the language structures generated by DFHWS2LS (or RDz). See Figure 3-4 on page 67.

Meet In The Middle

- **If you have an existing application and...**
 - an existing WSDL is to be used as interface to the client
 - e.g. WSDL defined from a requesters perspective
 - only want to expose fields that are necessary to the requester
 - existing language structure may be complex, contain unnecessary fields for the requester
 - use an interface more suitable for the requester
 - the existing language structure uses data types not supported by the utility
 - wrapper program converts the data type to a supported data type
 - the existing application is written in a language not supported by the utility
 - Assembler or Java programs
 - etc.

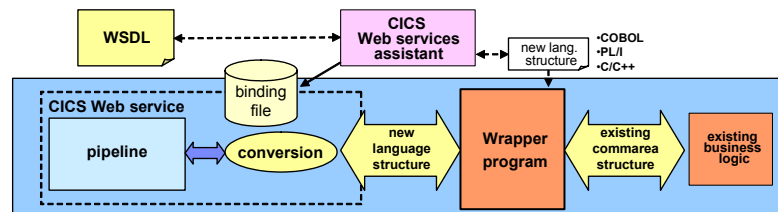


Figure 3-4 Meet-in-the-middle approach

The process flow for this approach is as follows:

1. Start with the WSDL. In some scenarios this might have been generated by DFHLS2WS (or RDz) following the bottom-up approach described in 3.2, “Bottom-up approach” on page 63 (perhaps using a simplified version of the language structures). In some scenarios generated WSDL might have been modified so that it is no longer suitable for use with the original application.
2. Generate new language structures and a WSBind file from the WSDL using DFHWS2LS or RDz.
3. Create a new application that expects input in the form described by the new language structures and which LINKs to the existing PROGRAM using data in the form described by the old language structures.
4. Deploy and test the artifacts as previously described.

RDz contains a set of tools that can simplify the process of performing meet-in-the-middle application development. It can be used to match an existing application to an existing WSDL without writing new application code.

3.5 The advantages of using RDz

RDz contains a lot of tools that simplify the application development exercise for CICS Web services.

For bottom-up development RDz supports two different technologies for transforming SOAP data into application data:

- ▶ Interpreted conversion
- ▶ Compiled conversion

For top-down development it only has Interpreted conversion.

It also has additional capabilities, such as the ability to deploy generated artifacts to a unit testing CICS region, or the ability to view the contents of a WSBind file and to change the deployment information within that file. It can validate that WSDL documents are standards compliant, and it can be used to test Web services deployed in CICS.

The Interpretive XML Conversion (see Figure 3-5 on page 69):

- ▶ Provides a wizard that invokes the CICS Web Services Assistant Java classes 'under the covers' within RDz to produce the WSDL and WSBind file
- ▶ Uses the same CICS-supplied SOAP transformation technology as used by DFHLS2WS and DFHWS2LS
- ▶ Provides additional options beyond the capabilities of the Web services assistants, such as the ability to identify language structures from anywhere within a COBOL program, or the ability to suppress undesirable fields from the WSDL interface during bottom-up enablement
- ▶ Generates source code for a skeleton service implementation for top-down provider mode scenarios
- ▶ Generates source code for a skeleton program containing the Web service INVOKE command for top-down requester mode scenarios

RDz “Interpretive” XML Conversion

- **Invokes the CICS Web Services Assistant**
 - Same Java classes as used on mainframe supplied with CICS
 - Graphical user interface
 - WSBind and WSDL file generation is performed on your workstation

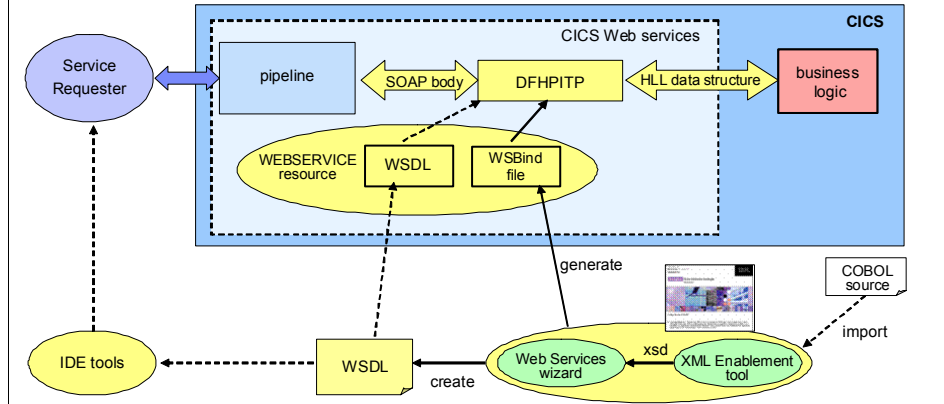


Figure 3-5 RDz interpretive XML conversion

The Compiled XML Conversion (see Figure 3-6 on page 70)

- ▶ Provides a similar wizard to the Interpreted conversion that also generates a WSBind file and WSDL.
- ▶ Generates the source code for a COBOL converter program that implements the XML transformations. This converter program can make use of the z/OS XML System Services XML parser.
- ▶ Supports a broader range of COBOL data structures than are supported by DFHLS2WS, including OCCURS DEPENDING ON.
- ▶ Allows you to modify the generated code for local requirements.
- ▶ Provides the same additional capabilities available with the Interpreted conversions (advanced input selection and field suppression), along with further options to control the mappings on a field by field basis.
- ▶ Requires the generated converter program to be compiled and deployed to the CICS environment.

RDz “Compiled” XML Conversion

- Generates COBOL programs that have to be compiled
 - Graphical user interface
 - WSBind and WSDL file generation is performed on your workstation
 - Use of a “Vendor Segment” in the WSBind file tells DFHPITP to use the converter program instead of using its own conversion mechanism.

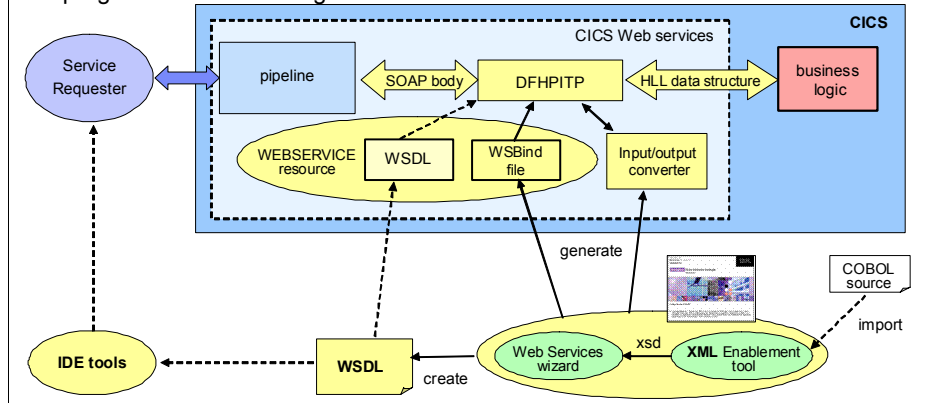


Figure 3-6 RDz compiled XML conversion

3.6 Web services versus CICS TCP/IP connectivity

It is worth looking at the key differences between CICS Web services and the other ways for interacting with CICS over TCP/IP. The main alternative options are:

- ▶ CICS Web Support (the EXEC CICS WEB api)
- ▶ CICS Transaction Gateway
- ▶ z/OS Communications Server IP CICS Socket Interface
- ▶ Link3270 Bridge

All of these components involve a tightly coupled approach. The CICS programs might have to be Web or Sockets aware. The client programs often require detailed knowledge of the commarea (or equivalent) interface with CICS, or are limited to a browser-like interface to the CICS application.

This contrasts with the whole philosophy of Web services where, due to the published WSDL, the client application can determine the required interface and is totally unaware of the language and environment of the runtime executable.

Most developers who write Web service client applications do not know or care that the target service is implemented in COBOL or hosted in CICS.

3.7 Conclusions

This chapter looked at the application interface and discuss the current approaches to developing a Web service, especially in light of the additional CICS V3 and V4 tools (such as the Web Services Assistant) that are now available.

We looked at the bottom-up, top-down, and meet-in-the-middle approaches and examined when each was most appropriate.

We also attempted to put into perspective how the Web services facilities in CICS V3, and V4 differed from the plethora of components in the CICS Web support area, the key difference being the concepts behind tightly coupled and loosely coupled interfaces.



CICS catalog manager example application

The CICS catalog example application is a working COBOL application that is designed to illustrate best practice when connecting CICS applications to external clients and servers. This sample is available in CICS TS 4.1.

In this chapter we review the steps that are required to install this in your CICS environment.

4.1 Samples for use with CICS Web Services

There are several sample applications available to demonstrate the use of Web services in CICS. In this chapter we focus on the CICS Catalog Sample Application, which is a sample distributed with CICS and which demonstrates many different aspects of the CICS Web services infrastructure.

There are other samples available that also demonstrate CICS Web services. In particular, it is recommended that you review the samples in CICS SupportPac CA1P, which is available at the following Web page:

<http://www-01.ibm.com/support/docview.wss?uid=swg24020774>

Chapter 6, “Exposing the Catalog Sample CICS application as a Web service” on page 137 demonstrates how to expose the catalog sample application as a Web service, and how to test it using the Web Services Explorer in Rational Developer for System z (RDz) (or the free Eclipse product).

4.2 Introduction to the catalog manager application

The Web services example application demonstrates how you can use SOAP and Web services to make existing, CICS-controlled information available to SOA service requesters.

The catalog manager example application accesses an order catalog stored in a VSAM file.

The example application is a catalog-management, purchase order style application. It is a simple application that provides the functions to list details of an item in the catalog and then select a quantity of that item to order. The catalog is then updated to reflect the new stock levels. If selected in the application configuration, an outbound Web service call is then made to an external dispatch manager Web service. Figure 4-1 on page 75 shows an overview of this.

Before we can consider the Web service enablement of this sample, it is first necessary to install the sample.

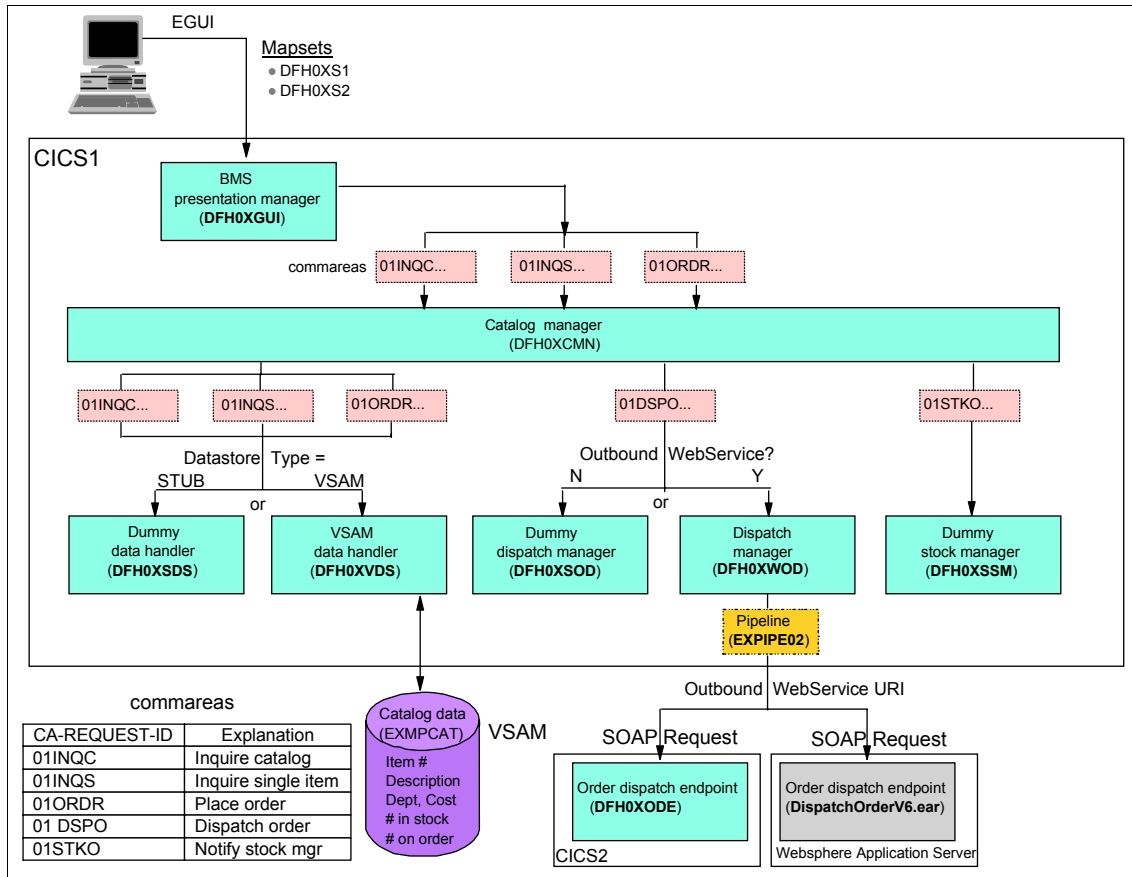


Figure 4-1 Catalog sample overview

4.3 Installation and set up of the base application

The base application is an implementation of the catalog manager application using a 3270 interface initiated by running transaction EGUI.

Before you can run the base application, you must define and populate two VSAM data sets:

- ▶ EXMPCAT: The application configuration file
- ▶ EXMPCONF: The VSAM catalog data file

You must also install two transaction definitions:

- ▶ ECFG
- ▶ EGUI

4.3.1 Creating the VSAM data sets

The SDFHINST dataset, created when you installed CICS TS 4.1, supplies JCL to create and load both the configuration file and the catalog data file.

The two members required are as follows:

- ▶ DFH\$ECNF

This member contains the JCL required to generate the configuration data set. The job contains two IDCAMS steps. The first defines the dataset and the second loads the configuration data for the application. The configuration data can be changed after the dataset is loaded by using the ECFG transaction as shown by Figure 4-2

- ▶ DFH\$ECAT

This member contains the JCL required to generate the catalog data set. The job contains two steps. The first defines the dataset and the second loads the dataset with the all the items in the catalog.

Both of these jobs should be modified as necessary and run.

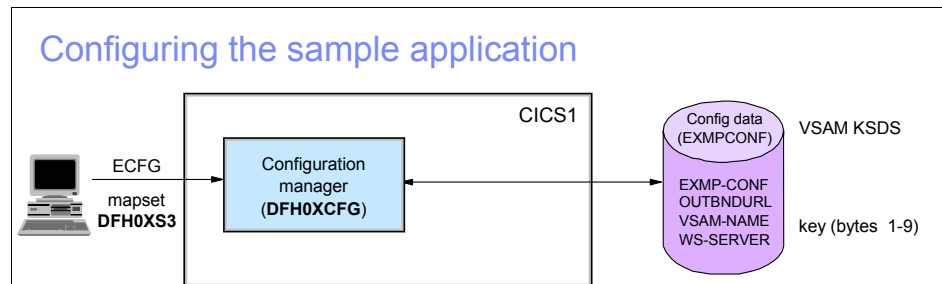


Figure 4-2 Catalog sample configuration

4.3.2 Defining the base application to CICS

The example application is supplied with a 3270 interface to customize and run the application. This interface consists of two transactions (EGUI and ECFG) which, along with the two files created earlier (EXMPCAT and EXMPCONF), must be defined to CICS.

The definitions for these (and related) resources can be found in CICS-supplied CSD group DFH\$EXBS. You will need to modify the definitions for the two files to use the DSNAMES that you have selected.

Copy the definitions into your own CSD group from the CICS-supplied group: DFH\$EXBS (for example, SOADEV). Your group should be similar to that in Example 4-1

Example 4-1 Base application files

EXMPCAT	FILE	SOADEV
EXMPCONF	FILE	SOADEV

Ensure that the both files are defined with Add, BRowse, DElete, READ, and UPDATE all set to Yes, as in Example 4-2.

Example 4-2 Catalog file attributes in CICS

OPERATIONS			
Add	: Yes	No	Yes
BRowse	: Yes	No	Yes
DElete	: Yes	No	Yes
READ	: Yes	Yes	No
UPDATE	: Yes	No	Yes

Now install this group in CICS using **CEDA I G(SOADEV)**.

4.3.3 Configuring the example application

The base application includes a transaction (ECFG) that can be used to configure the example application.

The configuration transaction uses mixed-case information. You must use a terminal that can handle mixed-case information correctly. Use the CEOT transaction (**CEOT UCTRAN**) to switch off uppercase translation.

The transaction enables you to specify several aspects of the example application, including:

- ▶ The overall configuration of the application, such as the use of Web services
- ▶ The network addresses used by the Web services components of the application
- ▶ The names of resources, such as the file used for the data store
- ▶ The names of programs used for each component of the application

The configuration transaction enables you to replace CICS-supplied components of the example application with your own, without restarting the application.

Enter the transaction ECFG to start the configuration application. CICS displays the screen shown in Figure 4-3.

Figure 4-3 shows that the application is configured to use the VSAM datastore and that we will not use an outbound Webservice at this stage (that is, the dummy dispatch manager program DFH0XSOD program will be used).

```
CONFIGURE CICS EXAMPLE CATALOG APPLICATION

      Datastore Type ==> VSAM                STUB|VSAM
      Outbound Webservice? ==> NO           YES|NO
      Catalog Manager ==> DFHOXCMN
      Data Store Stub ==> DFHOXSDS
      Data Store VSAM ==> DFHOXVDS
      Order Dispatch Stub ==> DFHOXSOD
      Order Dispatch Webservice ==> DFHOXWOD
      Stock Manager ==> DFHOXSSM
      VSAM File Name ==> EXMPCAT
      Server Address and Port ==> myserver:99999
      Outbound Webservice URI ==> HTTP://9.42.170.141:9081/EXAMPLEAPP/SERVICES
      ==> /DISPATCHORDERPORT
      ==>
      ==>
      ==>
      ==>

PF              3  END                      12
CNCL
```

Figure 4-3 Configure CICS CATALOG sample application

A full description for each of the fields on the configuration screen follows:

- ▶ Datastore Type
 - Specify STUB if you want to use the Data Store Stub program.
 - Specify VSAM if you want to use the VSAM data store program.

- ▶ Outbound WebService
 - Specify YES if you want to use a remote Web service for your Order Dispatch function (that is, if you want the catalog manager program to link to the Order Dispatch Web service program).
 - Specify NO if you want to use a stub program for your Order Dispatch function (that is, if you want the catalog manager program to link to the Order Dispatch Stub program).
- ▶ Catalog Manager

Specify the name of the catalog manager program. The program supplied with the example application is DFH0XCMN.
- ▶ Data Store Stub

If you specified STUB in the Datastore Type field, specify the name of the Data Store Stub program. The program supplied with the example application is DFH0XSDS.
- ▶ Data Store VSAM

If you specified VSAM in the Datastore Type field, specify the name of the VSAM data store program. The program supplied with the example application is DFH0XVDS.
- ▶ Order Dispatch Stub

If you specified NO in the Outbound WebService field, specify the name of the Order Dispatch Stub program. The program supplied with the example application is DFH0XSOD.
- ▶ Order Dispatch WebService

If you specified YES in the Outbound WebService field, specify the name of the program that functions as a service requester. The program supplied with the example application is DFH0XWOD.
- ▶ Stock Manager

Specify the name of the Stock Manager program. The program supplied with the example application is DFH0XSSM.
- ▶ VSAM File Name

If you specified VSAM in the Datastore Type field, specify the name of the CICS FILE definition. The name used in the example application as supplied is EXMPCAT.
- ▶ Server Address and Port,

If you are using the CICS Web service client, specify the IP address and port of the system on which the example application is deployed as a Web service

- ▶ Outbound WebService URI

If you specified YES in the Outbound WebService field, specify the location of the Web service that implements the dispatch order function. If you are using the supplied CICS endpoint, set this to:

```
http://myserver:myport/exampleApp/dispatchOrder
```

In the above location, *myserver* and *myport* are your CICS server address and port, respectively.

4.3.4 Configuring code page support

As supplied, the example application uses two coded character sets. You must configure your system to enable data conversion between the two character sets.

The CCSID description for the coded character sets used in the example application are:

- ▶ **037** EBCDIC Group 1: USA, Canada (z/OS), Netherlands, Portugal, Brazil, Australia, New Zealand)
- ▶ **1208** UTF-8 Level 3

You must have support for the following statements added to the conversion image for your z/OS system:

- ▶ CONVERSION 037,1208;
- ▶ CONVERSION 1208,037;

To determine whether support is already there, issue the following command from SDSF:

```
/DISPLAY UNI,ALL
```

This returns a display similar to Figure 4-4 on page 81.

```
RESPONSE=system
CUN3000I 13.50.10 UNI DISPLAY 611
  ENVIRONMENT: CREATED      07/02/2005 AT 08.59.15
                MODIFIED   07/02/2005 AT 08.59.17
                IMAGE CREATED 10/22/2004 AT 10.43.15
  SERVICE: CHARACTER      CASE          NORMALIZATION  COLLATION
  STORAGE: ACTIVE        440 PAGES
                LIMIT      51200 PAGES
  CASECONV: NORMAL
  NORMALIZE: DISABLED
  COLLATE: DISABLED
  CONVERSION: 00850-01047-ER          01047-00850-ER
                00037-01200-ER          01200-00037-ER
                00037-01208-ER          01208-00037-ER
                00437-01208-ER          01208-00437-ER
                00037-00367-ER          01252-00037-ER
```

Figure 4-4 DISPLAY UNI, ALL command

If the conversion is not evident, this should be discussed with the z/OS system programmer to have such support added. This will involve the system programmer in generating a conversion image using the CUNMIUTL utility program and enabling access to the image through the corresponding SYS1.PARMLIB member updates. For more details about conversion images, read *z/OS Support for Unicode: Using Conversion Services*, SA22-7649.

4.4 Web service support for the example application

After the base application has been implemented successfully we extend it further to use Web services. Web service support extends the example application by providing:

- ▶ A Web client front end
- ▶ A CICS Web service client
- ▶ Two versions of a Web service endpoint for the order dispatcher component

4.4.1 The Web client front end

The Web client front end is a Web-based front end to the catalog manager application, which can be used in place of the 3270 front end used in the base application.

The Web client front end is supplied as an enterprise archive file (EAR) that can be deployed into an application server such as one of the following:

- ▶ WebSphere Application Server Version 6 or later.
- ▶ Rational Application Developer Version 7 (RAD) or later with a WebSphere unit test environment (version 6 or higher)
- ▶ Rational Application Developer for zSeries® Version 7 (RDz) or later with a WebSphere unit test environment (version 6 or higher)

The supplied ear file is `ExampleAppClientV6.ear`.

The Web client front end is configured to call the catalog manager application as a Web service provider that has three endpoints in CICS. These correspond to the three commarea interfaces that the catalog manager program (DFH0XCMN) uses in the base application. This client replaces the BMS interface used in the base application (DFH0XGUI).

The Web client front end can be seen in Figure 4-5 on page 83.

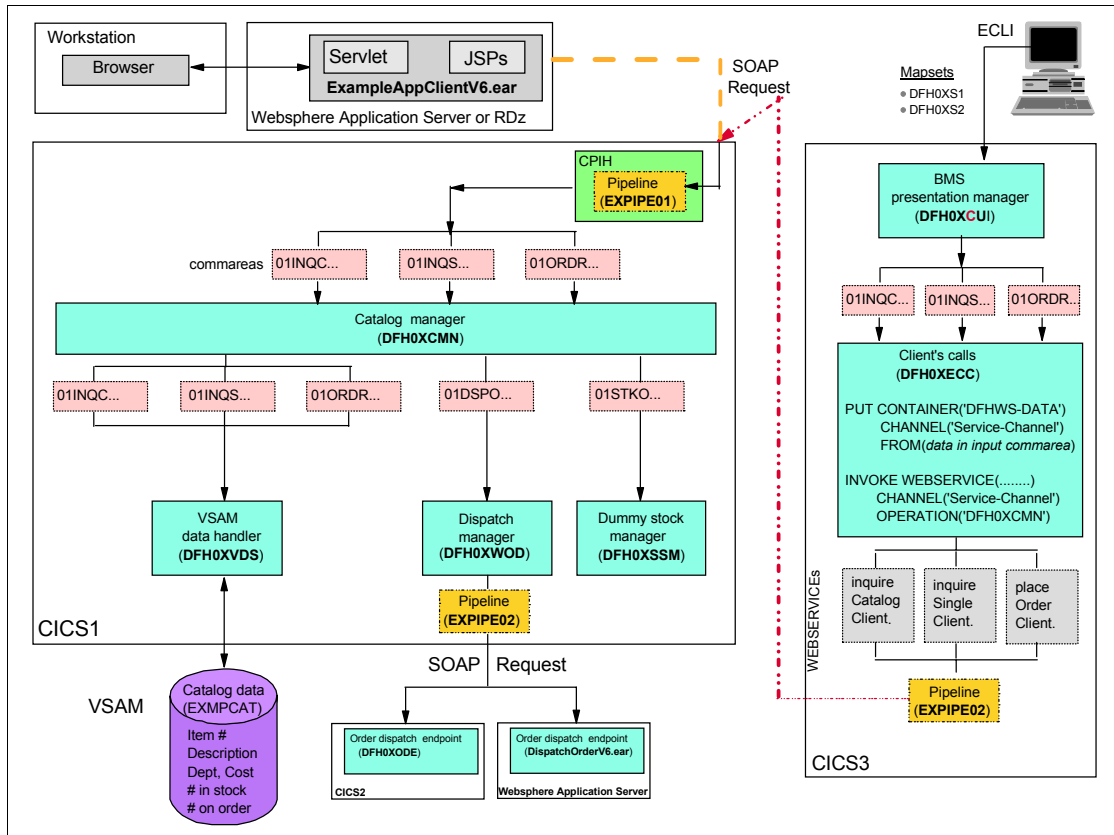


Figure 4-5 Web service client enablement

4.4.2 The CICS Web service client front end

As we have seen in the Web client front end we were able to replace the BMS presentation layer in the base application with a client application that runs on an application server such as Websphere Application Server. This client application makes a Web service call to one of three end points.

CICS also provides a CICS client application that runs within CICS and uses a CICS INVOKE SERVICE call to call the catalog manager application.

This is invoked in CICS by running the transaction ECLI. The configuration transaction ECFG must be used to change the server address and port field to specify where the CICS Web service client will find the catalog manager service end point.

This can also be seen in Figure 4-5.

4.4.3 Order dispatch Web services endpoints

The catalog manager supplies two endpoints for the dispatch order component of the application.

The first version of the dispatch order endpoint is supplied as an ear file (`DispatchOrderV6.ear`) that can be deployed to any of the application server environments listed in 4.4.1, “The Web client front end” on page 81. Using this version of the order dispatch end point demonstrates how CICS can call a Web Service which runs in another application server such as Websphere Application Server.

The external dispatch order end point will be called when the “Outbound WebService?” field is set to **Yes** in the configuration file.

The second version of the dispatch order endpoint is supplied as a CICS service provider application program (`DFH0XODE`). This will be called when the “Outbound WebService?” field is set to **No** in the configuration file.

The dispatch order endpoints can be seen in Figure 4-5 on page 83

4.4.4 Alternative Web service provider configuration

In this configuration, the Web browser client is connected to WebSphere Application Server, in which `ExampleAppWrapperClient.ear` is deployed. In CICS, three wrapper applications (for the inquire catalog, inquire single, and place order functions) are deployed as service provider applications. They link to the base application. This configuration can be seen in Figure 4-6 on page 85.

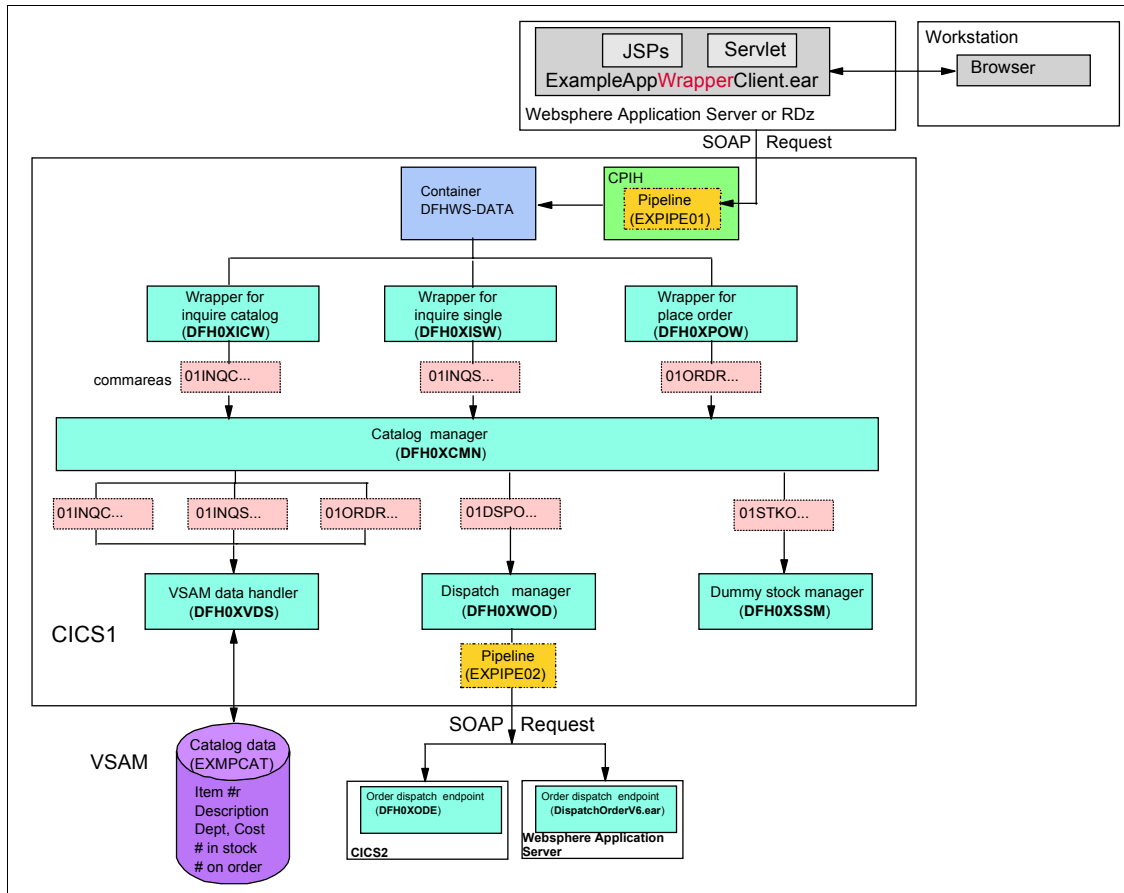


Figure 4-6 Web service provider alternate configuration

4.5 Web services setup

Before you can run the Web service support for the example application, you must create two zFS directories, and create and install several CICS resource definitions.

4.5.1 Creating the zFS directories

Web service support for the example application requires two directories to be created in the zFS (zSeries file system):

- ▶ A shelf directory
- ▶ A pickup directory

Shelf

The shelf directory is used to store the Web service binding files that are associated with WEBSERVICE resources. Each WEBSERVICE resource is, in turn, associated with a PIPELINE. The shelf directory is managed by the PIPELINE resource, and you should not modify its contents directly. Several PIPELINES can use the same shelf directory, as CICS ensures a unique directory structure beneath the shelf directory for each PIPELINE.

The example application uses `/var/cicsts` for the shelf directory.

Pickup

The pickup directory contains the Web service binding files associated with a PIPELINE. When a PIPELINE is installed, or in response to a PERFORM PIPELINE SCAN command, information in the binding files is used to dynamically create the WEBSERVICE and URIMAP definitions associated with the PIPELINE.

A pipeline will read an XML pipeline configuration file at install time. It is therefore also useful to define a directory in which to store these.

4.5.2 Creating the PIPELINE definition

The complete definition of a pipeline consists of a PIPELINE resource and a pipeline configuration file. The file is an XML file that contains the details of the message handlers that will act on Web service requests and responses as they pass through the pipeline.

The example application uses the CICS-supplied SOAP 1.1 handler to deal with the SOAP envelopes of inbound and outbound requests. CICS provides sample pipeline configuration files that you can use in your service provider and service requester.

More than one WEBSERVICE can share a single PIPELINE, so you only have to define one pipeline for the inbound requests of the example application, but you must define a second PIPELINE for the outbound requests because a single

PIPELINE cannot be configured to be both a provider and requester pipeline at the same time.

Note: The zFS entries are case sensitive and assume a default CICS zFS install root of /usr/lpp/cicsts.

You can copy the PIPELINE definition from CICS-supplied group DFH\$EXWS. You must update the following additional attributes (Figure 4-7 and Figure 4-8 on page 88 show our details):

```
STATUS(Enabled)
CONFIGFILE(/usr/lpp/cicsts41/samples/pipelines/basicsoap11provider.xml)
SHELF(var/cicsts)
WSDIR(/usr/lpp/cicsts/cicsts41/samples/Webservices/wsbind/provider/)
```

OBJECT CHARACTERISTICS	CICS RELEASE = 0660
CEDA View Pipeline(EXPIPE01)	
Pipeline	: EXPIPE01
Group	: CATMGR
DEscription	:
STatus	: Enabled Enabled Disabled
Respwait	: Deft Default 0-9999
COnfigfile	: /usr/lpp/cicsts/cicsts41/samples/pipelines/basicsoap11prov
(Mixed Case)	: ider.xml
	:
	:
	:
SHelf	: /var/cicsts/
(Mixed Case)	:
	:
	:
	:
Wsdire	: /usr/lpp/cicsts/cicsts41/samples/webservices/wsbind/provid
(Mixed Case)	: er

Figure 4-7 EXPIPE01 definition

OBJECT CHARACTERISTICS		CICS RELEASE = 0660
CEDA View Pipeline(EXPIPE02)		
Pipeline	: EXPIPE02	
Group	: CATMGR	
DEscription	:	
STatus	: Enabled	Enabled Disabled
Reswait	: Deft	Default 0-9999
COnfigfile	: /usr/lpp/cicsts/cicsts41/samples/pipelines/basicsoap11requ	
(Mixed Case)	: ester.xml	
	:	
	:	
	:	
SHelf	: /var/cicsts/	
(Mixed Case)	:	
	:	
	:	
Wsdir	: /usr/lpp/cicsts/cicsts41/samples/webservices/wsbind/reques	
(Mixed Case)	: ter	

Figure 4-8 EXPIPE02 definition

4.5.3 Creating a TCPIP SERVICE

As the client connects to your Web services over an HTTP transport, you must define a TCPIP SERVICE to receive the inbound HTTP traffic. Figure 4-9 on page 89 shows our example.

1. Use the CEDA transaction to create a TCPIP SERVICE definition to handle inbound HTTP requests.
2. Enter CEDA DEF TCPIP SERVICE(EXMPPPORT) G(EXAMPLE)
Alternatively, you can copy the TCPIP SERVICE definition from CICS-supplied group DFH\$EXWS.
3. Enter the following additional attributes:
 - URM(NONE) PORTNUMBER(*port*)
port is an unused port number in your CICS system.
 - PROTOCOL(HTTP) TRANSACTION(CWXN)

Use the default values for all other attributes.

```

OBJECT CHARACTERISTICS                                     CICS RELEASE = 0660
CEDA View TCpipservice( TCIPISIN )
TCpipservice      : TCIPISIN
GRoup             : CATMGR
DEscription      :
Urm               : NONE
POrtnumber       : 03071                               1-65535
SStatus          : Open                               Open | Closed
PRotocol         : Http                               IIop | Http | Eci | User | IPic
TRansaction      : CWXN
Backlog          : 00001                               0-32767
TSqprefix        :
Host             : ANY
(Mixed Case)     :
Ipaddress        : ANY
SOcketclose     : No                                 No | 0-240000 (HMMSS)
Maxdatalen      : 000032                               3-524288
SECURITY
SSL              : No                               Yes | No | Clientauth Certificate :
(Mixed Case)
PRivacy         :                               Notsupported | Required | Supported
Ciphers         :
Authenticate    : No                               No | Basic | Certificate | AUTOregister
| AUTOMatic | ASserted
Realm           :
(Mixed Case)
ATTachsec       :                               Local | Verify
DNS CONNECTION BALANCING
DNsgroup        :
GRPcritical     : No                               No | Yes
DEFINITION SIGNATURE
DEFinetime      : 08/24/09 04:10:55
CHANGETime      : 08/31/09 23:21:22
CHANGEUsrid     : CICSUSER
CHANGEAGEnt     : CSDApi                               CSDApi | CSDBatch

```

Figure 4-9 EXMPPORT example

4.5.4 Dynamically installing WEBSERVICE and URIMAP resources

Each function exposed as a Web service requires:

- ▶ A WEBSERVICE resource to map the incoming XML of the SOAP BODY and the COMMAREA interface of the program,
- ▶ A URIMAP resource that routes incoming requests to the correct PIPELINE and WEBSERVICE.

Although you can use RDO to define and install your WEBSERVICE and URIMAP resources, you can also have CICS create them dynamically when you install a PIPELINE resource.

Install the PIPELINE resources. Use the following commands:

```
CEDA INSTALL PIPELINE(EXPIPE01) G(SOASDEVWS)
CEDA INSTALL PIPELINE(EXPIPE02) G(SOASDEVWS)
```

When you install each PIPELINE resource, CICS scans the directory specified in the PIPELINE's WSDIR attribute (the pickup directory). For each Web service binding file in the directory (that is, for each file with the .wsbind suffix), CICS installs a WEBSERVICE and a URIMAP if one does not already exist. Existing resources are replaced if the information in the binding file is newer than the existing resources.

If the PIPELINE is later disabled and discarded, all associated WEBSERVICE and URIMAP resources will also be discarded.

If you have already installed the PIPELINE and later update the wsbind files in the WSDIR directory, use the PERFORM PIPELINE SCAN command to initiate the scan of the PIPELINE's pickup directory. CICS will then install any new files. Any files that are already installed will be reinstalled if the file in the directory is newer than the one currently in use.

When you have installed the PIPELINES, the following WEBSERVICES and their associated URIMAPs will be installed in your system:

- ▶ dispatchOrder
- ▶ dispatchOrderEndpoint
- ▶ inquireCatalog
- ▶ inquireSingle
- ▶ placeOrder

The names of the WEBSERVICES are derived from the names of the Web service binding files. The names of the URIMAPs are generated dynamically. You can view the resources with a CEMT INQUIRE WEBSERVICE command, as shown in Figure 4-10 on page 91.

```

I WEBS
STATUS: RESULTS - OVERTYPE TO MODIFY
Webs(dispatchOrder                ) Pip(EXPIPE02)
  Ins Ccs(00000)
Webs(dispatchOrderEndpoint        ) Pip(EXPIPE01)
  Ins Ccs(00000) Uri (£246340 ) Pro(DFHOXODE) Com
Webs(inquireCatalog               ) Pip(EXPIPE01)
  Ins Ccs(00000) Uri (£246341 ) Pro(DFHOXCMN) Com
Webs(inquireCatalogClient         ) Pip(EXPIPE02)
  Ins Ccs(00000)
Webs(inquireCatalogWrapper        ) Pip(EXPIPE01)
  Ins Ccs(00000) Uri (£246344 ) Pro(DFHOXICW) Cha
Webs(inquireSingle                 ) Pip(EXPIPE01)
  Ins Ccs(00000) Uri (£246342 ) Pro(DFHOXCMN) Com
Webs(inquireSingleClient          ) Pip(EXPIPE02)
  Ins Ccs(00000)
Webs(inquireSingleWrapper         ) Pip(EXPIPE01)
  Ins Ccs(00000) Uri (£246345 ) Pro(DFHOXISW) Cha
Webs(placeOrder                   ) Pip(EXPIPE01)
  Ins Ccs(00000) Uri (£246343 ) Pro(DFHOXCMN) Com

```

Figure 4-10 CEMT inquire Web command

For each WEBSERVICE the display shows the following information that is associated with each WEBSERVICE:

- ▶ The PIPELINE name
- ▶ The URIMAP
- ▶ The target program

Note: in this example, there is no URIMAP or target program displayed for WEBSERVICE(dispatchOrder) because the WEBSERVICE is for an outbound request.

Also in this example note that WEBSERVICE(dispatchOrderEndpoint) represents the local CICS implementation of the dispatch order service

4.5.5 Creating the WEBSERVICE resources with RDO

As an alternative to using the PIPELINE scanning mechanism to install WEBSERVICE resources, you can create and install them using Resource Definition Online (RDO).

Note: If you use RDO to define the WEBSERVICE and URIMAP resources, you must ensure that their Web service binding files are not in the PIPELINE's pickup directory.

- ▶ Use the CEDA transaction to create a WEBSERVICE definition for the inquire catalog function of the example application.
 - Enter CEDA DEF WEBSERVICE(EXINQCWS) G(EXAMPLE)
 - Enter the following additional attributes:

```
PIPELINE(EXPIPE01)
WSBIND(/usr/lpp/cicsts/samples
/webServices/wsbind/inquireCatalog.wsbind)
```
- ▶ Repeat the preceding step for each of the functions of the example application shown in Table 4-1.

Table 4-1 Example application functions

Function	WEBSERVICE name	PIPELINE attribute	WSBIND attribute
INQUIRE SINGLE ITEM	EXINQSW	EXPIPE01	/usr/lpp/cicsts/samples /webServices/wsbind /provider/inquireSingle. wsbind
PLACE ORDER	EXORDRWS	EXPIPE01	/usr/lpp/cicsts/samples /webServices/wsbind /provider/placeOrder. wsbind
DISPATCH STOCK	EXODRQWS	EXPIPE02	/usr/lpp/cicsts/samples /webServices/wsbind /requester/dispatch Order.wsbind
DISPATCH STOCK (endpoint optional)	EXODEPWS	EXPIPE01	/usr/lpp/cicsts/samples /webServices/wsbind /provider/dispatch OrderEndpoint.wsbind

4.5.6 Creating the URIMAP resources with RDO

As an alternative to using the PIPELINE scanning mechanism to install URIMAP resources, you can create and install them using RDO.

Note: If you use RDO to define the WEBSERVICE and URIMAP resources, ensure that their Web service binding files are not in the PIPELINE's pickup directory.

- ▶ Use the CEDA transaction to create a URIMAP definition for the inquire catalog function of the example application.
 - Enter CEDA DEF URIMAP(INQCURI) G(EXAMPLE)
 - Enter the following additional attributes:

```
USAGE(PIPELINE)
HOST(*)
PATH(/exampleApp/inquireCatalog)
TCPIService(SOAPPORt)
PIPELINE(EXPIPE01)
WEBSERVICE(EXINQCWS)
```
- ▶ Repeat the preceding step for each of the remaining functions of the example application. Use the names in Table 4-2 for your URIMAPs.

Table 4-2 URIMAP names

Function	URIMAP name
INQUIRE SINGLE ITEM	INQSURI
PLACE ORDER	ORDRURI
DISPATCH STOCK	Not required
DISPATCH STOCK endpoint (optional)	ODEPURI

- Specify the distinct attributes in Table 4-3 for each URIMAP.

Table 4-3 URIMAP attributes

Function	URIMAP name	PATH	WEBSERVICE
INQUIRE SINGLE ITEM	INQSURI	/exampleApp/inquireSingle	EXINQSWS
PLACE ORDER	ORDRURI	/exampleApp/placeOrder	EXORDRWS
DISPATCH STOCK endpoint (optional)	ODEPURI	/exampleApp/dispatchOrder	EXODEPWS

- Enter the following additional attributes (same for all URIMAPs):
 - USAGE (PIPELINE)
 - HOST (*)
 - TCPIP SERVICE (SOAPPOR)
 - PIPELINE (EXPIPE01)

4.5.7 Completing the installation

To complete the installation, install the RDO group that contains your resource definitions.

Enter the `CEDA I G(EXAMPLE)` command at a CICS terminal.

At this point all CICS aspects of the Catalog application should now be installed and fully functional.

4.6 Installing the client application

To run the client application it must be installed into an application server such as Websphere Application Server or into the test environment of Rational Application Developer or Rational Application Developer for System z (RDz)

We installed the supplied client application into the Websphere Application Server 7.0 test client installed with IBM Rational Developer for System z (RDz). This enabled us to run the client and access it from our Web browser. The client code will then use Web services to talk to our CICS system, and we should be able to drive the CICS application with a friendly graphical user interface.

Figure 4-11 on page 95 shows the Websphere test server in RDz 7.5.1.

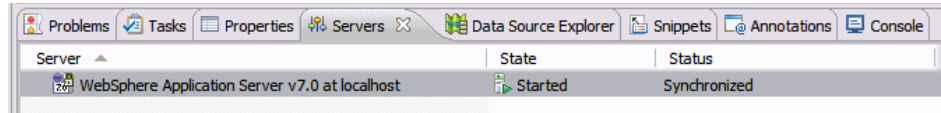


Figure 4-11 Rational Application Developer for System z test environment

4.6.1 FTP the client application

Before we can install the client application we downloaded it from the CICS install directory on UNIX Systems Services (USS). The directory that contains the client ear file is <cics install directory>/samples/webservices/client. We downloaded the ear file using FTP ensuring that it was downloaded to our workstation in binary,

The ear file for the client application is ExampleAppClientV6.ear

4.6.2 Install the client

The client is installed using the Wepshere Admin Console. This can be started either by right-clicking the Websphere Application Server and then selecting **Administration** followed by **Run Administrative Console**

Alternatively, the console can be accessed from a Web browser (as it would be for Websphere Application Server) by entering the following URL:

`http://localhost:9060/ibm/console`

Replace localhost and the port number (if necessary) for your system. After you have logged onto the Administrative Console, the window should look like Figure 4-12.

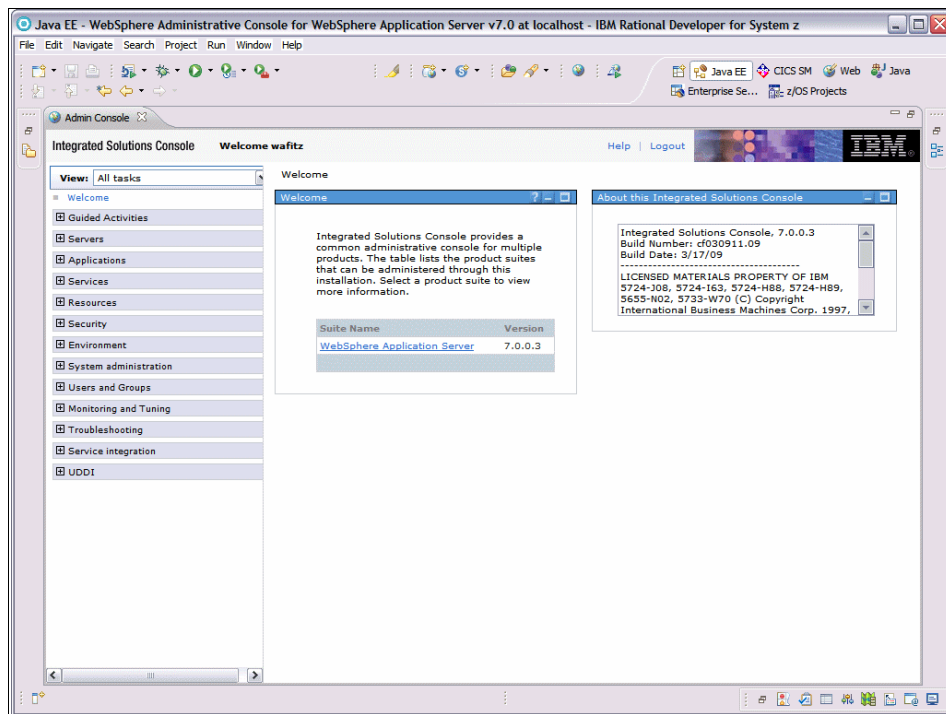


Figure 4-12 Websphere Application Server Admin Console

You must then install the client application in to the Application Server by opening up the Applications tab on the left then selecting **New Applications** and then **New Enterprise Application**. The window should look like Figure 4-13.

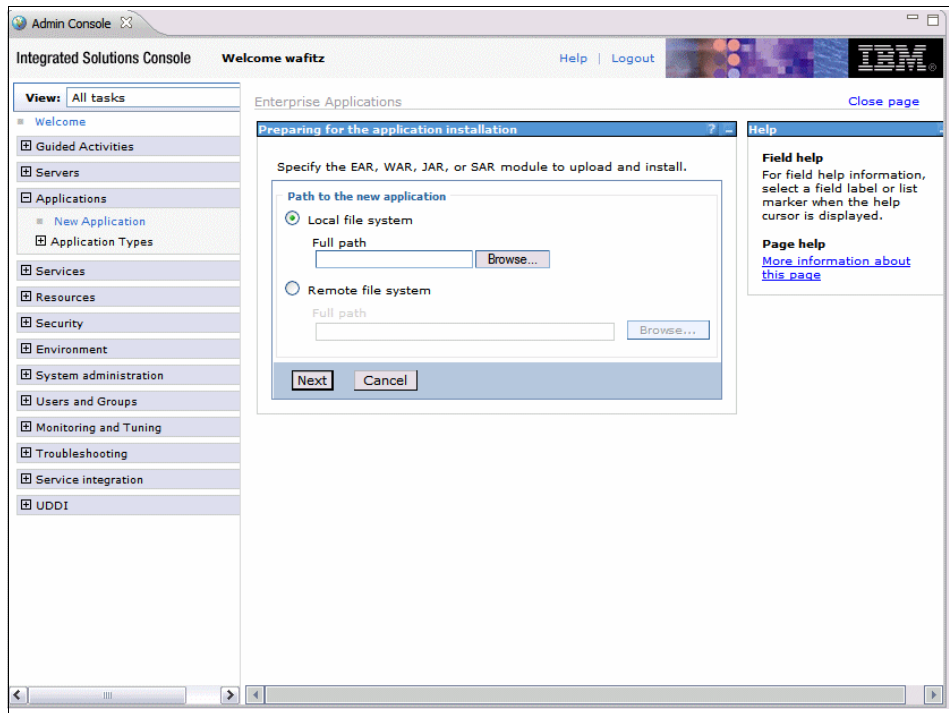


Figure 4-13 Ear file installation

Use the **Browse** button to locate the ear file on your system. We took all the default options when installing the application into Websphere.

4.6.3 Start the client

After the client application is installed successfully the application must be started so it can be called. From the Administrative console open the Applications tab on the left and click **Application Types** followed by **Websphere Enterprise Applications**. You should now see a window similar to Figure 4-14.

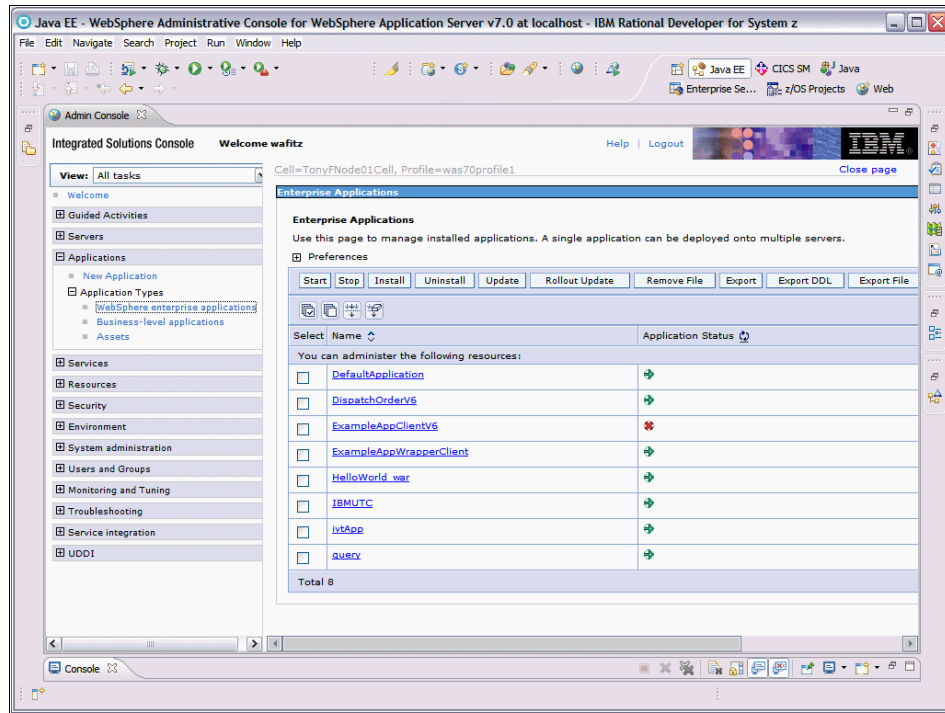


Figure 4-14 Installed applications.

Start the application by clicking the checkbox next to the ExampleAppClientV6 application and click **Start**.

4.6.4 Testing the client

The application is now started so now we can test it from a Web browser.

If you have installed the client application on your local workstation in the Websphere Test Environment, for example, then you should be able to enter the following URL into your Web browser:

`http://localhost:9080/ExampleAppClientV6Web`

Alternatively, the IP address of your workstation can be obtained by issuing the **ipconfig** command in a Windows® command window. In this case replace localhost with the IP address of your workstation.

You should now see a window like Figure 4-15 in your browser.

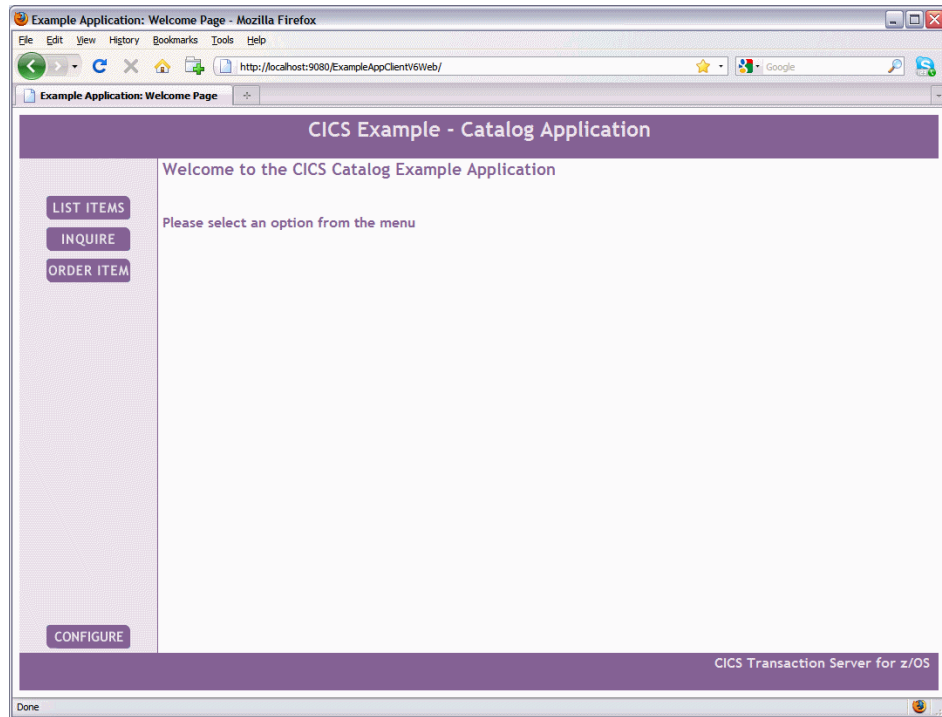


Figure 4-15 Initial client application window

Configure the application

The client, at this point, has no idea where our CICS region is or what port use, so click **CONFIGURE**. This opens a window to send this information to the client. Figure 4-16 shows an example of the CICS configuration screen.

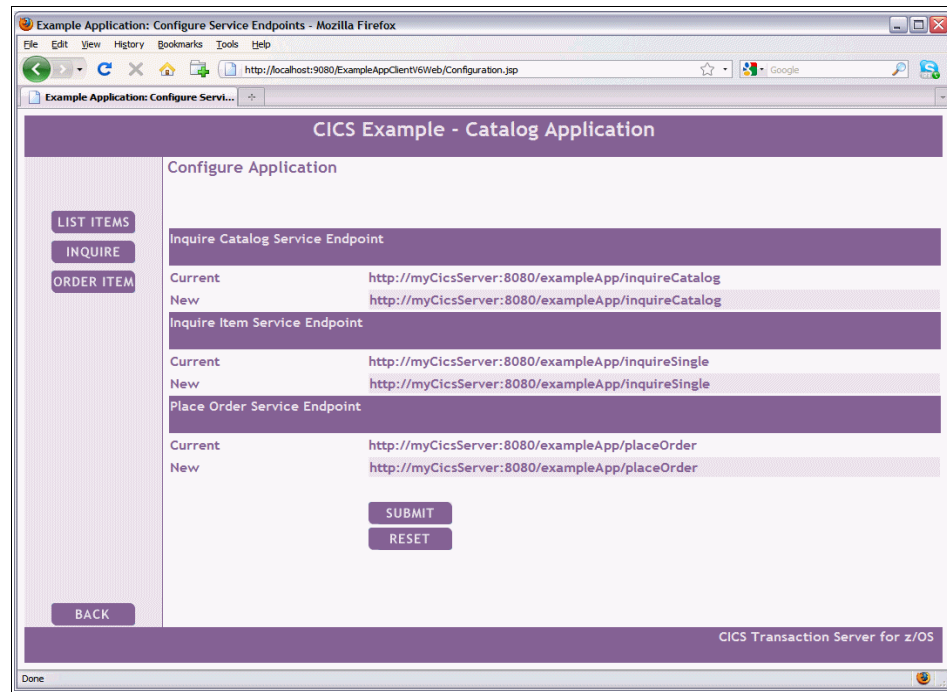


Figure 4-16 CICS Example application configure screen.

Update the three New lines with the correct local host URL and the correct port for your CICS region. To determine this data, go to your CICS region and enter the **CEMT I TCPIPS** command.

This will show you the installed TCPIP services on your region. If you select the correct TCPIP service and expand the details you can determine the correct ipaddress and port number to be used.

After the New lines have been updated click **Submit**. The application is now ready to be tested.

List Items

Click **LIST ITEMS** and wait for the client to talk to your CICS system. You should see the window in Figure 4-17.

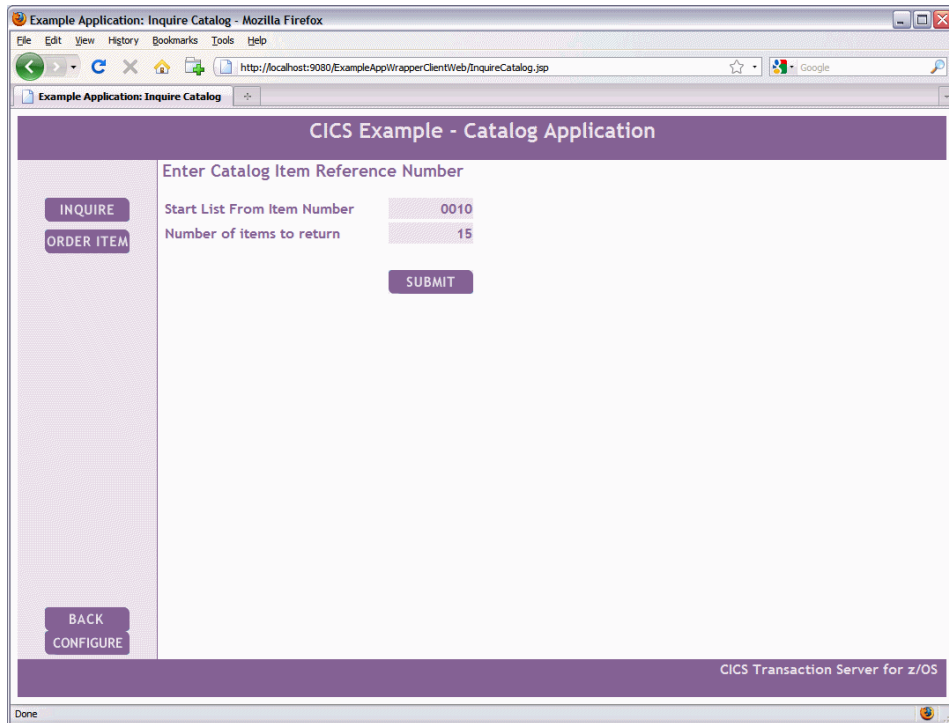


Figure 4-17 Client List Catalog window

Allow the two fields to default and click **SUBMIT**. You should see a window showing a list of items as in Figure 4-18.

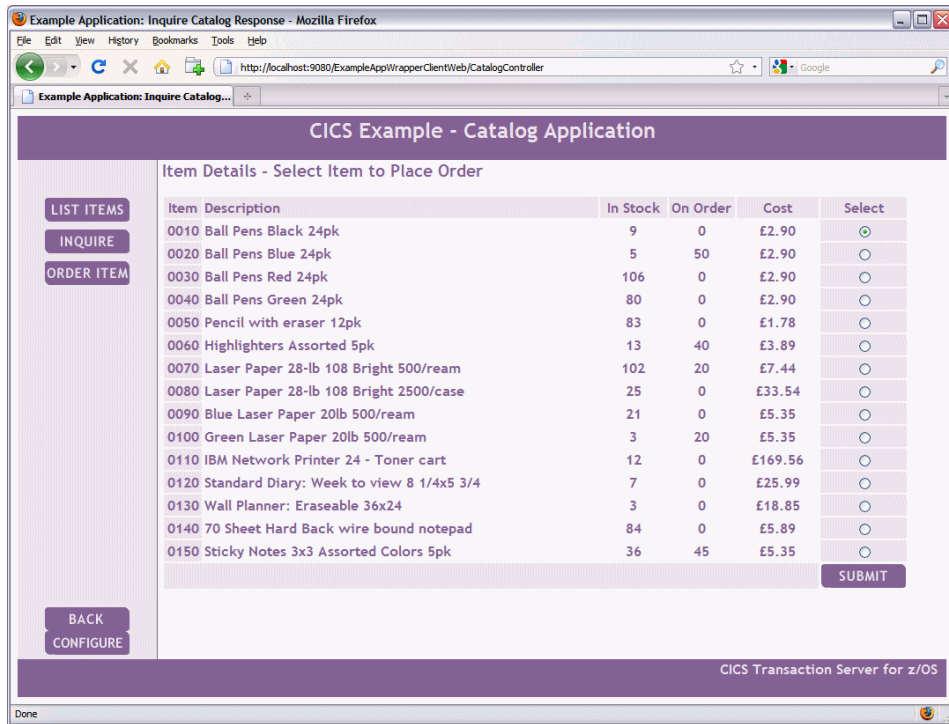


Figure 4-18 Client listing of items

At this point, we have proven that our client is working and making the appropriate Web service requests across our CICS region.

You might want to try some other testing variations at this point.



Rational Developer for System z (RDz)

This chapter introduces you to using the development productivity tool Rational Developer for System z for CICS application development. In Chapter 6, “Exposing the Catalog Sample CICS application as a Web service” on page 137 we demonstrate how RDz can be used to generate and test CICS Web services.

5.1 What is Rational Developer for System z?

Rational Developer for System z (RDz) consists of a common workbench and an integrated set of tools that support application development and maintenance, run-time testing, and rapid deployment of simple and complex applications. It offers an integrated development environment (IDE) with advanced, easy-to-use tools and features to support application development for multiple runtimes like CICS, WebSphere, IMS, and DB2. It helps developers rapidly design, code, and deploy complex applications.

RDz provides support for Cobol, Assembler, PL/I, Java, JEE, C/C++, SQL, and stored procedures.

5.2 RDz and CICS application development

CICS application developers can use IBM Rational Developer for System z to significantly increase productivity and efficiency when creating and maintaining CICS applications. Some of the tasks being performed by developers on a routine basis can be significantly simplified with the help of the features and tools available with RDz.

The following list details some of the tasks that CICS developers can perform efficiently with the help of RDz:

- ▶ View and edit source code with full syntax checking
- ▶ Edit, compile, debug and test application code
- ▶ Handle Web services and XML development
- ▶ Develop BMS maps using visual productivity tools
- ▶ Generate JCL
- ▶ Use the Enterprise Generation Language (EGL)

5.3 Components of RDz

In this section we introduce the different components of RDz.

5.3.1 Workspace

The workspace is a place where all the artifacts related to our work will be stored. It is equivalent to a folder in the file system. We need to specify a workspace as a follow up step of launching RDz. See Figure 5-1 on page 105.

The RDz workspace is a private work area created for the individual developer. It holds the following information:

- ▶ RDz environment, configuration information, and temporary files
- ▶ Projects that developers have created, which include source code, project definition, configuration files and generate files

Resources that are modified and saved are reflected on the local file system. Users can have many workspaces on their local file system to contain different projects that they are working on, or different versions.

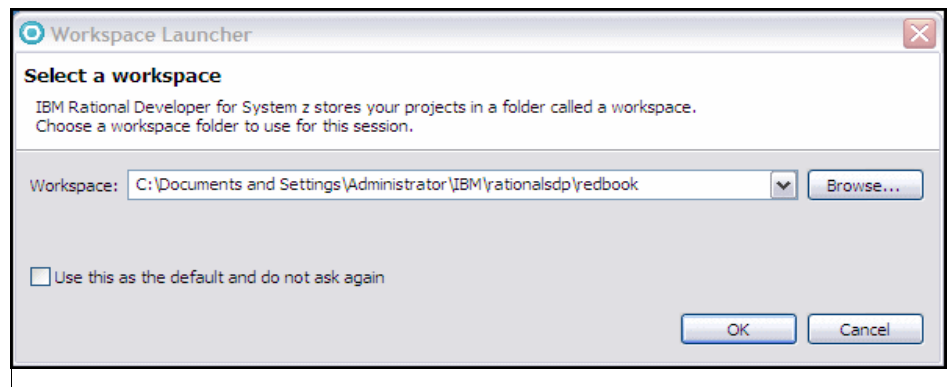


Figure 5-1 Workspace Launcher

5.3.2 Workbench

The workbench is the user interface for RDz. The workbench features an integrated development environment with customizable perspectives that support role-based development. The workbench provides a common way for all members of your project team to create, manage, and navigate resources easily. It consists of interrelated views and editors. (See Figure 5-2 on page 106.) Views provide different ways of looking at the resources you are working on. Editors allow you to create and modify code.

The workbench is made up of several components, such as the Perspective, View, and Editor. See Figure 5-2 for an example workbench window.

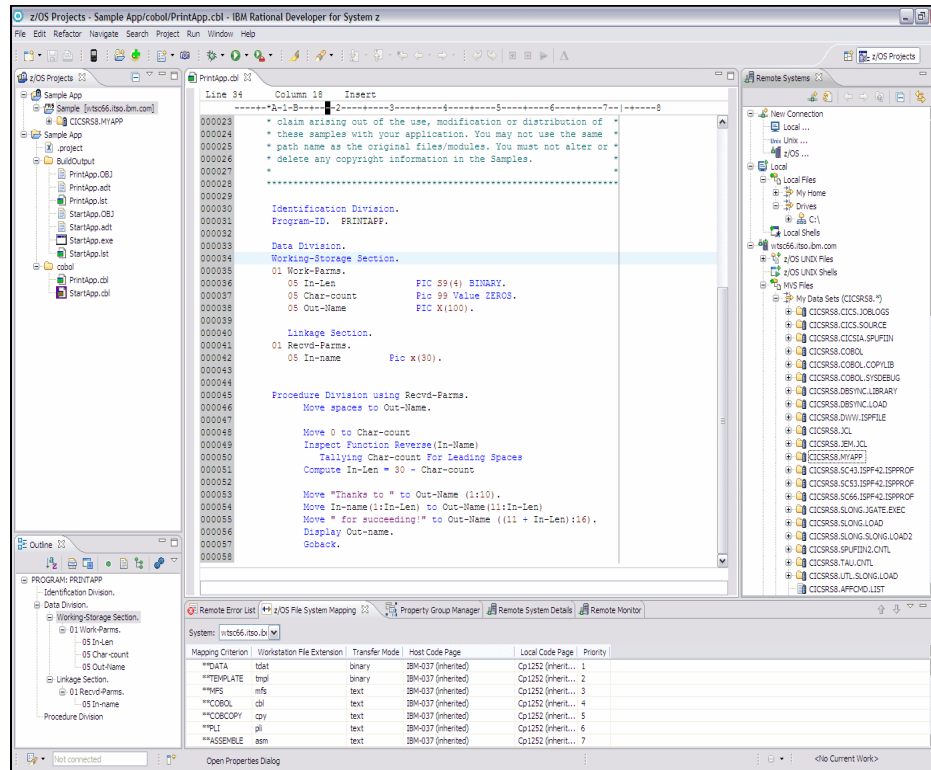


Figure 5-2 Rational Developer for System z Workbench

5.3.3 Perspective

RDz supports a role-based development model, which means that the development environment provides different tools, depending on the role of the user. It does this by providing several different perspectives that contain different editors and views necessary to work on tasks associated with each role.

For each perspective, RDz defines an initial set and layout of views and editors for performing a particular set of development activities. For example, a developer working on System z projects will work with the z/OS Projects perspective. Similarly, a Java programmer responsible for writing, debugging and testing Java code will work using the Java perspective, and so on.

The layout and the preferences in each perspective can be changed and saved as a customized perspective and used again later.

There are two ways to open another perspective:

- ▶ Click the **Open a perspective** icon in the top right corner of the workbench working area and select the appropriate perspective from the list. See Figure 5-3.
- ▶ Select **Window** → **Open Perspective** and select one from the drop-down list. See Figure 5-3.

In both cases, there is also an **Other** option, which when selected displays the “Open Perspective” dialog box that shows a list of perspectives (see Figure 5-3). To show the complete list of perspectives, select the **Show all** check box, if it exists. Here you can select the required perspective and click **OK**.

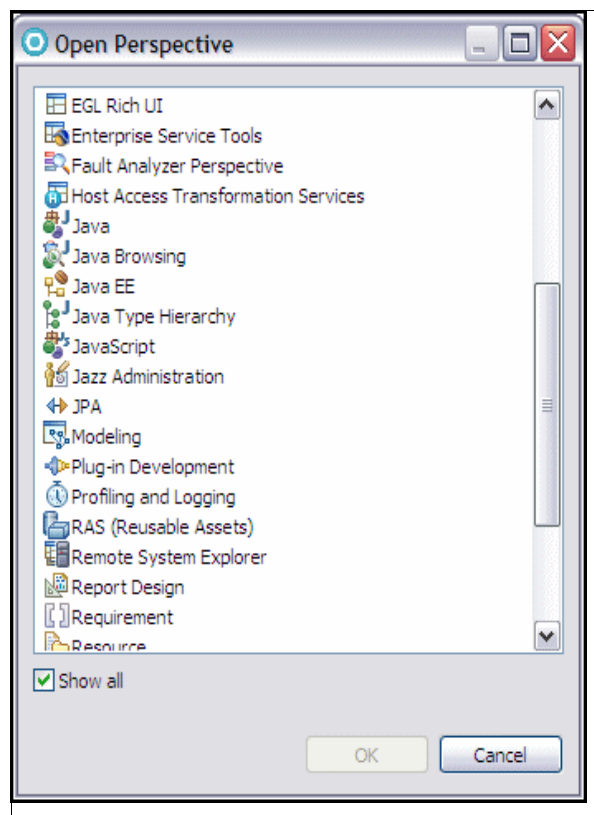


Figure 5-3 List of all perspectives

5.3.4 View

Views provide different presentations of artifacts and resources or ways of navigating through the information in your workspace. For example, the Remote Systems view can help you to connect to z/OS remotely and provides a hierarchical view of the local or remote systems and navigate through folders/datasets hierarchy. From here, you can open files for editing or create, copy and delete datasets. (See Figure 5-4).

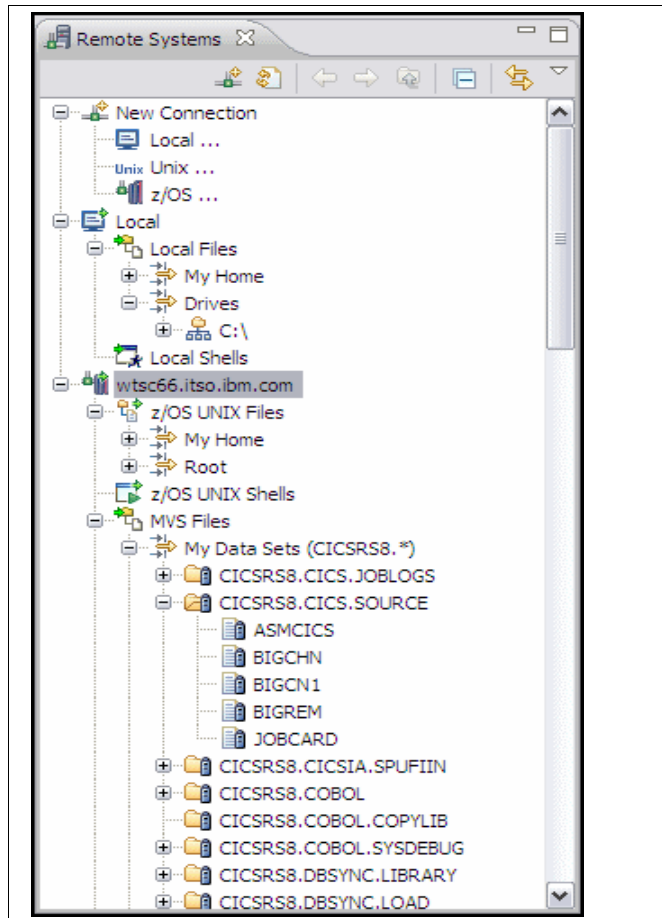


Figure 5-4 Remote Systems view

The Outline view displays an outline of a structured file (COBOL source code in this case) that is currently open in the editor area, and lists structural elements. (See Figure 5-5).

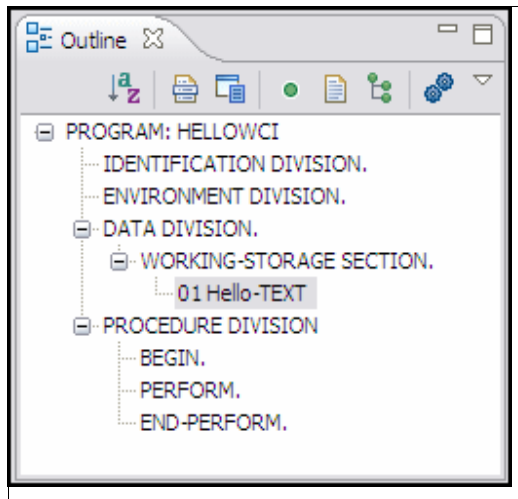


Figure 5-5 Outline view

RDz provides synchronization between views and editors, so that changing the focus or a value in an editor or view can automatically update another. In addition, some views display information obtained from other software products, such as database systems or software configuration management (SCM) systems.

5.3.5 Editor

When you open a file, RDz opens the editor that is associated with that file type. For example, the System z LPEX editor is opened for COBOL copybook files (see Figure 5-6 on page 110), while the Java editor is opened for Java files.

Editors that have been associated with specific file types open in the editor area of the workbench. By default, editors are stacked in a notebook arrangement inside the editor area. If there is no associated editor for a resource, RDz will open the file in the default editor, which is a text editor.

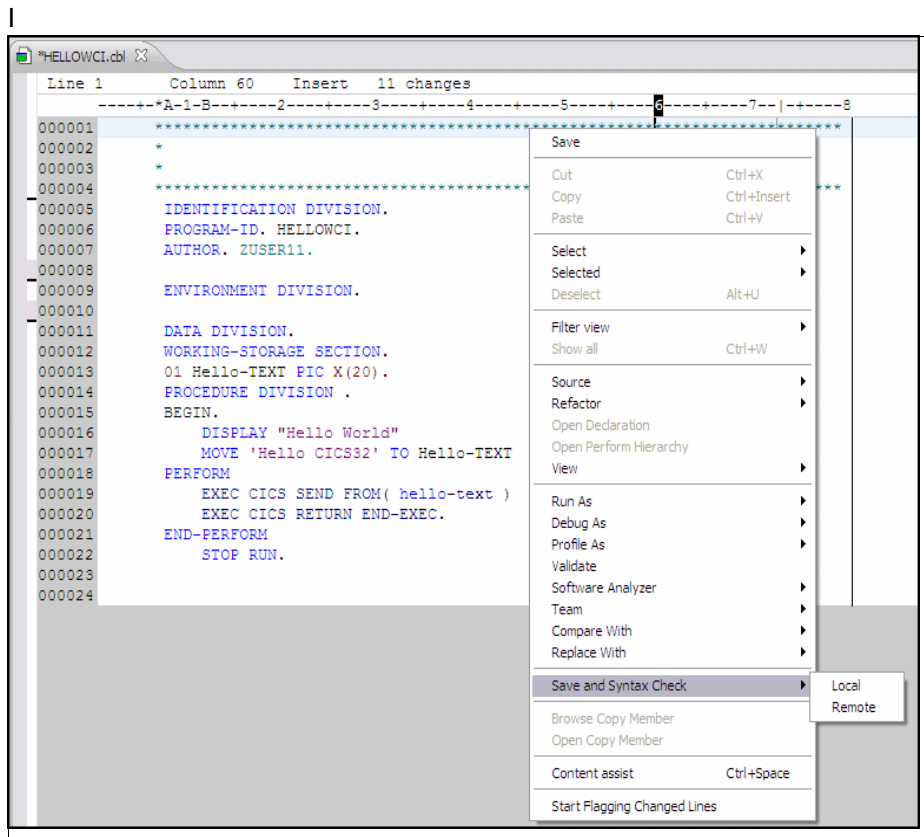


Figure 5-6 System z LPEX editor

5.3.6 Projects and subprojects

A project is the top level of organization of resources in the workbench. It can contain one or more subprojects. A subproject contains files and folders that are grouped into buildable units. Projects are used for building, version management, sharing, testing, and deployment. You can create several different types of projects in RDz, such as COBOL, PL/I, C/C++ projects targeted for CICS, or Web and Java projects. Different types of projects have different structures, different associated builders, and different automatic validation routines.

When you create a project or subproject, you indicate a file system location (folder) to store all associated resources.

A project can be created by navigating to **File** → **New** → **Project**.

Select the type of project from **New Project** window (Figure 5-7).

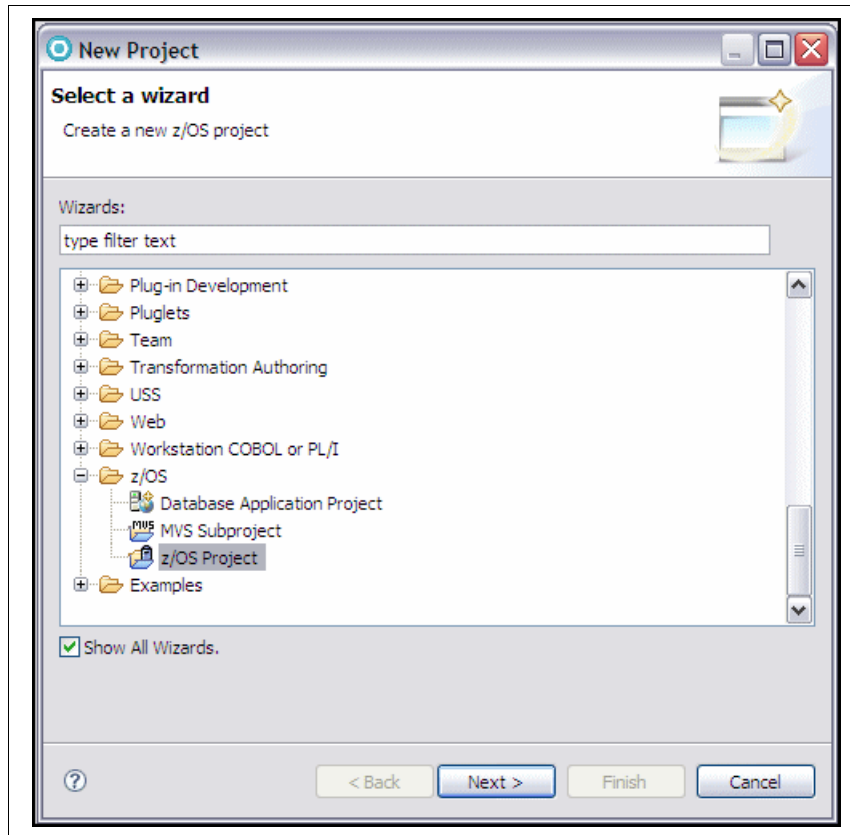


Figure 5-7 New Project window

5.4 Writing your first COBOL Program with RD/z

In this section we write and test our first COBOL program with RDz. With the help of this exercise we will also understand some basic concepts involved in COBOL development with RDz. You may find some differences in screen shots in parts of the exercise depending upon which version of RDz you are using.

1. First, we need to Switch to z/OS Projects perspective. Do that by navigating to **Window** → **Open perspective** → **z/OS Projects**.
2. To create a new cobol sample program, navigate to **File** → **New** → **Other**.

- Expand **Examples** → **Workstation COBOL** and choose **COBOL Sample 1** (see Figure 5-8).

Cobol Sample 1 is sample program provided with RDz, it will create a local z/OS project.

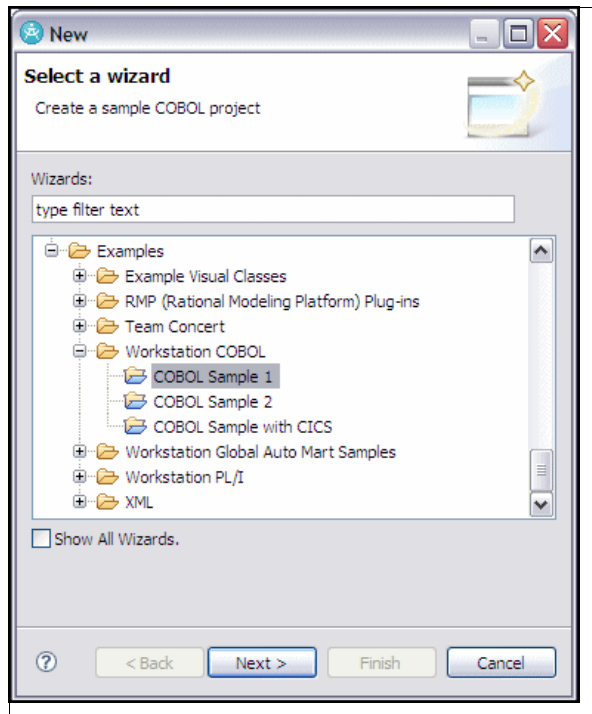


Figure 5-8 Select a wizard

- Click **Next**.

5. Insert name of the z/OS project as **Local Cobol Program**. (see Figure 5-9).
You need to associate a property group with the project. Property group maintains all configuration information required to compile, link, and build the program. We have added a detailed discussion about property groups in the next few pages.

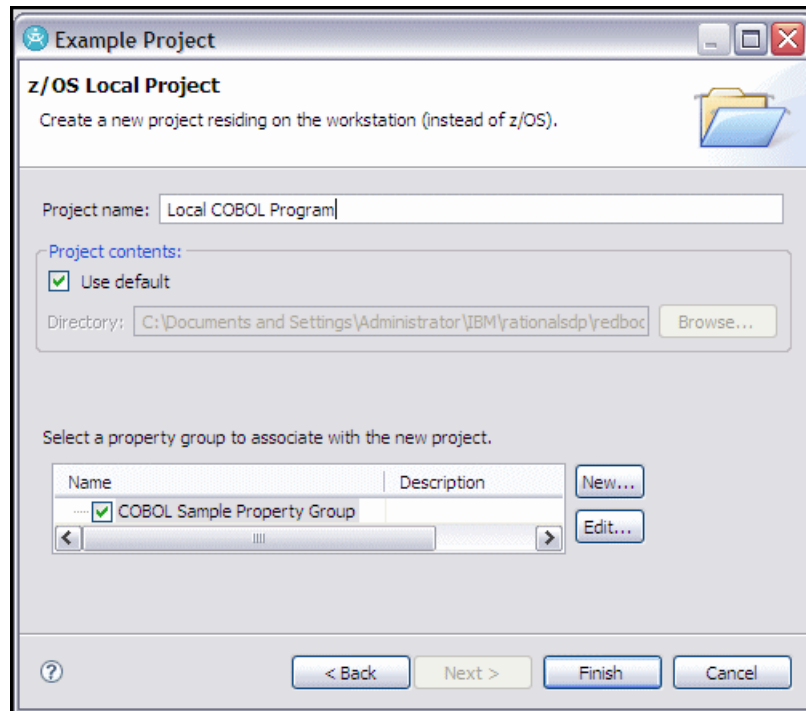


Figure 5-9 z/OS local project

6. Select the **COBOL Sample Property Group** check box as well.
You can also click the **Edit** button in Figure 5-9 on page 113 for checking properties of propertygroup and in case you want to change any parameters.
See Figure 5-10.

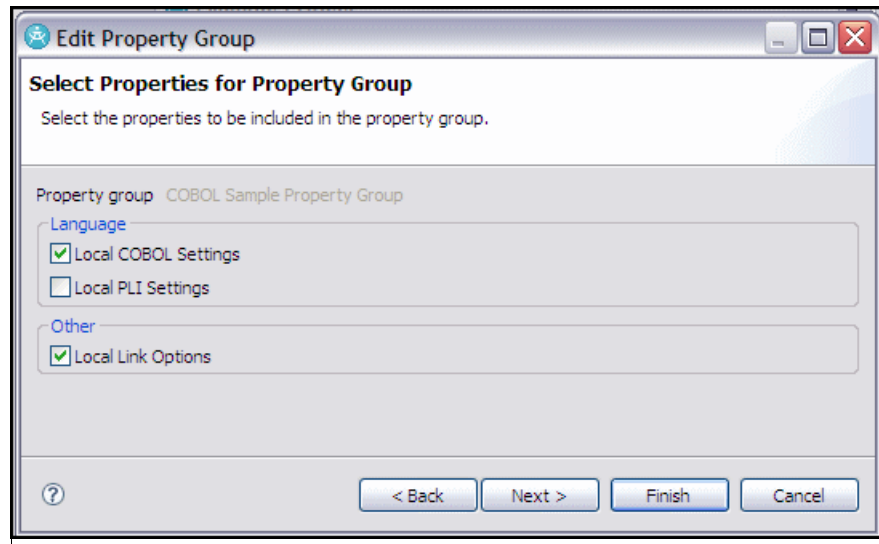


Figure 5-10 Edit property group

5.4.1 Property groups

You can create property groups with property values that can be shared by z/OS projects, subprojects, and resources.

A property group is a set of property values that you define for local COBOL and PL/I projects or specific remote systems. Once defined, the values in a property group can be applied to the z/OS projects, subprojects, and resources that you create on that system. Property groups provide a way to manage resource properties, share them easily across systems, projects, resources, and users, and maintain consistency in your development and build environment.

You can, for example, define a property group with values required for debugging in your environment and apply that property group to your resources when you need to debug the programs in your project or subproject. If you need to change a specific property value, for example, the JCL job card and data set, you can change this property once in the property group and the change is propagated to all resources associated with that property group.

System programmers can create property groups and default property values and make them available to users. When a connection is made to a system, RDz searches the system for system property group and default value files. If these files are found, then those property groups or default values are loaded and can be used.

Figure 5-11 show the “Edit Property Group” window, in which you can change compiler options.

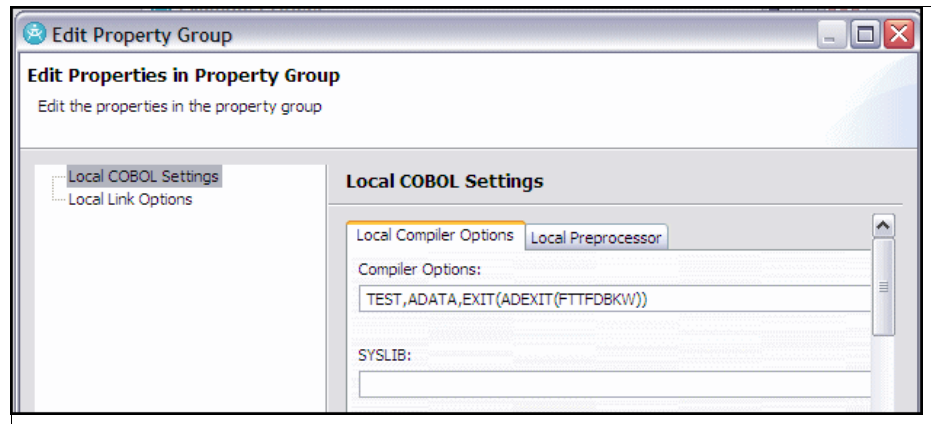


Figure 5-11 Local COBOL settings

5.4.2 Compiler options

- ▶ TEST
This option produces object code that contains the symbol and statement information that enables the debugger to perform symbolic source-level debugging.
- ▶ ADATA
Use this option when you want the compiler to create a SYSADATA file, which contains records of additional compilation information.
- ▶ EXIT
This option allows the compiler to accept user-supplied modules in place of SYSIN, SYSLIB (or copy library), and SYSPRINT. When ADEXIT is specified, the compiler loads the exit module during initialization. The exit module is called for each record written to the SYSADATA data set.
- ▶ SYSLIB
Specify paths to directories to be used in searching for COBOL copybooks when you do not specify an explicit library-name on the COPY statement.

5.4.3 SQL options

Use the SQL compiler option to enable the DB2 coprocessor and to specify DB2 suboptions. The DB2 suboption string that you provide in the SQL compiler option is made available to the DB2 coprocessor. Only the DB2 coprocessor views the contents of the string.

5.4.4 CICS options

Use the CICS compiler option to enable the integrated CICS translator and to specify CICS suboptions. The COBOL compiler makes available to the integrated CICS translator the CICS suboption string that you provide in the CICS compiler option. Only that translator views the contents of the string.

Note that the option for Local Link is **EXE**. This means that when a COBOL program from this project is built, a name.exe is generated instead of a name.d11. Also note the option **/de** means that the program will include debugging information. See Figure 5-12.

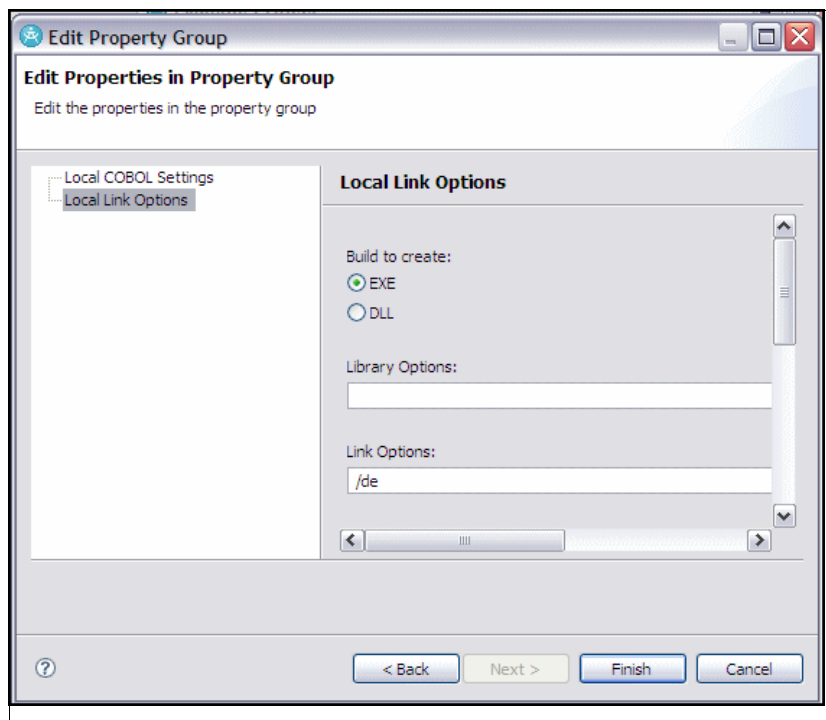


Figure 5-12 Local link options

Project structure in z/OS projects view will have .project file and COBOL folder. See Figure 5-13.

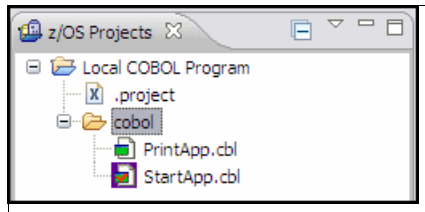


Figure 5-13 z/OS projects - Local cobol project structure

You can build this sample COBOL program by selecting **Project** → **Build All** from the menu bar of the RDz workbench. After successfully building the program, you will see the BuildOutput folder added in the project structure. (See Figure 5-14.)

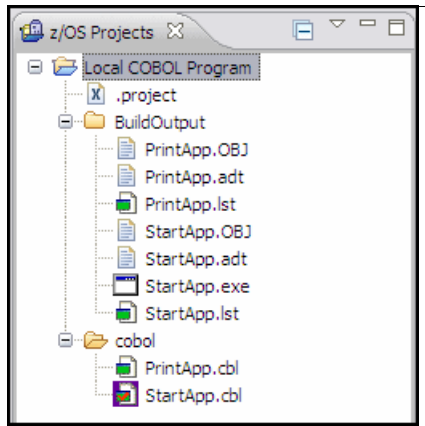
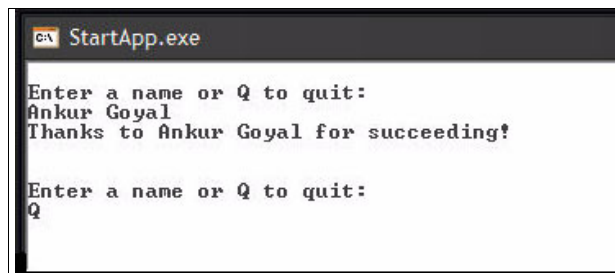


Figure 5-14 Local COBOL program - BuildOutput folder

The BuildOutput folder includes all binaries related to the program. You will also notice a StartApp.exe file there. Double-click this exe file and you will see a window with the COBOL program results will display. (see Figure 5-15)



```
StartApp.exe
Enter a name or Q to quit:
Ankur Goyal
Thanks to Ankur Goyal for succeeding!

Enter a name or Q to quit:
Q
```

Figure 5-15 Sample Cobol program - End screen

You have successfully developed and tested your first COBOL program with RDz.

At the time of writing this IBM Redbooks publication, the latest version of RDz is 7.6. There has been significant additions in the property group management area in this release. The following list details these improvements:

- ▶ Manageable property groups of build/syntax check properties can easily be created and applied to a separate z/OS project. This simplifies the tedious process of setting up and determining which properties apply to which z/OS application projects.
- ▶ The usability of property groups has improved through the use of an editor dialog rather than a wizard dialog. This allows you to view property groups and source code side-by-side, allowing editing of both concurrently.
- ▶ The property group editor includes new validation logic to help developers locate missing information, missing datasets, or improperly formatted inputs.

We discuss Property Group view (5.4.5, “Property Group Manager view” on page 119) and Property Group editor (5.4.6, “Property Group editor” on page 121), two major additions in RDz 7.6, are discussed in the following sections.

5.4.5 Property Group Manager view

The Property Group Manager view provides tools for creating, deleting, editing, importing, exporting, and copying property groups.

To open the Property Group Manager view, follow these steps:

1. From the menu bar, click **Window** → **Show View** → **Other**. In the “Show View” window, expand **z/OS Project Views** (see Figure 5-16).

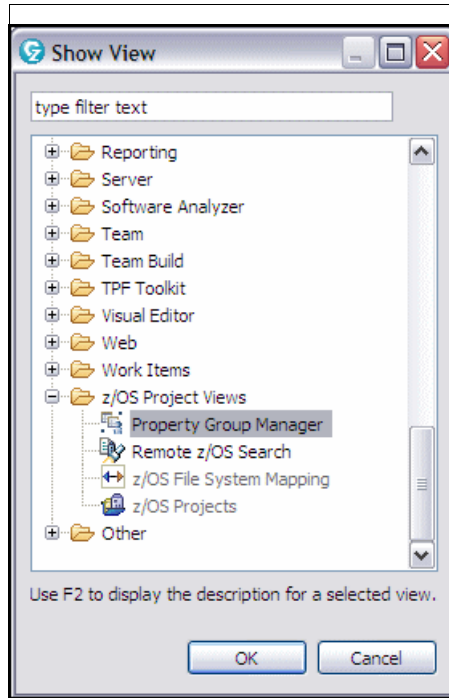


Figure 5-16 Open Property Group Manager view

2. Select **Property Group Manager** and click **OK**. The “Property Group Manager” window will display.(see Figure 5-17)

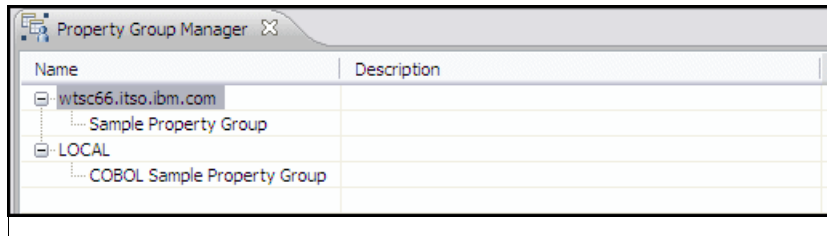


Figure 5-17 Property Group Manager view

3. To create a property group and associate it with a resource from a Property Group Manager, follow the steps below:
 - a. Connect to a remote system.
 - b. Select a connected remote system and create a new property group for it (see Figure 5-18). Add property values to the group and save the group.

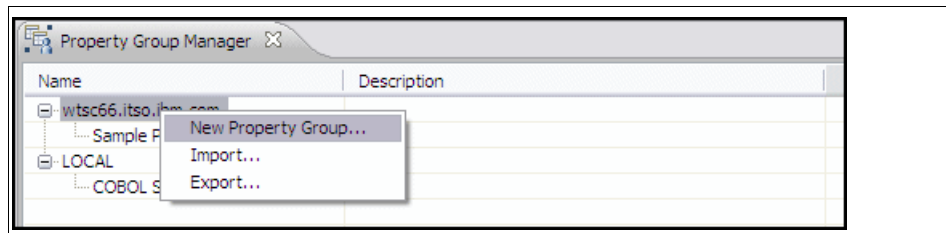


Figure 5-18 Add new property group

- c. Select the resource you want to assign the properties to and associate the group with the resource. Once the property group is created it can be edited, exported, copied or deleted from property group manager itself. (see Figure 5-19)

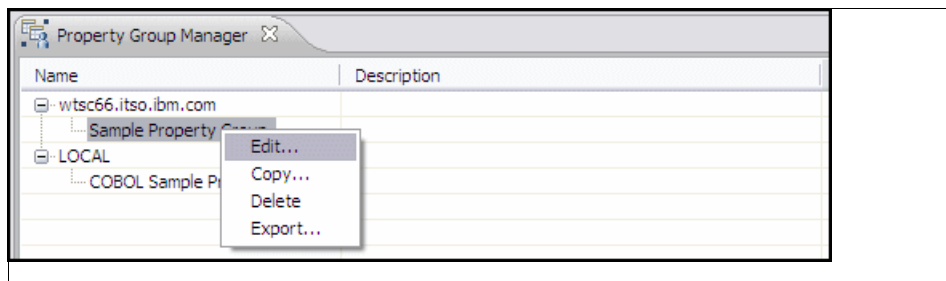


Figure 5-19 Edit, Delete, Export ,or Copy property groups

5.4.6 Property Group editor

To create a new property group or edit an existing property group, the Property Group editor can be used. In RDz 7.6 it has been changed to the form of an editor from a wizard based window.

From a property group editor you can change the property categories or values defined for a property group. Editing a property group enables you to select new or different property categories for the group or to change specific property values for the group. This action is a good way to propagate property changes across multiple projects, subprojects, or data sets because it enables you to make a property change in one place and have it take effect for all resources associated with the property group.

To edit a property group:

1. From the Property Group Manager view, double-click a property group name or select a property group to edit and select **Edit** from the pop-up menu. The the Group Editor will display. See Figure 5-20.

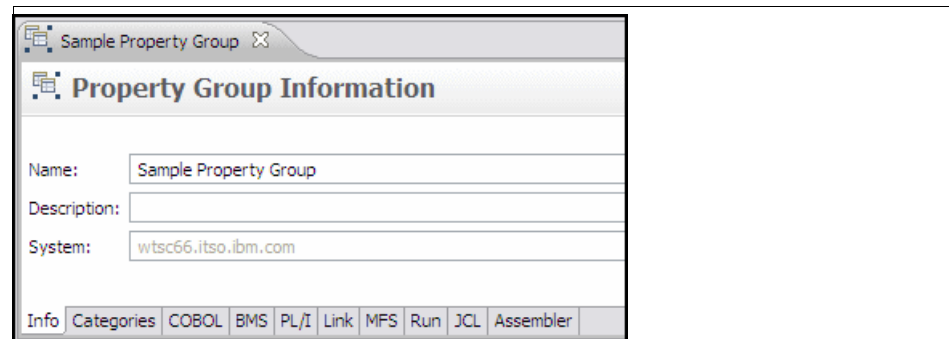


Figure 5-20 Property Group Editor.

You can edit the name and description for the property group.

2. Click the Categories tab and select the check boxes beside the categories for which you want to enter values for this property group. See Figure 5-21 on page 122.

You can choose from the following categories:

- Assembler Settings
- COBOL Settings
- C/C++ Settings
- PL/I Settings
- BMS Settings
- MFS Settings

- JCL Job Card and Data Set
- Link Options
- Runtime Options

The selections you make on this page determine the property tabs that appear in the editor. As you select or clear check boxes, the corresponding tabs are added or removed from the editor. (See Figure 5-21.)

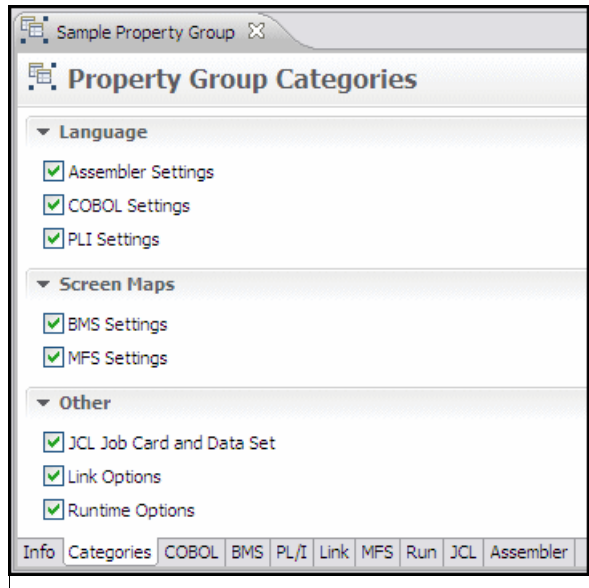


Figure 5-21 Property group editor categories and property tabs

3. Select each property category tab to open the property page for that category.
4. On each property category page, specify property values to be included in the property group.

For fields that take partitioned data sets as their value, you can drag a PDS name from the Remote Systems view, z/OS Projects view, or Remote z/OS Search results view and drop it in the field.

To change the order of PDS names in a field, place the cursor in the field and click Change Order from the pop-up menu. The property group editor checks that the value in a data set field is valid:

- The system for the data set is compared to the system for the property group. If the system for the data set is different, an error message is displayed.
- The property group editor verifies that the value is an MVS data set rather than a PDS member, z/OS UNIX System Services file, or local file.

- If multiple data sets are dropped in a field that accepts only one data set, an error message is displayed. You can also click the Check Data Sets button to check that data sets specified in the step options field exist on the remote system.

When you close the Property Group Editor, RDz prompts you to save your changes.

5.5 Writing your first Java program with RD/z

Perform the following steps to write a Java program with RD/z

1. Switch to Java perspective. Navigate to **Window** → **Open Perspective** → **Other**.
2. Select **Java** from pop-up menu and click **OK**.
3. Navigate to **File** → **New** → **Project**, expand the **Java** folder in the pop-up window and select **Java Project**.
4. Click **Next** and enter project name as HelloRDz, everything else as default.
5. Click **Finish**.
6. Add a new package. Select **File** → **New** → **Package** and enter the package name (all in lower case, as per Java coding conventions) in the pop-up window. Click **Finish**. (See Figure 5-22.)

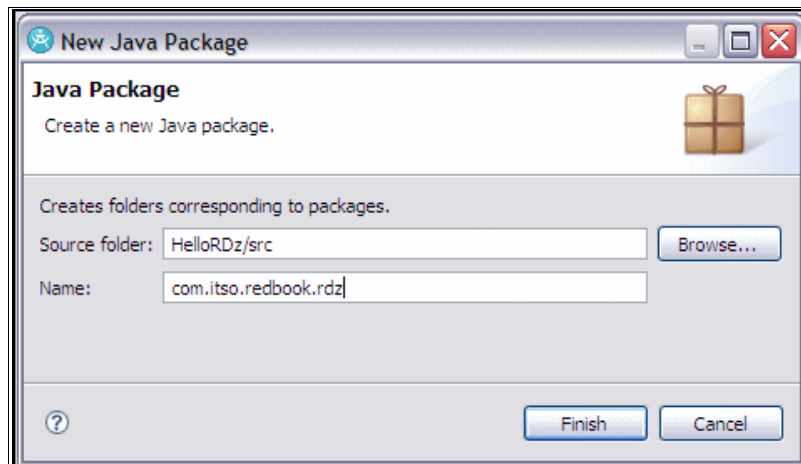


Figure 5-22 New Java package - insert package name

7. Navigate to **File** → **New** → **Class**. Browse for package name `com.itso.redbook.rdz` and insert class name as `HelloWorld`. Select the public static void `main(String args[])` check box. Click **Finish**. (See Figure 5-23.)

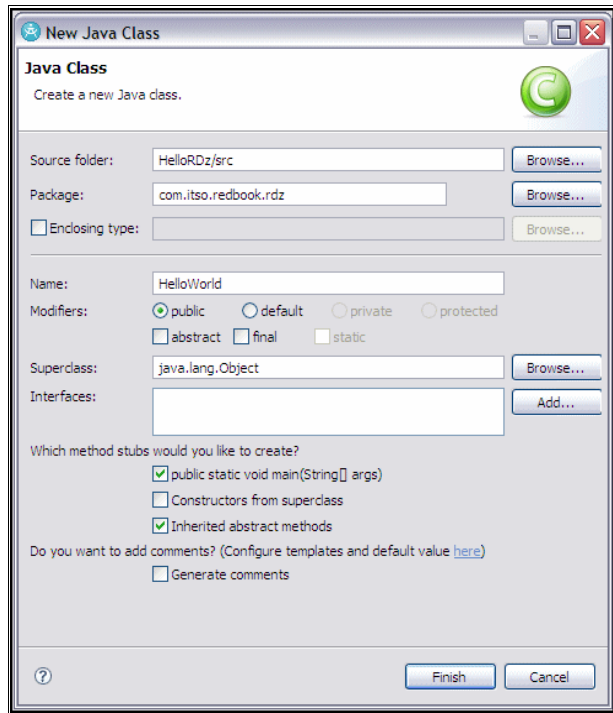


Figure 5-23 Adding a new Java Class

The `HelloWorld.java` source file is opened in Java editor.

8. Replace statement

```
// TODO Auto-generated method stub
```

with

```
System.out.println("Hello to RDz.");
```

The Editor code will now look as shown Figure 5-24.

```
package com.itso.redbook.rdz;

public class HelloWorld {

    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("Hello to RDz.");
    }
}
```

Figure 5-24 Sample Java code for HelloWorld program

9. Select **File** → **Save** to save the program.

10. Select **Run** → **Run As** → **Java Application** to run the program. You should see results in the Console view. (Figure 5-25)

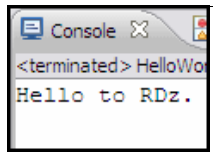


Figure 5-25 Result of HelloWorld program

You have successfully written and executed first Java program with Rational Developer for System z.

5.6 Overview of Debugging with RDz

RDz can also be of great help in debugging programs and applications. We can debug with RDz by switching to the Debug perspective.

The following tasks can be performed for debugging purposes:

- ▶ Set and clear breakpoints at a specific line.
- ▶ Set and clear breakpoints for an error or warning-level error that is based on Language Environment® severities.
- ▶ Run to a breakpoint.
- ▶ Step into a procedure.
- ▶ Step over a procedure.
- ▶ View variable values and change them as you step through the code.
- ▶ View variable values in the context of a larger area of storage.
- ▶ View the call stack.

5.6.1 Supported languages and environments

RDz includes support for debugging many different languages and environments. These are as follows:

- ▶ COBOL
- ▶ PL/I
- ▶ C/C++
- ▶ CICS / IMS
- ▶ Java
- ▶ JavaScript
- ▶ DB2 stored procedures
- ▶ XSL transformations (XSLT)
- ▶ SQLJ
- ▶ Jython Scripts for WebSphere Application Server administration
- ▶ Mixed language applications (for example XSLT called from Java)
- ▶ WebSphere Application Server (servlets, JSPs, EJBs, Web services)

Applications in all these languages and environments can be debugged within RDz using a similar process of setting breakpoints, running the application in debug mode and, within the Debug perspective, stepping through the code to track variables and logic in order to find and fix problems. Furthermore, the interface for debugging within the Debug perspective is intended to be consistent across all these languages and environments.

5.6.2 Local and remote debug

Using RDz, you can debug a wide range of applications in several languages, running either on local test environments or on remote servers, such as CICS, IMS or WebSphere Application Server.

Local

RDz can use the workstation-based debugging engine to debug code in a local project

Remote

This is one area where productivity could increase substantially. With the interactive remote debugging feature, you can run a program on z/OS and view and change data contents, establish breakpoints, jump backward and forward in the execution, recover data exceptions, and more. The remote debugger supports debugging of code that runs in the following z/OS environments:

- ▶ CICS
- ▶ Batch
- ▶ TSO
- ▶ IMS, both IMS Database Manager and IMS Transaction Manager, with or without Batch
- ▶ Terminal Simulator (BTS)
- ▶ DB2 (including stored procedures)
- ▶ WebSphere Application Server

The debugging sessions are cooperative. The remote distributed debugger resides on the workstation and interfaces with the IBM Debug Tool Utilities and Advanced Functions, which runs on the host with your application. The workstation interface communicates with the host z/OS products through TCP/IP.

5.6.3 Basic debugging features and tools

In this section we look at the basic debugging features and tools of RDz.

Views within the Debug perspective

When you run an application in debug mode and reach a breakpoint, you are prompted to switch to the Debug perspective. Although you can debug in any perspective, the Debug perspective includes views that are the most helpful for debugging. Therefore, we recommend that you use the Debug perspective. By default, when debugging, the views shown in the Debug perspective are as follows:

- ▶ Source view
This view shows the file of the source code that is being debugged, highlighting the current line being executed.
- ▶ Outline view
This view contains a list of variables and methods for the code listing shown in the display view.
- ▶ Debug view
This view shows a list of all active threads, and a stack trace of the thread that is currently being debugged.
- ▶ Servers view
This view is useful if the user wants to start or stop test servers while debugging.
- ▶ Variables view
Given the selected source code file shown in the Debug view, this view shows all the variables available to that program and their values. Also, step-by-step debugging variables that change value are highlighted in a different color.
- ▶ Breakpoints view
This view shows all breakpoints in the current workspace and gives a facility to activate/de-activate them, remove them, change their properties, and to import/export a set of them to other developers.
- ▶ Display view
This view allows the user to execute any Java command or evaluate an expression in the context of the current stack frame.
- ▶ Expressions view
During debugging, the user has the option to inspect or display the value of expressions from the code or even evaluate new expressions. The Expressions view contains a list of expressions and values which the user has evaluated and then selected to track.
- ▶ Console view
This view shows the output to System.out.
- ▶ Tasks view
This view shows any outstanding source code errors, warnings or informational messages for the current workspace.
- ▶ Error Log
This view shows all errors and warnings generated by plug-ins running in the work space.

5.7 Establishing Connection to remote Websphere Application Server

There are two editions of RDz available, RDz with Java and RDz with EGL. If you have the first of these variants then you will have access to the Java Enterprise Edition (J2EE) tools described below.

Rational Developer for System z can help you test and debug J2EE applications on local test environments as well as remote servers. To test the application locally you can install the WebSphere Application Server test environment and make a connection to local test environment from the Server view. In this section we list the steps to connect to a remote Websphere Application Server for application testing purposes.

1. Define a new server. See Figure 5-26.

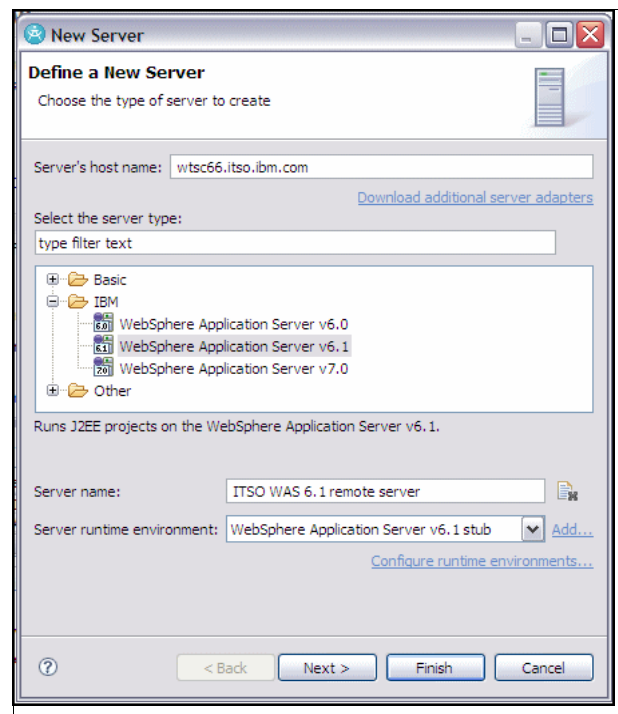


Figure 5-26 Define a new server - need to select as per server availability

2. Update ports for RMI and SOAP. (Figure 5-27)

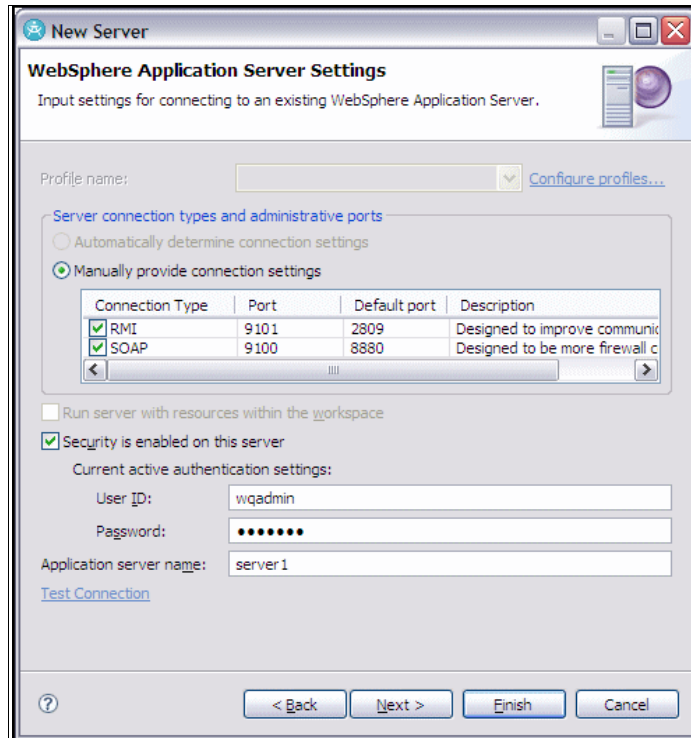


Figure 5-27 New Server definition - define port and security settings

3. Click **Finish**.

Once you finish with the new server wizard you can see the status of the server in the Servers view (Check Figure 5-28). The Server state shows as **Started**. We have established a connection to the remote server for application development and testing purposes.

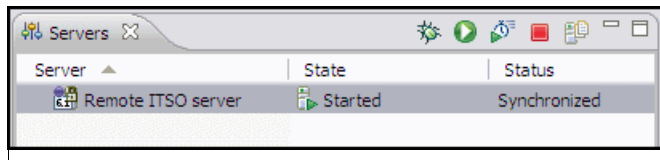


Figure 5-28 Screen showing servers - Server is started

In case you want to update the setting of a server connection configuration you can double click the server in the Servers view, which opens the overview page. The Server connection configuration information can be updated here. (See Figure 5-29.)

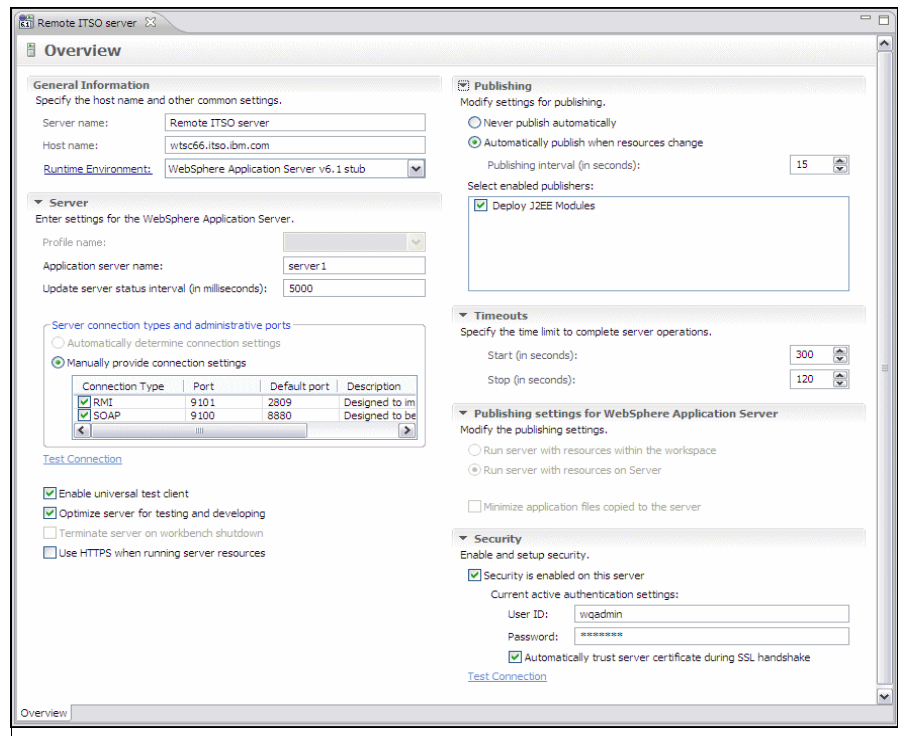


Figure 5-29 Overview screen - server connection configuration

To run administration console on the application server right-click the server in the Servers view and select **Administration** → **Run administrative console**. (See Figure 5-30.)

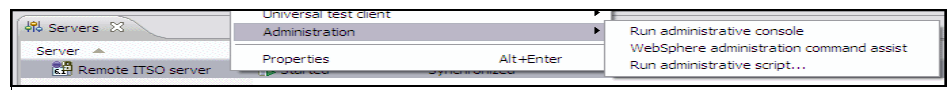


Figure 5-30 Server - launch administration console

You will see the administration console opened in RDz and if security is enabled at the remote WebSphere Application Server it will prompt you to provide a user ID and password. (See Figure 5-31.)

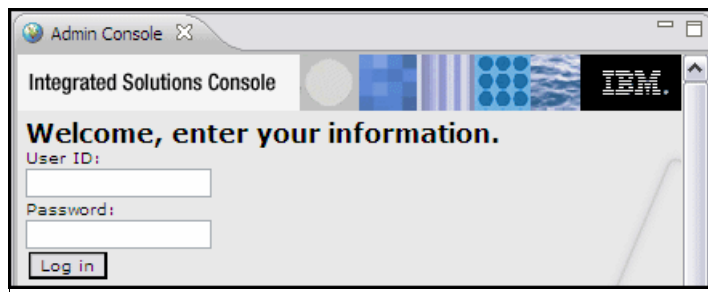


Figure 5-31 Sign on screen - Administration console

After successfully establishing a connection to a remote server you can also perform various administrative tasks on server, provided you have administrative access.

5.8 Import and Export EAR/WAR files

In this section we discuss importing and exporting of EAR/WAR files. This function is usually required by developers to export these archive files and then deploy or test it on application server. Perform the following steps to import an ear file to RDz.

1. Switch to the Java EE perspective. Navigate to **Window** → **Open Perspective** → **Others** and select **JavaEE** from the window.

You will see the Project Explorer view in your workbench.

2. Select **File** → **Import**. Expand the Java EE folder on the window and select **EAR file**. (See Figure 5-32.)

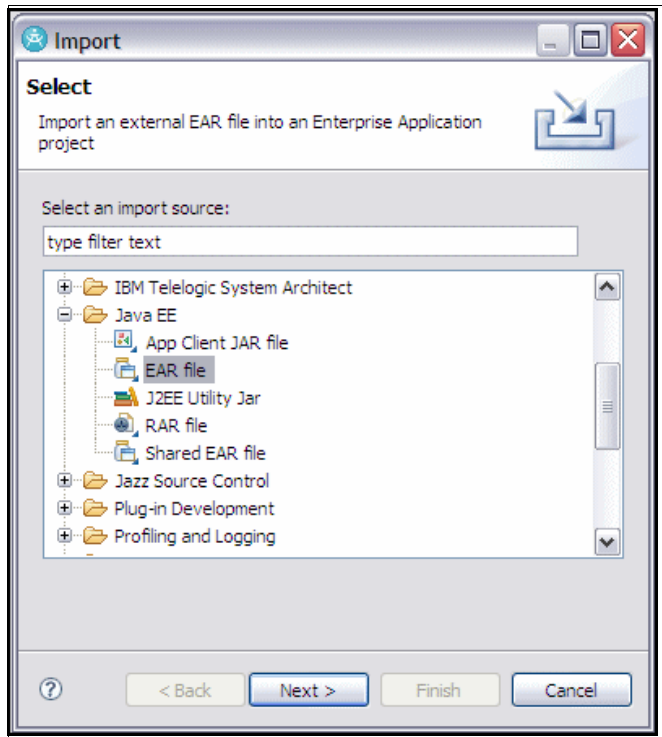


Figure 5-32 Import screen - EAR File.

3. Click **Next**. The window in Figure 5-33 will display.

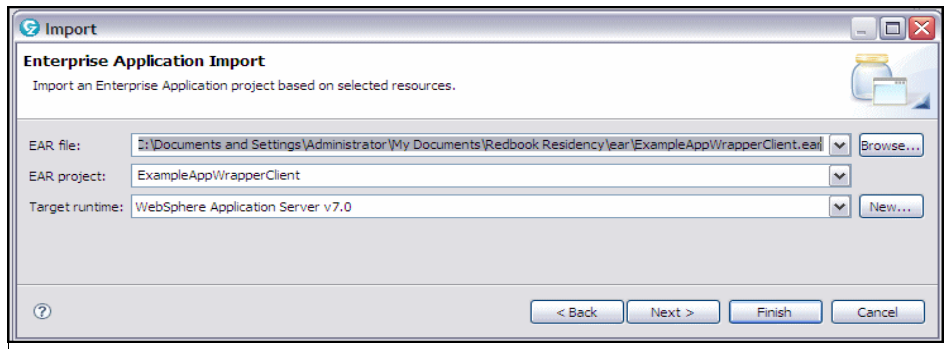


Figure 5-33 Browse for ear file location

4. Browse for the ear file location and specify the target application server. Click **Next**.
5. Select any utility jars or web libraries required related to the ear file. Click **Next**. The window in Figure 5-34 will display.

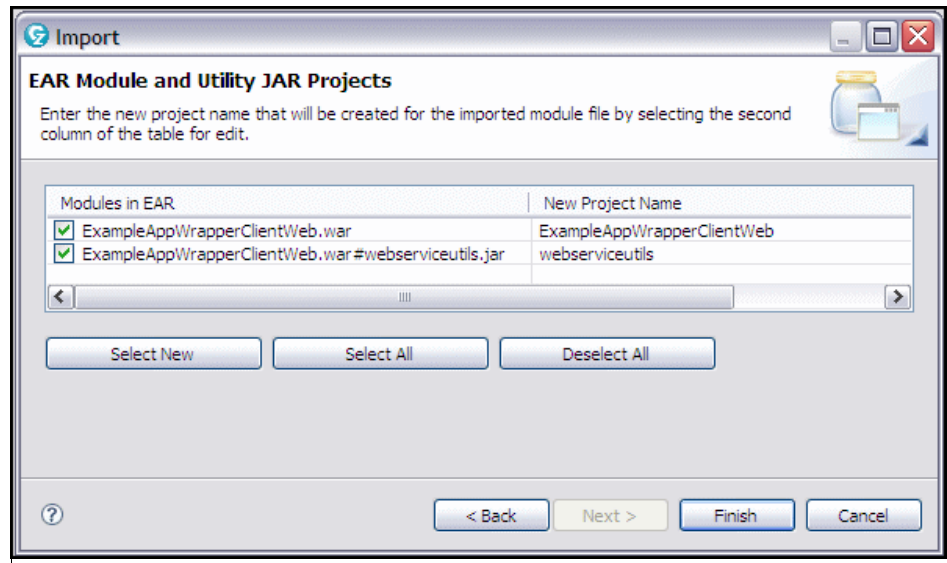


Figure 5-34 List of all utility jars or modules as part of EAR file.

6. Click **Finish**. The ear file and related modules will be imported. You can browse for all these modules in Project Explorer. (See Figure 5-35.)

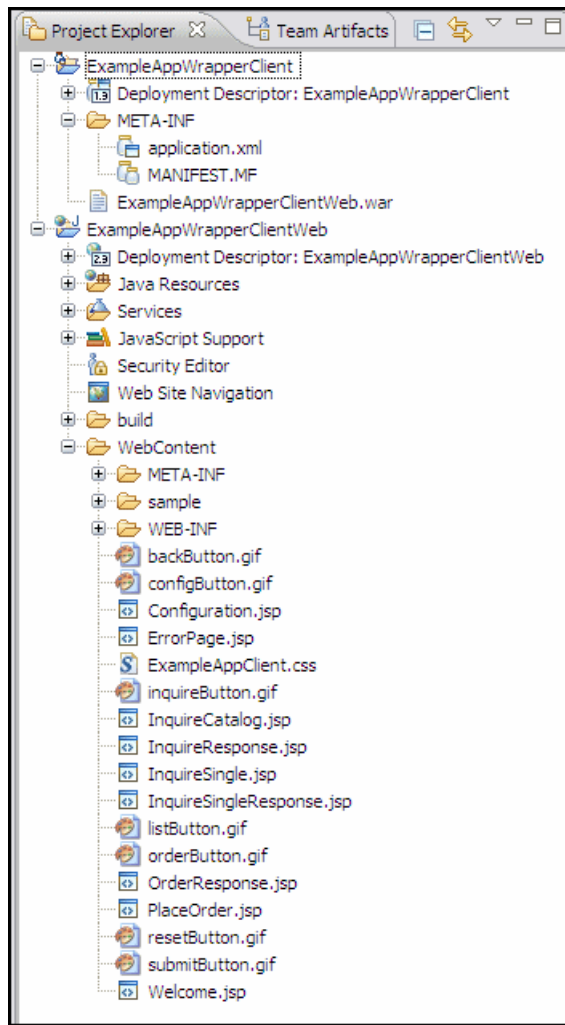


Figure 5-35 - EAR file project hierarchy / structure

Similarly you can export any Web project or enterprise Java project as a WAR or EAR file. Right-click any project in Project Explorer and select **Export** from the context menu. Select the appropriate format of the project being exported and specify a destination as a path to the location where you want to save the exported project. Click **Finish**.

5.9 Summary

In this chapter we have introduced RDz, an IDE for System z application development and maintenance. We have observed different features, components, and tools available as part of RD/z for editing, debugging, and testing of COBOL as well as Java/J2EE applications.



Exposing the Catalog Sample CICS application as a Web service

This chapter demonstrates how an existing CICS application can be exposed as a Web service. The focus is on how to:

- ▶ Create Web service resources
- ▶ Set up the CICS runtime infrastructure
- ▶ Test the Web service provider

The CICS Catalog Manager Application is used as an example. It is assumed that you have completed Chapter 4, “CICS catalog manager example application” on page 73.

6.1 Introduction

In principle, there are two ways to expose a CICS program as a Web service:

- ▶ Use the Web Services Assistant (or RDz) to expose an existing application as a Web service with little or no application changes.
- ▶ Write programs that interact directly with the CICS pipeline and that handle the XML natively.

The second scenario is an advanced concept that usually involves more effort. As such it is only discussed at the end of the chapter.

Figure 6-1 gives a basic overview of a Web service scenario. It assumes an existing CICS program (scenario A in Figure 6-1), which is partitioned to ensure a separation between the communication (or presentation) logic and business logic. Access to the business logic is performed using commareas and EXEC CICS LINK. The application is ideally structured for reuse of the business logic in a provider mode Web service.

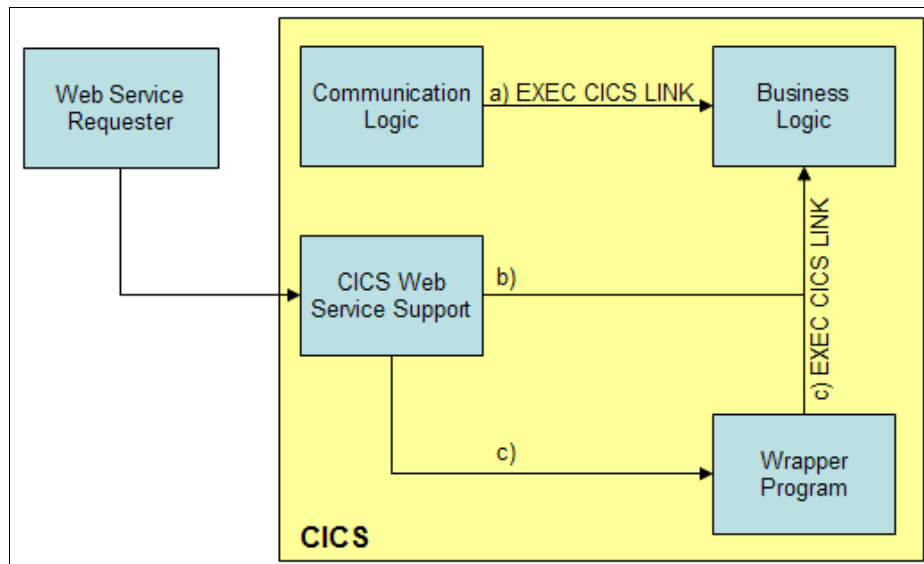


Figure 6-1 CICS as Web service provider: basic overview

In most cases, you can deploy the business logic directly as a Web service as shown in scenario B. You use DFHLS2WS or RDz to create the WSBIND file and deploy it to CICS. This is the technique previously referred to as bottom-up Web service enablement.

This figure shows the parts of the CICS Catalog Manager Application that you will use in this chapter. You will focus on the `inquireSingle` function, which returns a single item from the catalog upon request. The request ID is `01INQS`. The original application uses a BMS map as graphical user interface.

In this chapter we create the Web service enablement artifacts and set up CICS Web Service Support with them. A first approach (6.3.1, “Using the CICS Web Services Assistant” on page 142) uses the Web Services Assistant to create `inquireSingleSelf.wsbind` and `inquireSingleSelf.wsdl`, which are equivalent to `inquireSingle.wsbind` and `inquireSingle.wsdl` (provided with the example).

In 6.3.2, “Use Rational Developer for System z” on page 148, we show how to create the enablement components using the Rational Developer for System z. We create `inquireSingleDriverSelf.wsbind`, `inquireSingleDriverSelf.wsdl`, and the converter and driver files `CMNISD01`, `CMNISI01`, and `CMNISO01`.

In 6.3.1, “Using the CICS Web Services Assistant” on page 142 we focus on best practice and demonstrate how to optimize the WSDL file and work with a wrapper program. You will learn how the provided `inquireSingleWrapper.wsbind` file and the wrapper copy books `DFH0XWC3` and `DFH0XWC4` have been created and how to write a wrapper program using those copybooks.

In 6.4, “Testing the Web service” on page 156, we test those Web services using a Web service client in the Rational Developer for System z test environment. You will also learn how to create your own client from a WSDL file using the Rational Developer.

6.2 Install the provider mode resources

To set up the CICS Web service runtime, you need both a `TCPIP SERVICE` and a `PIPELINE` resource. We considered the installation of these resources in Chapter 4, “CICS catalog manager example application” on page 73.

When you are ready to install `WSBind` files (as discussed in the subsequent sections), do so by following these instructions:

1. Copy the `WSBind` file into the `WSDIR` directory for the `PIPELINE`.
2. Cause CICS to scan the `WSDIR` directory by issuing the command:

```
CEMT PERFORM PIPELINE(EXPIPE01) SCAN
```

3. To check whether your Web service was installed properly, use the following command:

```
CEMT INQUIRE WEBSERVICE
```

The inquireSingleSelf Web service which will be described shortly should appear as shown in Figure 6-3.

```
INQUIRE WEBSERVICE
STATUS: RESULTS - OVERTYPE TO MODIFY
Webs(inquireSingleSelf ) Pip(EXPIPE03)
    Ins Uri($509340 ) Pro(DFHOXCMN) Cha Xopsup Xopdir
```

Figure 6-3 Inquire the created WEBSERVICE resource

4. Note that a URIMAP resource was assigned to this WEBSERVICE. Inquire on this URIMAP using the identifier. For this example:

```
CEMT INQUIRE URIMAP($509340)
```

This should yield a result similar to that shown in Figure 6-4.

```
INQUIRE URIMAP($509340)
STATUS: RESULTS - OVERTYPE TO MODIFY
Uri($509340 ) Pip Ena Http
    Host(* )
Path(/exampleApp/inquireSingleSelf )
```

Figure 6-4 Inquire the created URIMAP resource

6.3 Create the provider mode deployment artifacts

This section shows two different approaches to creating the WSBind file used to enable an existing CICS COBOL program as a Web service. As an example, we use the inquire single operation of the CICS catalog manager example application, which returns a single item from a catalog.

We consider two approaches to Web service enablement. First we look at using the Web Services Assistant, then we consider the use of RDz. We see that using RDz simplifies the steps involved.

We consider both the bottom-up and meet-in-the-middle scenarios.

6.3.1 Using the CICS Web Services Assistant

The Web Services Assistant was introduced in chapters 2 and 3. It can be used to generate the WSBIND file that, in turn, contains the conversion instructions used by CICS to transform SOAP messages into application data.

In provider mode, this will usually involve bottom-up enablement through DFHLS2WS. However, advanced users might consider modifying the generated WSDL resource as part of a meet-in-the-middle scenario. Both scenarios are discussed in the following sections.

The bottom-up approach with DFHLS2WS

As shown in Chapter 3, “Development approaches” on page 61, the bottom-up approach implies creating a WSDL file from an existing application. This is done using DFHLS2WS.

There is however a complication. The CICS catalog sample application makes use of COBOL ‘REDEFINES’ statements, and these are not supported by DFHLS2WS. Therefore, there is an additional step required that would not normally be needed, and that is to simplify the language structure.

A simplified copybook is provided in DFHOXCP4.

Some example JCL for calling DFHLS2WS is shown in Example 6-1. This JCL is suitable for use with CICS TS V4.1. For earlier versions of CICS you should change the USSDIR and MAPPING-LEVEL accordingly. You will also have to modify the dataset and UNIX System Services (USS) directory names to whatever is suitable at your site.

Example 6-1 DFHLS2WS JCL

```
//LS2WSIS JOB (999,P0K),'CICS LS2WS TOOL',MSGCLASS=T,
//          CLASS=A,NOTIFY=&SYSUID,TIME=1440,REGION=0M
//*
// JCLLIB ORDER=CICSTS41.CICS.SDFHINST
//*
// SET QT='''
//LS2WS EXEC DFHLS2WS,
//  JAVADIR='java/J6.0',
//  USSDIR='cicsts41',
//  PATHPREF=''
//INPUT.SYSUT1 DD *
  PDSLIB=//CICSTS41.CICS.SDFHSAMP
  PGMNAME=DFHOXCMN
  LANG=COBOL
  PGMINT=COMMAREA
```



```
REQMEM=DFH0XCP4
RESPMEM=DFH0XCP4
MAPPING-LEVEL=3.0
LOGFILE=/u/cicsrs9/provider/wsbinding/inquireSingleSelf.log
WSBIND=/u/cicsrs9/provider/wsbinding/inquireSingleSelf.wsbinding
WSDL=/u/cicsrs9/provider/wsd1/inquireSingleSelf.wsd1
URI=exampleApp/inquireSingleSelf
*/
```

These are the input parameters:

- PDSLIB** The library containing DFH0XCP4.
- PGMNAME** The name of the program for the CICS catalog manager example application DFH0XCMN.
- LANG:** Specifies the programming language DFH0XCP4 is written in
- PGMINT** Describes the program input. DFH0XCMN uses a COMMAREA.
- REQMEM** and **RESPMEM**
 Defines the copybooks for request and response. In this example they are both set to DFH0XCP4.
- LOGFILE, WSBIND, and WSDL**
 Specifies the fully qualified UNIX file names of the files to be generated.
- URI** This is the URI at which you want the resultant Web service to be available. In this example a relative URI has been specified, but it is advisable to use a full URI if you have CICS TS V3.2 or above (as this will avoid having to change the generated WSDL later).

MAPPING-LEVEL

Specifies the level of mapping that DFHLS2WS uses when generating the Web service binding file and Web service description. For CICS TS v3.1 you are recommended to use 1.2. For CICS TS v3.2 you are recommended to use 2.2. For CICS TS v4.1 you are recommended to use 3.0.

For the detailed difference of each mapping level. Refer to Chapter 2, "CICS implementation of Web services" on page 31

Submit the job. DFHLS2WS creates the WSDL and the WSBinding file. A log file is also produced, but you will not need to use this unless you have to contact IBM support.

Deploy the generated WSBinding file to your CICS region by copying it into the WSDIR directory of your provider mode PIPELINE and issuing a SCAN command against that PIPELINE.

Enhancing the generated service (Meet-in-the-middle)

The bottom-up approach is adequate for most purposes. However, the service you expose to the outside world using the bottom-up approach will appear machine-generated to your client-side developers.

You might consider modifying the generated WSDL for several reasons, such as:

- ▶ You want to minimize your network traffic. Look at the WSDL you just generated. The Web Services Assistant maps the whole copybook, which is reflected in the WSDL, so for every request a client sends conforming to this WSDL the complete data structure has to be provided. This includes elements that might not be necessary for a request, such as the return code and response message. At some mapping level it might even include filler fields. So you might want an optimized WSDL that specifies requests consisting only of the required elements and responses that contain only relevant information.
- ▶ The Web Services Assistant maps the names from the copybook. For convenience you might want to change them (for example, from `ca_return_code` to `rc`).
- ▶ You might not like some of the mappings used by CICS. You might decide that a particular field would be better expressed in the WSDL with a different set of restrictions.
- ▶ You might want to add version control fields to the data structures to help with future application evolution.
- ▶ You might want to combine multiple generated WSDL documents together as one composite Web service with multiple operations.

In these cases you have to approach the problem from the WSDL side to meet the solution in the middle. This approach is considered to be a best-practice.

Important: If you change the WSDL file, you must also regenerate the WSBind file and the language structures. This, in turn, will require application changes.

The following steps are an example of changing the WSDL that has just been generated using DFHLS2WS.

1. Modify the generated WSDL file. A simplified WSDL file for the CICS catalog manager application example is available here:

```
/cicsts41/samples/webservices/wsd1/inquireSingleWrapper.wsdl
```

Look at the simplified WSDL file. See Example 6-2 on page 145 for details. Note that only the 'itemRequiredReference' is required for a request.

Example 6-2 Excerpt from inquireSingleWrapperwsdl

```
<xsd:element name="inquireSingleRequest" nillable="false">
  <xsd:complexType mixed="false">
    <xsd:sequence>
      <xsd:element name="itemRequiredReference" nillable="false">
        <xsd:simpleType>
          <xsd:restriction base="xsd:unsignedShort">
            <xsd:maxInclusive value="9999" />
            <xsd:minInclusive value="0" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

2. This time, use the DFHWS2LS batch job from the Web Services Assistant to create language structures and a new WSBIND file. Some example JCL for doing this with CICS TS V4.1 follois shown in Example 6-3. It takes as input your modified WSDL.

Example 6-3 WS2LSISW.jcl - sample jobcard to execute DFHWS2LS

```
//WS2LS1 JOB (999,P0K),'CICS LS2WS TOOL',MSGCLASS=T,
//          CLASS=A,NOTIFY=&SYSUID,TIME=1440,REGION=0M
//*
//JOBPROC JCLLIB ORDER=CICSTS41.CICS.SDFHINST
//*
//WS2LS EXEC DFHWS2LS,
// JAVADIR='java/J6.0',
// USSDIR='cicsts41',
// PATHPREF=''
//INPUT.SYSUT1 DD *
PDSLIB=//CICRS9.COPYLIB
LANG=COBOL
PGMINT=CHANNEL
CONTID=DFHWS-DATA
REQMEM=ISWCRQ
RESPMEM=ISWCRS
MAPPING-LEVEL=1.0
LOGFILE=/u/cicrs9/provider/wsbind/inquireSingleWrapperSelf.log
WSBIND=/u/cicrs9/provider/wsbind/inquireSingleWrapperSelf.wsbind
WSDL=/u/cicrs9/provider/wsd1/inquireSingleWrapperSelf.wsd1
URI=/exampleApp/inquireSingleWrapperSelf
BINDING=exampleAppInquireSingleHTTPSoapBinding
PGMNAME=DFHOXISW
/*
```

Take a closer look at the parameters:

PDSLIB The PDS library where the new language structures will be created.

LANG The programming language to use. COBOL in this example.

PGMINT Specifies how CICS passes data to the target application program. This time we're going to use a CHANNEL.

CONTID Specifies the name of the container to use for the application data.

REQMEM and RESPMEM

The names of the copybooks that will be produced. These are restricted to 6 characters in length so that DFHWS2LS can add a 2 character suffix. For more complicated WSDL documents DFHWS2LS might need to produce multiple output files, typically one per Operation.

LOGFILE and WSBIND

The fully qualified UNIX file names of the generated WSBind file and log file.

WSDL The fully qualified name of the UNIX file containing the modified WSDL.

BINDING A required parameter if you WSDL contains multiple bindings such as inquireSingleWrapper.wsdl. Specify the name of your <binding> element from the WSDL.

PGMNAME The name of a program that will implement the new We3b service. The example specifies CMNISW for catalog manager inquire single wrapper.

MAPPING-LEVEL

Specifies the level of mapping that DFHLS2WS uses when generating the Web service binding file and Web service description. Normally you should use the most recent version that is available to you, however we have used mapping level 1.0 as there is a CICS supplied example PROGRAM that implements the language structures generated at this mapping level.

3. Write a wrapper program with the name you specified in the JCL. It uses the generated copybooks, maps between them, and the original application, and is responsible for calling the original application.

For the CICS catalog manager example application, a suitable program has been provided called DFH0XISW. It implements the new (mapping level 1.0) generated interface and links to the existing DFH0XCMN program.

The previously generated language structures can be found in copybooks DFH0XWC3 (See Example 6-4) and DFH0XWC4. See Example 6-5. See Example 6-6.

Example 6-4 DFH0XWC3 copybook

```
05 inquireSingleRequest.  
  10 itemRequiredReference      PIC 9(4) DISPLAY.
```

Example 6-5 DFH0XWC4 copybook

```
05 inquireSingleResponse.  
  10 returnCode                 PIC 9(2) DISPLAY.  
  10 responseMessage           PIC X(79).  
  10 singleItem.  
    15 itemReferenceNumber      PIC 9(4) DISPLAY.  
    15 itemDescription          PIC X(40).  
    15 department               PIC 9(3) DISPLAY.  
    15 unitCost                 PIC X(6).  
    15 inStock                  PIC 9(4) DISPLAY.  
    15 onOrder                  PIC 9(3) DISPLAY.
```

Example 6-6 DFH0XISW excerpts

```
WORKING-STORAGE SECTION.  
  ...  
  01 REQUEST-CONTAINER-DATA.  
    COPY DFH0XWC3.  
  01 RESPONSE-CONTAINER-DATA.  
    COPY DFH0XWC4.  
  01 CATALOG-COMMAREA.  
    COPY DFH0XCP1.  
PROCEDURE DIVISION.  
  ...  
  EXEC CICS GET CONTAINER('DFHWS-DATA')  
    INTO(inquireSingleRequest)  
    RESP(WS-RESP)  
  END-EXEC  
  
  INITIALIZE CATALOG-COMMAREA  
  MOVE itemRequiredReference TO CA-ITEM-REF-REQ  
  MOVE '01INQS' TO CA-REQUEST-ID  
  
  EXEC CICS LINK PROGRAM(DFH0XCMN)  
    COMMAREA(CATALOG-COMMAREA)  
  END-EXEC  
  ...
```

4. Deploy the generated WSBind file to your CICS region by copying it into the WSDIR directory of your provider mode PIPELINE and issuing a SCAN command against that PIPELINE.

The Top-Down approach with DFHWS2LS

We've just seen an example of using DFHWS2LS top-down as part of the meet-in-the-middle scenario. The top-down scenario isn't considered any further as part of this chapter (though it will be seen again in the following chapter)."

6.3.2 Use Rational Developer for System z

Instead of using the Web Services Assistant, you can also create the Web service enablement components using RDz. A wizard guides you through this process.

P rearrangements

To start, you need a local project in Rational Developer that contains the program you want to expose as a Web service and all copybooks it uses:

1. Select **File** → **New** → **Project**.
2. Expand the Simple folder and select **Project** to create a simple project. Click **Next**.
3. Name your project (for example, LocalSOA) and click **Finish**.
4. Import the Catalog Manager program (DFH0XCMN) and the copybooks it is using (DFH0XCP1 and DFH0XCP2) into this project.

Generate enablement components

To generate enablement components:

1. Right-click your copybook, and select **Enable Enterprise Web service**. A wizard guides you through the generation process. See Figure 6-5 on page 149.
2. On the first page, select 'compiled XML Conversion' at dropdown list Conversion type

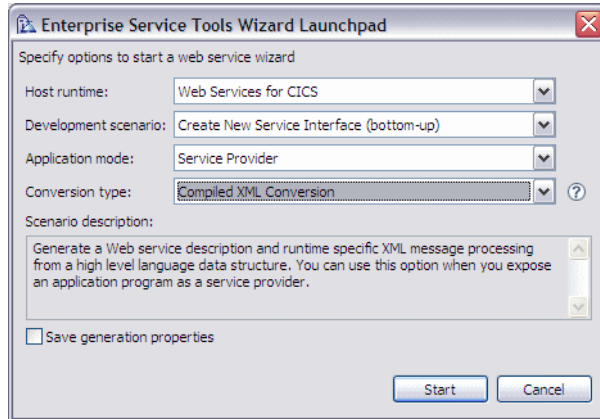


Figure 6-5 Enterprise Services Toolkit (EST) wizard Launchpad

3. Specify your data structures.
4. The first tab (Figure 6-6) asks you to specify your inbound data structure. Expand **DFHCOMMAREA** and select **CA-REQUEST-ID**. Expand **CA-INQUIRE-SINGLE** and select **CA-ITEM-REF-REQ**. These two parameters are required for a single request.

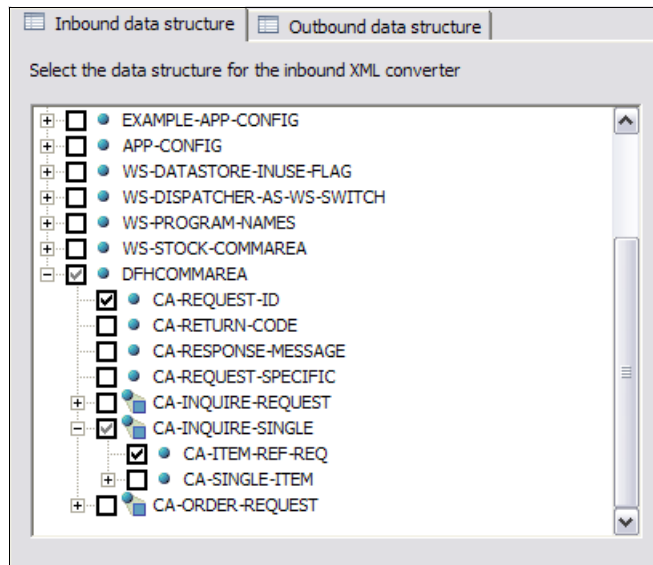


Figure 6-6 Specify inbound data structure

5. Select the other panel to specify your outbound data structure on the second tab (Figure 6-7). This panel offers the possibility to optimize your response to only the elements you really need. Select **CA-RETURN-CODE**, **CA-RESPONSE-MESSAGE**, and the complete **CA-SINGLE-ITEM** element from the CA-INQUIRE-SINGLE element. Click **Next**.

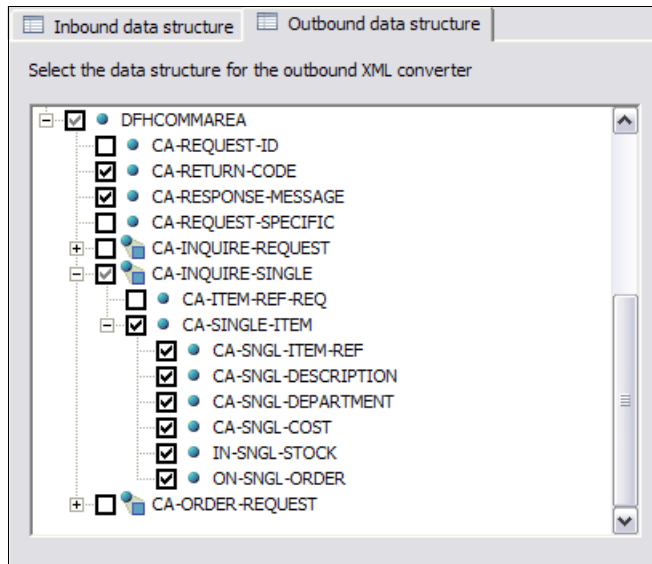


Figure 6-7 Specify outbound data structure

6. The second page (Figure 6-8 on page 151) prompts for properties of the generated artifacts.
 - Select **Web Services for CICS** as converter type.
 - Type CMNIS for Catalog Manager Inquire Single.

Important: The Program name prefix tells the WSBIND file the name of the driver it has to invoke for XML conversion. Be sure this name matches the prefix of the program names on the XML converters panel (Figure 6-11 on page 154). The generated WSBIND file will expect a driver program called CMNISD.

- For the Business program name, specify the name of the CICS catalog manager example application: DFH0XCMN. This is the program to be exposed as a Web service.
- Make sure that all code page entries are set to the code page of your host system.

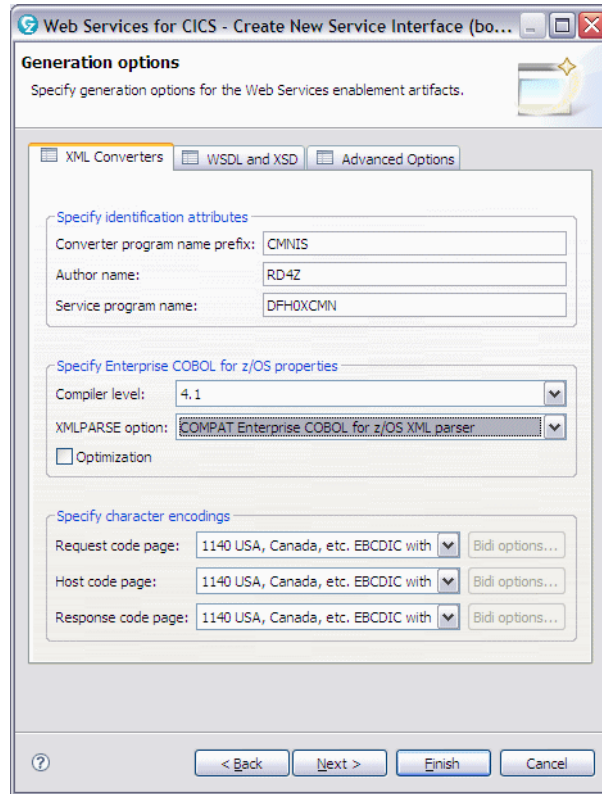


Figure 6-8 XML Converter Options tab

7. On the next panel, specify WSDL and XSD options (see Figure 6-9).
 - Insert the address of your Web service provider here in the following format:
`http://<hostname>:<soap_port>/<web_service_uri>`
 You can change this parameter later in your WSDL file. The local part of your URI (excluding server and port) will be taken as default for the local URI on the Advanced WSBIND Properties panel (
 - If you want to customise the namespaces for the XML schemas in the generated WSDL then specify Inbound and Outbound namespace here. In this example leave the defaults here.

Click **Next**.

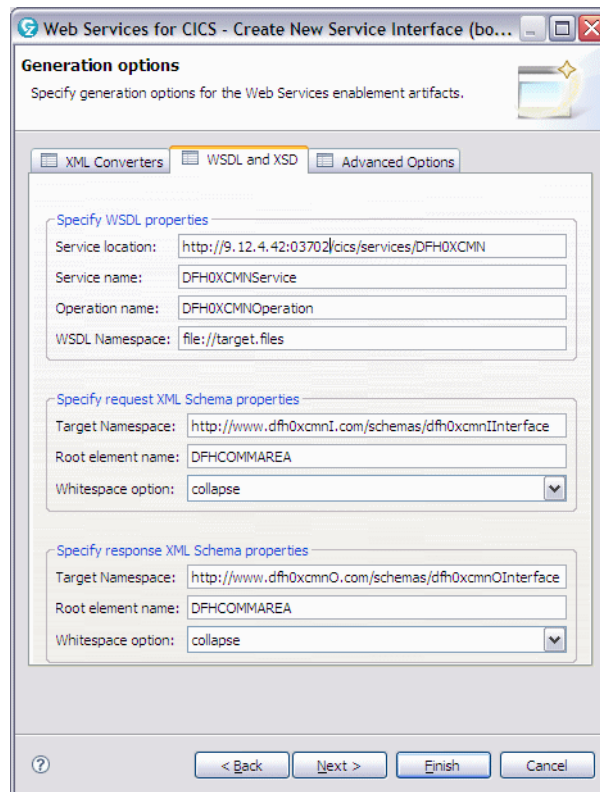


Figure 6-9 WSDL and XSD options

In the next two panels, you will set the WSBIND properties.

8. Specify general options on the WSBind Properties panel (Figure 6-10).

- The WSBind file folder is the local project where your WSBIND file will be created. Leave the default **/LocalSOA**.
- Enter the WSBind file name: `inquireSingleDriverSelf`.
- Select the program interface of your CICS program, which is **COMMAREA**.

Click **Next**.

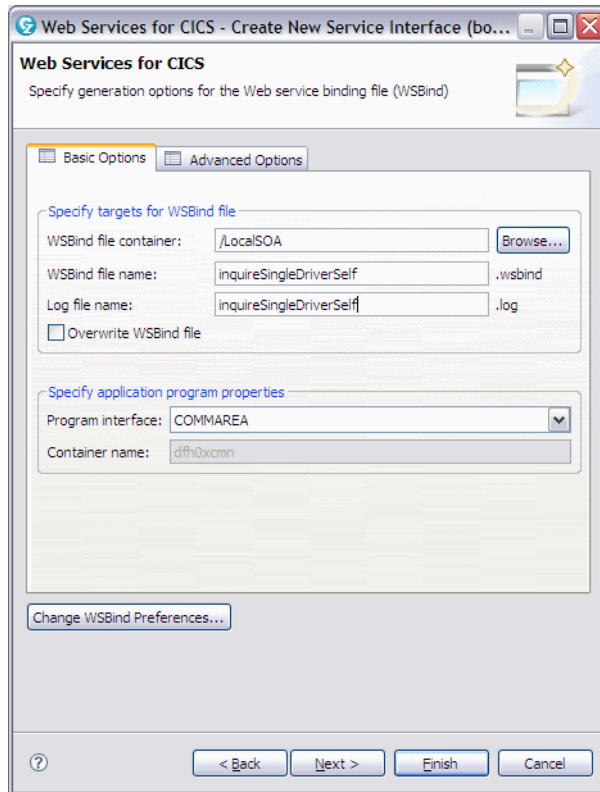


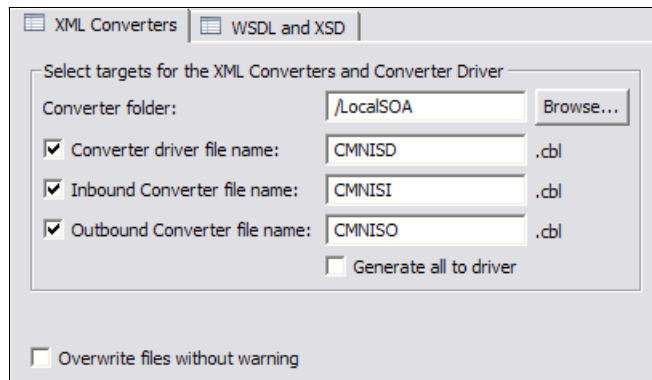
Figure 6-10 WSBind properties

9. Enter the name of your converter files on the following panel (Figure 6-11).

Converter driver CMNISD
Inbound Converter CMNISI
Outbound Converter CMNISO

Important: Check that the names are equal to the name you specified for the XML Converter Options panel (Figure 6-8) except the last letter, which should be D for driver, I for inbound and O for outbound converter.

Click **Next**.



The screenshot shows the 'XML Converters' dialog box with the 'WSDL and XSD' tab selected. The dialog is titled 'Select targets for the XML Converters and Converter Driver'. It contains the following fields and options:

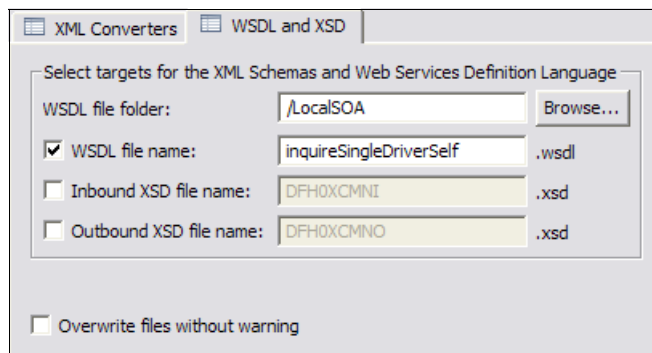
- Converter folder: /LocalSOA (with a 'Browse...' button)
- Converter driver file name: CMNISD .cbl
- Inbound Converter file name: CMNISI .cbl
- Outbound Converter file name: CMNISO .cbl
- Generate all to driver
- Overwrite files without warning

Figure 6-11 XML Converters

10. On the last panel (Figure 6-12), specify the WSDL file name:

inquireSingleDriverSelf

Click **Finish**.



The screenshot shows the 'WSDL and XSD' dialog box with the 'WSDL and XSD' tab selected. The dialog is titled 'Select targets for the XML Schemas and Web Services Definition Language'. It contains the following fields and options:

- WSDL file folder: /LocalSOA (with a 'Browse...' button)
- WSDL file name: inquireSingleDriverSelf .wsdl
- Inbound XSD file name: DFH0XCMNI .xsd
- Outbound XSD file name: DFH0XCMNO .xsd
- Overwrite files without warning

Figure 6-12 WSDL and XSD

11. Copy your driver and converter programs to the host where they must be compiled and statically linked with the converter driver program as the main entry point.
12. Example 6-7 shows a sample JCL. Your z/OS system requires a version of Enterprise COBOL that supports XML parsing (Version 3.1 or later). The target PDSE should be in the DFHRPL concatenation of the target CICS region so that CICS can find the load module. Submit the JCL.

Example 6-7 SVLCOB.jcl, compile and link driver and converter

```
//CICRS9B JOB (999,POK),NOTIFY=&SYSUID,
//      CLASS=A,MSGCLASS=T,MSGLEVEL=(1,1),TIME=1440
//*****
//* COMPILE INBOUND CONVERTER
//*****
//INBOUND EXEC IGYWC,PARM.COBOL='LIB'
//COBOL.SYSIN DD DSN=CICRS9.COBOL(CMNISI),DISP=SHR
//COBOL.SYSLIB DD DSN=CICSTS41.CICS.SDFHSAMP,DISP=SHR
//      DD DISP=SHR,DSN=CICSTS41.CICS.SDFHCOB
//COBOL.SYSLIN DD DSN=CICRS9.COBOL.OBJ(CMNISI),DISP=SHR
//*****
//* COMPILE OUTBOUND CONVERTER
//*****
//OUTBOUND EXEC IGYWC,PARM.COBOL='LIB'
//COBOL.SYSIN DD DSN=CICRS9.COBOL(CMNISO),DISP=SHR
//COBOL.SYSLIB DD DSN=CICSTS41.CICS.SDFHSAMP,DISP=SHR
//      DD DISP=SHR,DSN=CICSTS41.CICS.SDFHCOB
//COBOL.SYSLIN DD DSN=CICRS9.COBOL.OBJ(CMNISO),DISP=SHR
//*****
//* COMPILE AND LINK CONVERTERS AND DRIVER STATICALLY
//*****
//IGYWCL EXEC IGYWCL,PARM.LKED='MAP',
//      LIBPRFX='CEE'
//COBOL.SYSIN DD DSN=CICRS9.COBOL(CMNISD),DISP=SHR
//COBOL.SYSLIB DD DSN=CICRS9.COBOL,DISP=SHR
// DD DSN=CEE.SCEESAMP,DISP=SHR
// DD DSN=CICSTS41.CICS.SDFHSAMP,DISP=SHR
//COBOL.STEPLIB DD
// DD DSN=CICSTS41.CICS.SDFHLOAD,DISP=SHR
//LKED.OBJECT DD DSN=CICRS9.COBOL.OBJ,DISP=SHR
//LKED.SYSLIB DD
// DD DSN=CICSTS41.CICS.SDFHLOAD,DISP=SHR
//LKED.SYSLMOD DD DSN=CICSSYSF.ERWW.LOADLIB(CMNISD),DISP=SHR
//LKED.STUFF DD DSN=CICSSYSF.ERWW.LOADLIB,DISP=SHR
//LKED.SYSIN DD *
//      INCLUDE OBJECT(CMNISI)
//      INCLUDE OBJECT(CMNISO)
//*
```

13. Load your program into CICS.

Deploy the generated WSBind file to your CICS region by copying it into the WSDIR directory of your provider mode PIPELINE and issuing a SCAN command against that PIPELINE.

In this example we have used the Compiled technology in RDz. Another option is to use the Interpreted approach. If the Interpreted approach is used, we can go on to use the Application Deployment Manager to install all of the necessary resources into a Test CICS region from within the RDz tool.

Important: Be aware that in this case CICS is not parsing the request directly. The parsing is delegated to the driver and the converter programs using the information specified in the WSBind file.

RDz also has wizards to assist with meet-in-the-middle scenarios.

6.4 Testing the Web service

In this section we look at how you can test the Web service using the Web Services Explorer in RDz. The Web Services Explorer is part of the Eclipse platform on which RDz is built. This means that if you do not have RDz you can still use the free Eclipse tool to test your CICS Web Service using the techniques described below.

6.4.1 The Web Services Explorer

RDz gives you an excellent opportunity to test your CICS Web service. You just need the WSDL file. This chapter shows how to test `inquireSingle.wsdl` and `inquireSingleWrapper.wsdl`. However, no matter which file you want to test you will have to import it to your workspace if it was not generated in RDz.

To define which Web service you want to invoke, specify the Web services end point in the following format:

```
http://<host_address>:<soap_port>/<web_service_uri>
```

You can edit the endpoint in several ways:

- ▶ Change the `<soap:address>` element in your WSDL source code by pointing its location attribute to the required end point; for example, for `inquireSingle`:

```
<soap:address  
  location="http://9.12.4.42:03702/exampleApp/inquireSingle" />
```

- ▶ Or if you use the inquireSingleWrapper:


```
<soap:address
  location="http://9.12.4.42:03702/exampleApp/inquireSingleWrapper" />
```
- ▶ Use the graphical WSDL editor in Rational Developer. Right-click your WSDL file and select **Open With** → **WSDL Editor**. Click **port'DFH0XCMNPort'**. Then click the Properties tab to specify the Address. See Figure 6-13.

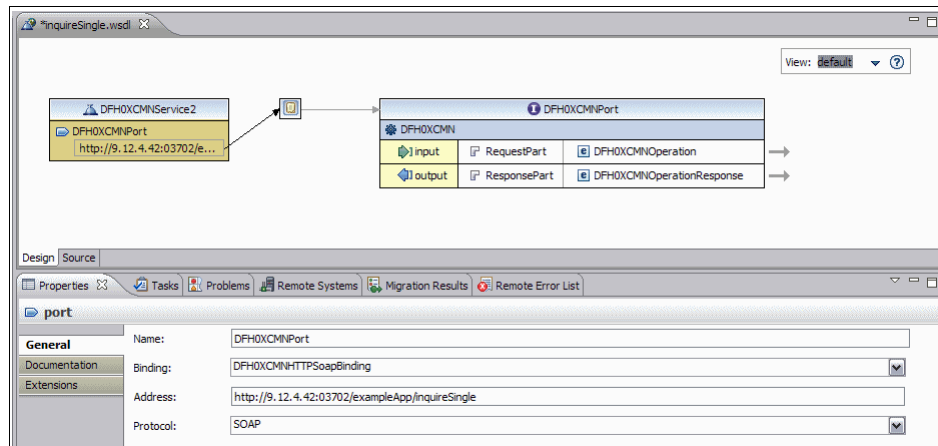


Figure 6-13 Using the WSDL Editor

You might also set the endpoint directly in the Web Services Explorer in the Actions window:

1. To start the Web Services Explorer, right-click your WSDL file and select **Web Services** → **Test with Web Services Explorer**. This opens a new window (Figure 6-14) that has three panes.

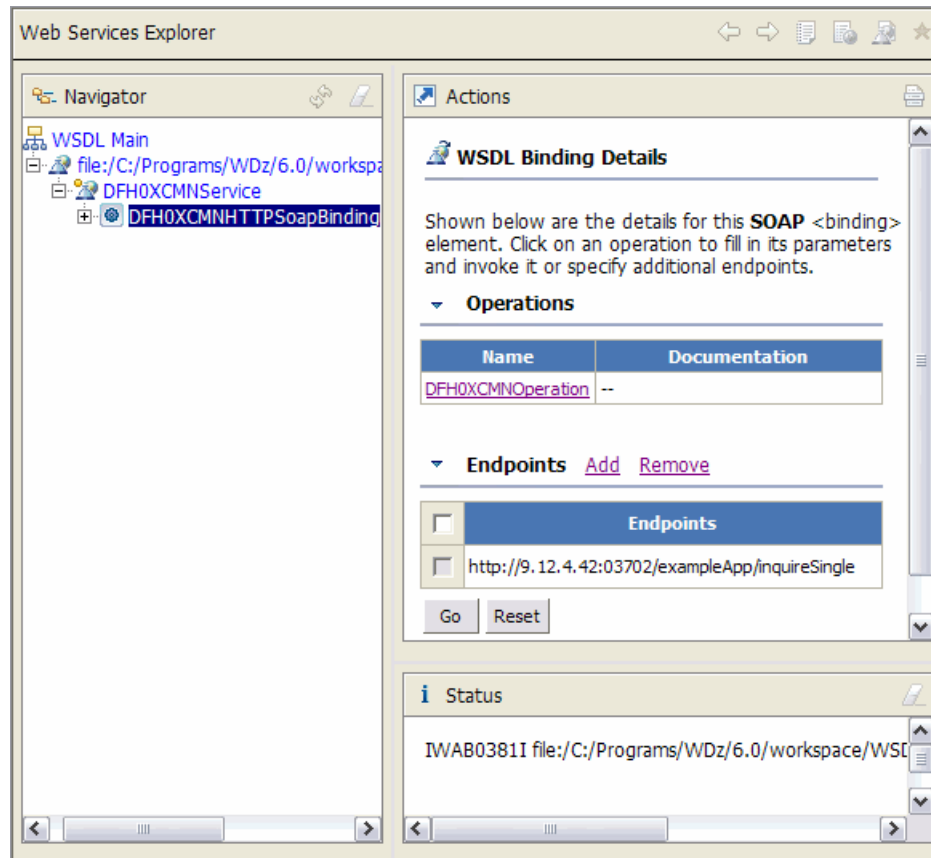


Figure 6-14 The Web Services Explorer after startup

- The Navigator shows all previously tested WSDL files. For each file you can navigate to its services, bindings, and operations.
- The Actions pane is used to change the endpoint at runtime or to execute an operation.
- The Status pane yields any output messages.

- To issue a Web service request, click the **DFH0XCMNOperation** link. The Web Services Explorer provides a form where you enter your request-specific data.

Figure 6-15 shows this form for `inquireSingle` and `inquireSingleWrapper`.

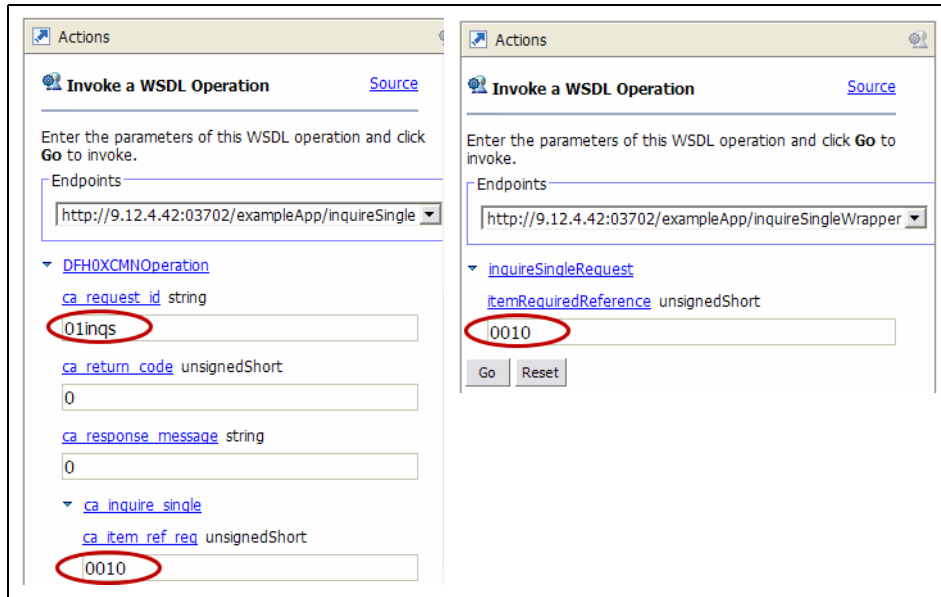


Figure 6-15 Issue a Web service request for `inquireSingle` and `inquireSingleWrapper`

- Fill in the values shown in Table 6-1.

Table 6-1 Sample Web service request values

Value	<code>inquireSingle</code>	<code>inquireSingleWrapper</code>
request id	<code>ca_request_id:01inqs</code>	n/a
item reference	<code>ca_item_ref_req:0010</code>	<code>itemRequiredReference: 0010</code>
any other value	0 ^a	n/a ^b

- Although your CICS program does not use these parameters, you must still provide dummy values for them to conform to your WSDL. You might want to optimize the WSDL file to submit request id and item reference only. Refer to Chapter 4, “CICS catalog manager example application” on page 73 as further changes are required.
- This WSDL is optimized, so you do not have to insert any dummy values.

- Click **Go**.

The responses of both requests look similar (see Figure 6-16), but where `inquireSingle` returns the complete dataset, `inquireSingleWrapper` yields only data you requested because its response was optimized.

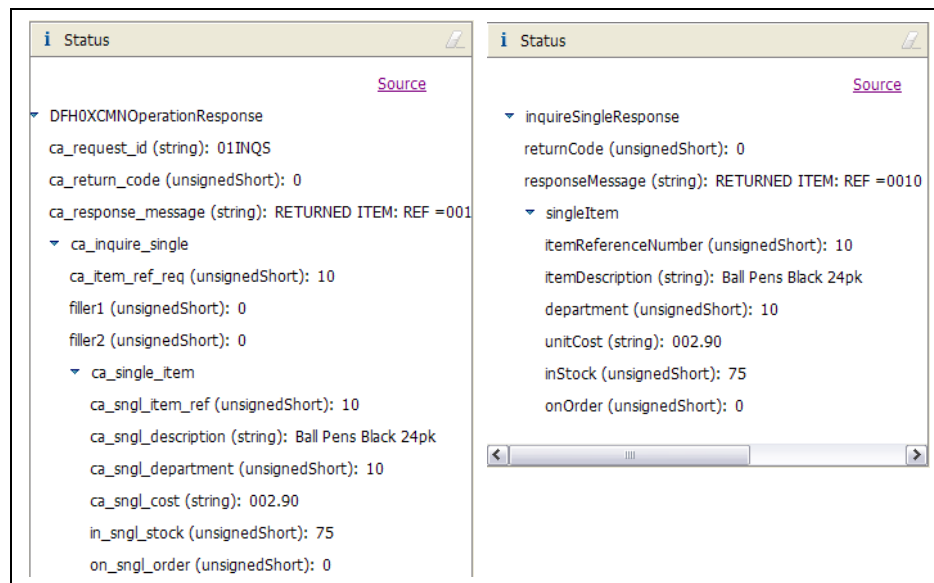


Figure 6-16 Web service responses for `inquireSingle` and `inquireSingleWrapper`

You have now performed a simple test of the Web service in CICS.

6.4.2 Generate a client

To invoke a Web service, you might want to generate a more complete client application that can be hosted in an environment such as Websphere Application Server. An example of such a client is the Example Application Client, which is shipped with the CICS Catalog Manager Example Application and can be found at:

```
/cicsts41/samples/webservices/client/ExampleAppClient.ear
```

This enterprise application provides a client to invoke the three functions of the catalog manager:

- ▶ `inquireSingle`
- ▶ `inquireCatalog`
- ▶ `placeOrder`

For more information about the client and installation guidance, refer to Chapter 3, “Development approaches” on page 61.

If you want to generate your own client, you can use RDz. It generates all the required Java classes to create a Web service request and to receive a Web service response. It can also build a basic graphical user interface to interact with those classes.

For this chapter, you will create a client to invoke the inquireSingle Web service. The corresponding WSDL file is located at:

```
/cicsts41/samples/webservices/wsd1/inquireSingle.wsdl
```

A precondition for the client generation is that the Web service description file (WSDL file) is in your workspace.

1. Right-click your WSDL file and select **Web Services** → **Generate Client**. If the Web services option is not displayed, ensure you enabled the Web service role in your Rational Developer profile.

The first panel (Figure 6-17) of the client generation wizard opens.

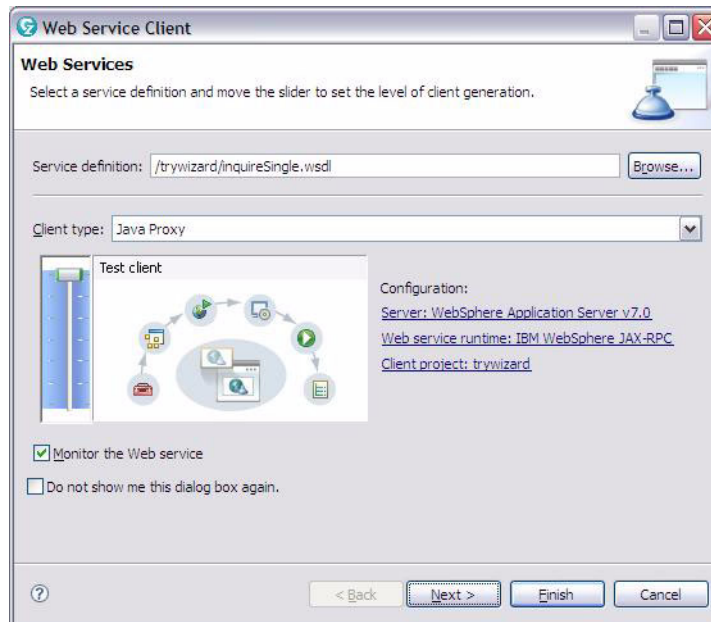


Figure 6-17 Select client type and options

2. Choose to generate a Java Proxy. Select the **Monitor the Web service** check box to enable the TCP/IP Monitor to monitor your traffic. Move the slider to set the level of client generation. In this sample, we move it to top. Click **Next**.

3. On the next panel (Figure 6-18), configure security. For this example, leave the defaults and click **Next**.

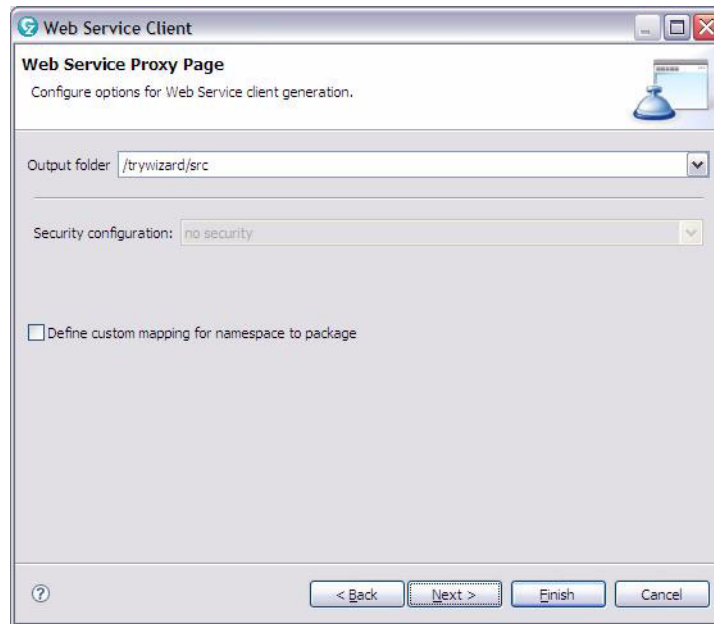


Figure 6-18 Client Security

- On the last panel (Figure 6-19), specify the test options. Choose **Web service sample JSPs** as the test facility, leave the default for the Folder name (sampleDFH0XCMNPortProxy), and select all of the methods. Click **Finish**.

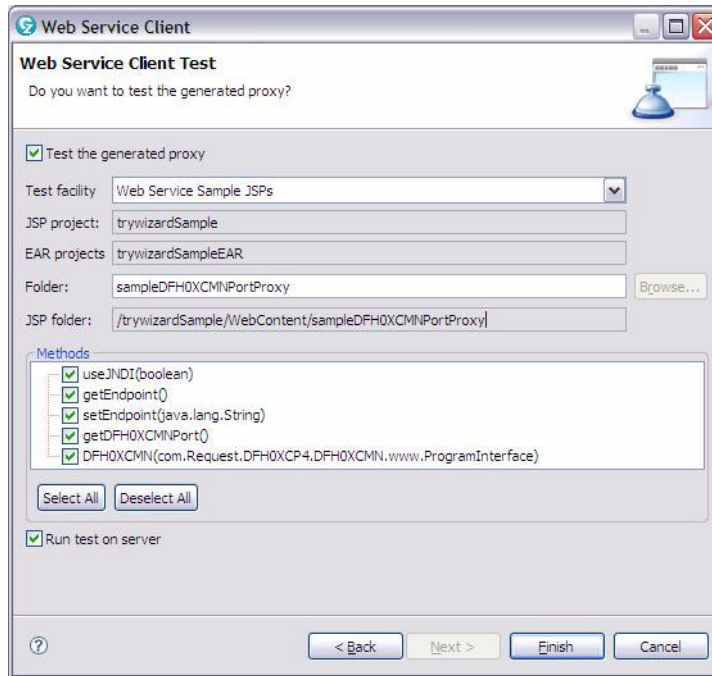


Figure 6-19 Test options

You can now invoke the Web service with the newly generated JSPs as shown in Figure 6-20. But be sure that you installed the Web services runtime environment of the CICS Catalog Manager Example application as described in Chapter 4, “CICS catalog manager example application” on page 73.

The screenshot displays a web application interface with two main sections: "Methods" and "Inputs".

Methods:

- [useJNDI\(boolean\)](#)
- [getEndpoint\(\)](#)
- [setEndpoint\(
\(java.lang.String\)](#)
- [getDFH0XCMNPort\(\)](#)
- [DFH0XCMN
\(java.lang.String,int,java.la](#)

Inputs:

ca_request_id:

ca_return_code:

ca_response_message:

ca_inquire_single:

ca_single_item:

ca_sngl_item_ref:

ca_sngl_cost:

on_sngl_order:

in_sngl_stock:

ca_sngl_department:

ca_sngl_description:

ca_item_ref_req:

Result

Figure 6-20 Sample JSPs of the generated client

Explore the generated files and classes to adapt the client to your requirements.

6.5 Publishing WSDL to WebSphere Service Registry and Repository

Some organizations like to have a central repository for WSDL-based services. The WebSphere Service Registry and Repository exists to help satisfy this requirement. WebSphere Service Registry and Repository helps you to manage and govern services and processes. A central repository can help you to find

Web services quickly and it can also help to enforce version control for your Web services.

If you use RDz to generate CICS Web Services, then you can publish WSDL to WebSphere Service Registry and Repository by simply clicking on the WSDL and following the instructions in the associated wizard.

In CICS TS V4.1 the Web services assistant also includes the ability to interact with WSRR. Both DFHLS2WS and DFHWS2LS have been extended to include parameters to interoperate with WebSphere Service Registry and Repository, optionally using SSL encryption. DFHLS2WS also includes an optional parameter so that you can add your own customized metadata to the WSDL document in WebSphere Service Registry and Repository.

Publishing and retrieving WSDL to and from WebSphere Service Registry and Repository is optional. The following information is provided to assist you to use WebSphere Service Registry and Repository with the Web Services Assistant in CICS TS V4.1, if you want to do so.

6.5.1 Changes to DFHLS2WS for WebSphere Service Registry and Repository in CICS TS V4.1.

When you create a new Web service from a language structure, you can now decide whether you want to publish it on a WebSphere Service Registry and Repository server. Example 6-8 is an example of how to use DFHLS2WS to publish generated WSDL to WebSphere Service Registry and Repository.

Example 6-8 sample jobcard to execute DFHLS2WS with WSRR

```
//LS2WSIS JOB (999,P0K),'CICS LS2WS TOOL',MSGCLASS=T,
//          CLASS=A,NOTIFY=&SYSUID,TIME=1440,REGION=0M
//*
// JCLLIB ORDER=CICSTS41.CICS.SDFHINST
//*
// SET QT=' '
//LS2WS EXEC DFHLS2WS,
//  JAVADIR='java/J6.0',
//  USSDIR='cicsts41',
//  PATHPREF=' '
//INPUT.SYSUT1 DD *
  PDSLIB=//CICSTS41.CICS.SDFHSAMP
  PGMNAME=DFHOXCMN
  LANG=COBOL
  PGMINT=COMMAREA
  REQMEM=DFHOXCP4
```

```
RESPMEM=DFH0XCP4
MAPPING-LEVEL=3.0
LOGFILE=/u/cicsrs9/provider/wsbind/inquireSingleSelf.log
WSBIND=/u/cicsrs9/provider/wsbind/inquireSingleSelf.wsbind
WSDL=/u/cicsrs9/provider/wsd1/inquireSingleSelf.wsd1
URI=exampleApp/inquireSingleSelf
WSRR-SERVER=9.12.4.45:3001
WSRR-NAME=inquireSingleSelf.wsd1
WSRR-USERNAME=cicsrs9
WSRR-PASSWORD=cicsrs9
WSRR-VERSION=1
WSRR-ENCODING=UTF-8
*/
```

The following new parameters are added to DFHLS2WS:

- ▶ **WSRR-ENCODING=value**

Use this optional parameter to specify the character set encoding of the WSDL document. If the WSRR-ENCODING parameter is not specified, WSRR uses the value specified in the WSDL document.

Use this parameter only when the WSRR-SERVER parameter is specified.
- ▶ **WSRR-PASSWORD=value**

Use this optional parameter if you must enter a password to access WSRR.

If the WSRR-USERNAME parameter is specified, you must also specify this parameter.

Use this parameter only when the WSRR-SERVER parameter is specified.
- ▶ **WSRR-NAME=value**

Specifies the name of the WSDL document to retrieve from WSRR. Use this parameter only when the WSRR-SERVER parameter is specified
- ▶ **WSRR-SERVER={domain name:port number}|{IP address:port number}**

Use this parameter to specify the location of the IBM® WebSphere® Service Registry and Repository (WSRR) server. If this parameter is specified, WSRR parameter validation is used.
- ▶ **WSRR-USERNAME=value**

Use this optional parameter if you are required to specify a user name to access WSRR. This user name is used by WSRR to set the owner property.

Use this parameter only when the WSRR-SERVER parameter is specified.

- ▶ WSRR-VERSION=1|value

Use this parameter to set the version property of the WSDL document in WSRR.

Use this parameter only when the WSRR-SERVER parameter is specified.

6.5.2 Changes to DFHWS2LS for WSRR in CICS TS V4.1

When you create a language structure from a WSDL document, you can now decide whether you want to use a WSDL document that is published on a WebSphere Service Registry and Repository server. Example 6-9 shows how to get a WSDL from WebSphere Service Registry and Repository and generate language structure base on this WSDL.

Example 6-9 Sample jobcard to execute DFHWS2LS with WSRR

```
//WS2LS1 JOB (999,P0K),'CICS LS2WS TOOL',MSGCLASS=T,
//          CLASS=A,NOTIFY=&SYSUID,TIME=1440,REGION=OM
//*
//JOBPROC JCLLIB ORDER=CICSTS41.CICS.SDFHINST
//*
//WS2LS EXEC DFHWS2LS,
//  JAVADIR='java/J6.0',
//  USSDIR='cicsts41',
//  PATHPREF=''
//INPUT.SYSUT1 DD *
  PDSLIB=//CICRS9.COPYLIB
  LANG=COBOL
  PGMINT=CHANNEL
  CONTID=DFHWS-DATA
  REQMEM=ISWCRQ
  RESPMEM=ISWCRS
  MAPPING-LEVEL=3.0
  LOGFILE=/u/cicrs9/provider/wsbind/inquireSingleWrapperSelf.log
  WSBIND=/u/cicrs9/provider/wsbind/inquireSingleWrapperSelf.wsbind
  WSDL=/u/cicrs9/provider/wsd1/inquireSingleWrapperSelf.wsd1
  URI=/exampleApp/inquireSingleWrapperSelf
  BINDING=exampleAppInquireSingleHTTPSoapBinding
  PGMNAME=CMNISW
  WSRR-SERVER=9.12.4.45:3001
  WSRR-NAME=inquireSingleSelf.wsd1
  WSRR-USERNAME=cicrs9
  WSRR-PASSWORD=cicrs9
  WSRR-VERSION=1
/*
```

The following new parameters are added to DFHWS2LS:

- ▶ **WSRR-NAME=value**
Specifies the name of the WSDL document to retrieve from WSRR. Use this parameter only when the WSRR-SERVER parameter is specified.
- ▶ **WSRR-PASSWORD=value**
Use this optional parameter if you must enter a password to access WSRR.
If the WSRR-USERNAME parameter is specified, you must also specify this parameter.
Use this parameter only when the WSRR-SERVER parameter is specified.
- ▶ **WSRR-SERVER={domain name:port number}|{IP address:port number}**
Use this parameter to specify the location of the IBM WebSphere Service Registry and Repository (WSRR) server. If this parameter is specified, WSRR parameter validation is used.
- ▶ **WSRR-USERNAME=value**
Use this optional parameter if you are required to specify a user name to access WSRR. This user name is used by WSRR to set the owner property.
Use this parameter only when the WSRR-SERVER parameter is specified.
- ▶ **WSRR-VERSION=value**
Specifies the version of the WSDL document to retrieve from WSRR. You can optionally use this parameter when the WSRR-SERVER parameter is specified.

6.5.3 New parameters to support SSL encryption in CICS TS V4.1

The following new parameters are added to DFHWS2LS and DFHLS2WS to support SSL:

- ▶ **SSL-KEYSTORE=value**
This optional parameter specifies the fully qualified location of the key store file.
Use this parameter if you want the Web services assistant to use secure sockets layer (SSL) encryption to communicate across a network to an IBM WebSphere Service Registry and Repository (WSRR).
- ▶ **SSL-KEYPWD=value**
This optional parameter specifies the password for the key store.

Use this parameter if you want the Web services assistant to use secure sockets layer (SSL) encryption to communicate across a network to an IBM WebSphere Service Registry and Repository (WSRR).

► **SSL-TRUSTSTORE=value**

This optional parameter specifies the fully qualified location of the trust store file.

Use this parameter if you want the Web services assistant to use secure sockets layer (SSL) encryption to communicate across a network to an IBM WebSphere Service Registry and Repository (WSRR).

► **SSL-TRUSTPWD=value**

This optional parameter specifies the password for the trust store.

Use this parameter if you want the Web services assistant to use secure sockets layer (SSL) encryption to communicate across a network to an IBM WebSphere Service Registry and Repository (WSRR).

For the full list of all the parameters to support WSRR, refer to DFHWS2LS and DFHLS2WS in the CICS Information Center.

6.6 Writing applications that process the XML

You might want to write application programs that work directly with the XML rather than use the transformation capabilities of CICS or RDz. There are several ways you can do this.

6.6.1 Creating a custom application handler

One option is to configure the PIPELINE resource to invoke a user-supplied program to perform the message transformations, and the most straightforward is to specify your program within the <apphandler> element in the pipeline configuration file (Example 6-10).

Example 6-10 Sample pipeline configuration file

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<provider_pipeline
xmlns="http://www.ibm.com/software/htp/cics/pipeline"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
provider.xsd ">
  <service>
    <terminal_handler>
```

```
        <cics_soap_1.1_handler/>
    </terminal_handler>
</service>
    <apphandler>LYC1PROG</apphandler>
</provider_pipeline>
```

In this example <apphandler> has been set to LYC1POG rather than the more typical DFHPITP application handler. In this configuration CICS will handle the SOAP envelope (and any SOAP headers included in the envelope), but LYC1PROG will be driven as the application handler.

LYC1PROG will be attached with a channel, which will include all of the control containers from the PIPELINE, including both (DFHWS-XMLNS and DFHWS-BODY).

This particular program is a simple example that gathers up some information about the system in which it is running and returns this in the SOAP body to the caller. It also records environmental information to Transient Data. Source programs and WSDL file are included in the supplemental materials.

6.6.2 Creating an XML-ONLY WEBSERVICE

A second mechanism for working directly with the XML exists for CICS TS V3.2 and above. This is to create an XML-ONLY WEBSERVICE resource. This is a special type of WEBSERVICE for which CICS knows not to attempt any data transformations, but which otherwise acts like a normal WEBSERVICE.

The advantage of using an XML-ONLY WEBSERVICE is that you have a single deployment model for all of your Web services, and you have access to all of the normal processing for a WEBSERVICE, including the PIPELINE SCAN, the INVOKE command, normal CICS statistics and monitoring and diagnostics, the resolution of the Operation name, and so forth.

To generate a XML-ONLY WSBInd file you must start from a WSDL description of the service. You process this WSDL using DFHWS2LS (or RDz) and set the the XML-ONLY parameter to TRUE. In this scenario DFHWS2LS will not create any language structures. A WSBInd file will be produced as normal.

An application will have to be created as in the previous example. It can access the XML from the DFHWS-BODY and DFHWS-XMLNS containers, and then do whatever is required with that XML. On output it will have to write XML back into these same containers.



Create a CICS Web service requester application using the catalog sample

This chapter guides you through setting up a CICS Web service requester. It focuses on how to:

- ▶ Create Web service enablement artifacts
- ▶ Set up the CICS runtime environment
- ▶ Test the Web service requester

7.1 Introduction

In the previous chapter you learned how to expose an existing CICS application as a Web service. Another common requirement is to create or extend an application in CICS to invoke a remote Web service.

Figure 7-1 shows a scenario where an existing application is to be extended to invoke a Web service.

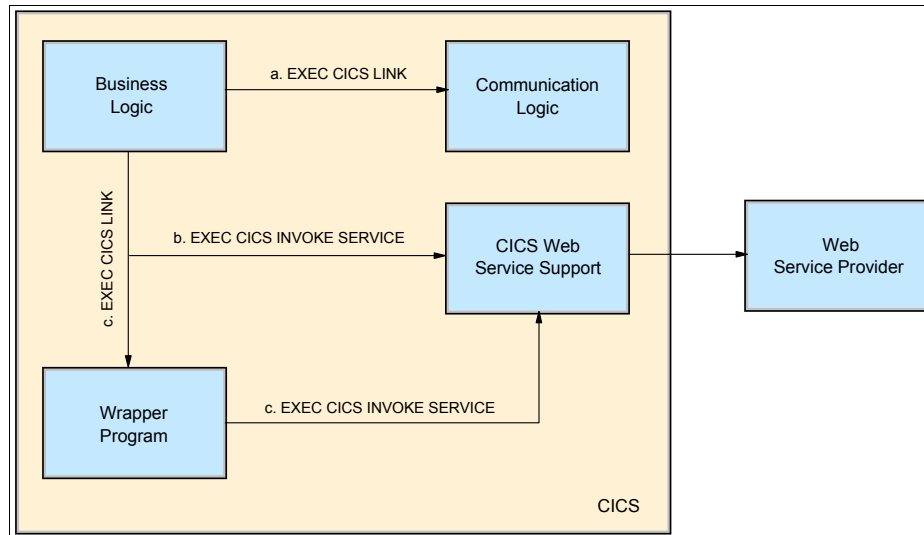


Figure 7-1 CICS as Web service requester: basic scenario

Option A shows a typical application that is partitioned to ensure a separation between communication (or presentation) logic and business logic. The application is ideally structured for code reuse. To invoke a remote Web service you require a copy of the WSDL that describes that service. You use DFHWS2LS (or RDz) to generate application bindings (language structures) and a WSBind file from the WSDL, and deploy the WSBind file as a WEBSERVICE in CICS.

Option B shows an application that uses the EXEC CICS INVOKE (WEB)SERVICE command to interact with the remote Web service. This will typically be a new application that implements this new business requirement, but it could be an existing application that is modified to invoke the Web service.

Option C shows a scenario that requires slightly more work, but is often preferable to option B. In this scenario the application uses EXEC CICS LINK to link to a wrapper program. The wrapper program in turn issues the EXEC CICS INVOKE command.

The wrapper, in this case, has two purposes:

- ▶ It encapsulates the Web service interaction away from the rest of the application.
- ▶ It provides an opportunity to distribute the workload across multiple CICS regions. The business logic can exist in an application owning region (AOR) while the wrapper program exists in another region that specializes in hosting WEBSERVICE resources. This means that you avoid the need for PIPELINE (and similar resources) in your AOR environment.

As an example, we will explore some functionalities of the catalog manager, in this case the dispatch order function that is illustrated in Figure 7-2. In previous examples, the order function of the catalog manager called a simple order dispatcher that returned a basic confirmation message.

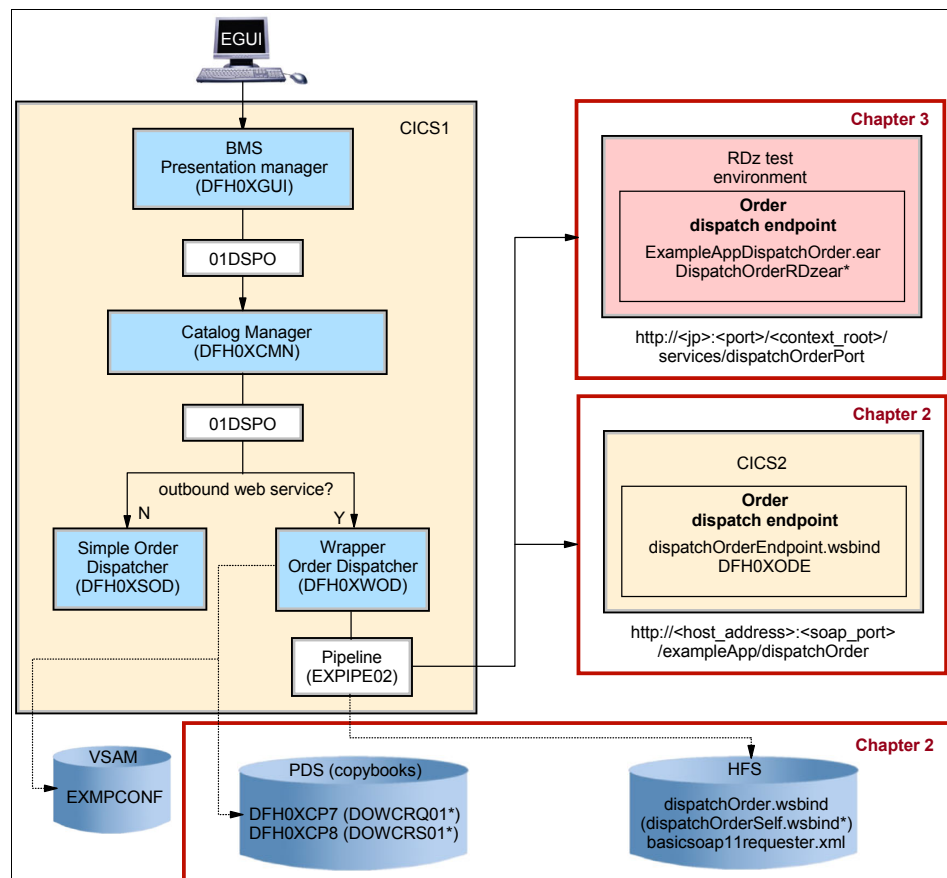


Figure 7-2 Overview over the catalog manager dispatch order function

In 7.2, “Create a Web service requester using the CICS Web Services Assistant” on page 174, we show how to use the Web Services Assistant to generate all required artifacts to request an existing Web service end point in CICS.

In 7.3, “Creating and testing a Web service hosted in RDz” on page 183, we show how you can create a dispatch order end point hosted in RDz that will be invoked by the sample application.

This chapter assumes that you have set up the CICS catalog manager application as described in Chapter 4, “CICS catalog manager example application” on page 73.

7.2 Create a Web service requester using the CICS Web Services Assistant

To generate a Web service requesting application, the following steps are necessary:

1. Create the deployment artifacts:
 - a. Use DFHWS2LS to generate a WSBind file and the associated language structures.
 - b. Write a program to invoke the Web service using the generated language structures.
2. Install the requester mode WEBSERVICE in CICS
3. Test your Web service requesting application using a provider mode Web service hosted in CICS.

7.2.1 Generate the required artifacts

In this step we use DFHWS2LS to process the WSDL using almost the same techniques as we used in the previous chapter.

Important: You are using DFHWS2LS to generate a WSBind file that will be used for a Web service requester. Therefore, the PGMNAME parameter must be omitted, If not, CICS will think the WSBind file is for a provider mode Web service. Furthermore, PGMNINT and URI will not be specified either.

Example 7-1 on page 175 shows a sample job card to generate the artifacts for the dispatch order requester that uses as input:

```
/usr/lpp/cicsts/cicsts41/samples/webservices/wsd1/dispatchOrder.wsd1
```


This file has been renamed to `dispatchOrderSelf.wsdl` and copied to a user UNIX file system directory. The example JCL is suitable for use with CICS TS V4.1.

Example 7-1 WS2LS Sample JCL

```
//WS2LS JOB (MYSYS,AUSER),MSGCLASS=T,
//          CLASS=A,NOTIFY=&SYSUID,REGION=0M
//*
//JOBPROC JCLLIB ORDER=CICSTS41.CICS.SDFHINST
//*
//WS2LS EXEC DFHWS2LS,
// JAVADIR='java/J6.0',
// USSDIR='cicsts41',
// PATHPREF=''
//INPUT.SYSUT1 DD *
PDSLIB=//CICRS6.COPYLIB
LANG=COBOL
REQMEM=DOWCRQ
RESPMEM=DOWCRS
LOGFILE=/u/cicrs6/requester/wsbinding/dispatchOrderSelf.log
WSBIND=/u/cicrs6/requester/wsbinding/dispatchOrderSelf.wsbinding
WSDL=/u/cicrs6/requester/wsd1/dispatchOrderSelf.wsdl
BINDING=dispatchOrderSoapBinding
OPERATIONS=dispatchOrder
MAPPING-LEVEL=1.0
/*
```

- PDSLIB** The PDS library where the language structures (copybooks) will be generated.
- LANG** Specifies the programming language of the language structure to be created.
- REQMEM** and **RESPMEM**
Defines the names of the request and the response language structure respectively. These names are limited to 6 characters so that DFHWS2LS can add a generated suffix.
- WSBIND** and **LOGFILE** parameters
The fully qualified UNIX file names of the WSBinding file and log file to be generated.
- WSDL** Specifies name and location of your input WSDL file.
- BINDING** Must specify if your WSDL contains multiple `<binding>` elements.

OPERATIONS

Specify the WSDL Operations you want to invoke to avoid the generation of unnecessary language structures and meta-data. This parameter is only available in CICS TS V3.2 and later.

MAPPING-LEVEL

Set the mapping level to the most recent version available to you. However, in this example we have used mapping level 1.0 to ensure that the generated language structures will be the same for all versions of CICS.

Submit your job and look at the generated copybooks (Example 7-2 and Example 7-3).

Example 7-2 request copybook:- DOWCRQ01 (DFH0XCP7)

```
05 dispatchOrderRequest.  
  10 itemReferenceNumber          PIC S9(4) DISPLAY.  
  10 quantityRequired             PIC S9(3) DISPLAY.  
  10 customerId                   PIC X(8).  
  10 chargeDepartment             PIC X(8).
```

Example 7-3 response copybook: DOWCRS01 (DFH0XCP8)

```
05 dispatchOrderResponse.  
  10 confirmation                 PIC X(20).
```

Now you would have to write a program that uses those copybooks to invoke a Web service. CICS provides an example of such a program called DFH0XWOD (wrapper order dispatcher). It uses the copybooks DFH0XCP7 and DFH0XCP8, which are equivalent to the copybooks you just created. Example 7-4 shows some excerpts.

Example 7-4 Excerpts from the outbound WebService order dispatcher (DFH0XWOD)

```
WORKING-STORAGE SECTION.  
* WebService Message Structures  
01 WS-DISPATCH-ORDER-MESSAGES.  
  COPY DFH0XCP7.  
  COPY DFH0XCP8.  
  
LINKAGE SECTION.  
01 DFHCOMMAREA.  
  COPY DFH0XCP2.  
  
PROCEDURE DIVISION.
```

```

MOVE 'DFHWS-DATA' TO WS-SERVICE-CONT-NAME
MOVE 'SERVICE-CHANNEL' TO WS-CHANNELNAME
MOVE 'dispatchOrder' TO WS-WEBSERVICE-NAME
MOVE 'dispatchOrder' TO WS-OPERATION

MOVE CA-ORD-ITEM-REF-NUMBER
    TO itemReferenceNumber IN dispatchOrderRequest
MOVE CA-ORD-QUANTITY-REQ
    TO quantityRequired IN dispatchOrderRequest
MOVE CA-ORD-USERID
    TO customerId IN dispatchOrderRequest
MOVE CA-ORD-CHARGE-DEPT
    TO chargeDepartment IN dispatchOrderRequest

EXEC CICS PUT CONTAINER(WS-SERVICE-CONT-NAME)
            CHANNEL(WS-CHANNELNAME)
            FROM(dispatchOrderRequest)

END-EXEC

EXEC CICS INVOKE WEBSERVICE(WS-WEBSERVICE-NAME)
            CHANNEL(WS-CHANNELNAME)
            URI(WS-ENDPOINT-URI)
            OPERATION(WS-OPERATION)
            RESP(RESP) RESP2(RESP2)

END-EXEC.

```

The program receives the data from the 3270 interface in a format according to copybook DFH0XCP2. It extracts all necessary data to build a dispatchOrderRequest according to DFH0XCP7 and puts it into a container which is placed into a channel. The channel is then passed as a parameter on the INVOKE WEBSERVICE command, which has this syntax:

```

>>-INVOKE-WEBSERVICE(name)--CHANNEL(name)----->

>--OPERATION(data-area)--+-----+-----><
                        '-URI(data-area)-'

```

CICS uses the OPERATION parameter as a Web service might have many different operations, each of which has a different programmatic interface. The URI parameter allows the application to override the default endpoint for the Web service request from the WSDL.

7.2.2 Set up the CICS infrastructure

When CICS acts as a Web service requester you will need a WEBSERVICE and a PIPELINE resource. For the dispatch order example, the PIPELINE was supplied by installing the CICS catalog manager application.

The pipeline to use is EXPIPE02. Perform a CEMT INQUIRE on this pipeline to yield a result similar to Figure 7-3.

```
INQUIRE PIPELINE(EXPIPE02)
  RESULT - OVERTYPE TO MODIFY
    Pipeline(EXPIPE02)
    Enablestatus( Enabled )
    Mode(Requester)
    Mtomst(Nomtom)
    Sendmtomst(Nosendmtom)
    Mtomnoxopst(Nomtomnoxop)
    Xopsupportst(Noxopsupport)
    Xopdirectst(Noxopdirect)
    Soaplevel(1.1)
    Respwait( )
    Configfile(/usr/lpp/cicsts/cicsts41/samples/pipelines/basicsoap11request
)
    Configfile(er.xml)
    Shelf(/var/cicsts/)
    Wsdir(/usr/lpp/cicsts/cicsts41/samples/webservices/wsbinding/requester/)
    Ciddomain(cicsts)
    Installtime(08/19/09 04:36:56)
    Installusrid(CICSUSER)
+   Installagent(Csdapi)
    Definesource(CATMGR6)
    Definetime(08/19/09 04:34:32)
    Changetime(08/19/09 04:34:32)
    Changeusrid(CICSUSER)
    Changeagent(Csdapi)
    Changeagrel(0660)
```

Figure 7-3 The requester pipeline

This uses the basic requester pipeline configuration file that is provided with CICS. The WSDIR of this pipeline is the CICS sample directory for requester mode WSBIND files. The provided dispatchorder.wsbinding file should be stored in this directory.

Upon Pipeline installation, the corresponding dispatch order Web service is discovered and installed by CICS. Perform a CEMT INQUIRE on the Web service. This should return a screen similar to Figure 7-4 on page 179.

```

INQUIRE WEBSERVICE(dispatchOrder)
RESULT - OVERTYPE TO MODIFY
  Webservice(dispatchOrder)
  Pipeline(EXPIPE02)
  Validationst( Novalidation )
  State(Inservice)
  Ccsid(00000)
  Urimap()
  Program()
  Pgminterface(Notapplic)
  Xopsupportst(Noxopsupport)
  Xopdirectst(Noxopdirect)
  Mappinglevel(1.0)
  Minrunlevel(1.0)
  Datestamp(20090708)
  Timestamp(22:46:34)
  Container()
  Wsdfile()
  Wsbind(/usr/lpp/cicsts/cicsts41/samples/webservices/wsbind/requester/dis
)
  Wsbind(patchOrder.wsbind)
  Endpoint(http://my-server:9080/exampleApp/dispatchOrder)
  Binding(dispatchOrderSoapBinding)
  Installtime(08/19/09 04:36:56)
  Installusrid(CICRSR1)
  Installagent(Dynamic)
  Definesource(EXPIPE02)
  Definetime(08/19/09 04:36:56)
  Changetime(08/19/09 04:36:56)
  Changeusrid(CICRSR1)
  Changeagent(Dynamic)
  Changeagrel(0660)

```

Figure 7-4 The dispatchOrder Web service

Note that the endpoint specified for this WEBSERVICE is invalid. It will be supplied by the wrapper program programmatically.

Before you start testing your requester, have a look at the Web service you are going to invoke. In this example we have used a CICS Web service called `dispatchOrderEndpoint` that is supplied as part of the catalog sample and installed in provider mode PIPELINE EXPIPE01. To see a result similar to Figure 7-5 on page 180, perform:

```
CEMT INQUIRE WEBSERVICE(dispatchOrderEndpoint)
```

```

INQUIRE WEBSERVICE(dispatchOrderEndpoint)
  RESULT - OVERTYPE TO MODIFY
  Webservice(dispatchOrderEndpoint)
  Pipeline(EXPIPE01)
  Validationst( Novalidation )
  State(Inservice)
  Ccsid(00000)
  Urimap($246340)
  Program(DFH0X0DE)
  Pgminterface(Commarea)
  Xopsupportst(Noxopsupport)
  Xopdirectst(Noxopdirect)
  Mappinglevel(1.0)
  Minrunlevel(1.0)
  Datestamp(20090708)
  Timestamp(22:46:34)
  Container()
  Wsdfile()
  Wsbind(/usr/lpp/cicsts/cicsts41/samples/webservices/wsbind/provider/dispatchOrderEndpoint)
  Wsbind(atchOrderEndpoint.wsbind)
  Endpoint(http://my-server:9080/exampleApp/dispatchOrder)
  Binding(dispatchOrderSoapBinding)
  Installtime(08/19/09 04:36:56)
  Installusrid(CICRSR1)
  Installagent(Dynamic)
  Definesource(EXPIPE01)
  Definetime(08/19/09 04:36:56)
  Changetime(08/19/09 04:36:56)
  Changeusrid(CICRSR1)
  Changeagent(Dynamic)
  Changeagrel(0660)

```

Figure 7-5 The dispatchOrderEndpoint Web service

This Web service provider returns a simple confirmation for a successfully placed order. A more typical example might involve a Web service hosted in WebSphere Application Server, or elsewhere on the network. This example uses a provider mode Web service hosted in CICS to avoid external dependencies within the sample.

7.2.3 Test the requester application

Before you can test the Web service requester, you must change the business logic to invoke the new wrapper program. The CICS catalog manager example application enables you to change the program name to be invoked without

changing the application. You can also specify the network endpoint at which the provider mode target service is available.

1. Type ECFG in CICS to start the configuration program. Change these parameters:

- Set Outbound WebService? to **YES**.

This option forces the catalog manager to use the Order Dispatch WebService (DFH0XWOD) instead of the Order Dispatch Stub (DFH0XSOD).

- Set Outbound WebService URI to the address of the dispatch order endpoint in your CICS region:

`http://<hostname>:<port>/exampleApp/dispatchOrder`

Press Enter to confirm your changes.

```
CONFIGURE CICS EXAMPLE CATALOG APPLICATION

      Datastore Type ==> VSAM           STUB!VSAM
Outbound WebService? ==> YES           YES!NO
      Catalog Manager ==> DFH0XCMN
      Data Store Stub ==> DFH0XSDS
      Data Store VSAM ==> DFH0XVDS
      Order Dispatch Stub ==> DFH0XSOD
Order Dispatch WebService ==> DFH0XWOD
      Stock Manager ==> DFH0XSSM
      VSAM File Name ==> EXMPCAT
Server Address and Port ==> 9.12.4.42:03702
Outbound WebService URI ==> http://9.12.4.42:03702/exampleApp/dispatchOr
                        ==> der
                        ==>
                        ==>
                        ==>
```

Figure 7-6 Adapt the catalog manager configuration

Important: URIs are case sensitive. If the characters you type are transformed to upper case after saving, you should set your terminal to mixed case by typing:

```
CE0T Tra
```

This capitalizes transaction IDs only.

2. Start the catalog manager by typing EGUI. Select option 2 with, for example, element 0010, which returns a panel similar to Figure 7-7. Insert some parameters and press Enter.

```
CICS EXAMPLE CATALOG APPLICATION - Details of your order

Enter order details, then press ENTER

Item      Description                                Cost    Stock    On
Order
-----
0010     Ball Pens Black 24pk                        2.90    0047    000

      Order Quantity: 2
            User Name: cicsuser
            Charge Dept: cicsdptm
```

Figure 7-7 Dispatch an order in CICS

CICS will tell you that your order has been placed successfully, as shown in Figure 7-8.

```
CICS EXAMPLE CATALOG APPLICATION - Main Menu

Select an action, then press ENTER

Action . . . .  1. List Items
                 2. Order Item Number
                 3. Exit

ORDER SUCESSFULLY PLACED
```

Figure 7-8 Successfully dispatched order in CICS

7.3 Creating and testing a Web service hosted in RDz

This chapter will show you how you can create a Web service from a WSDL file and host it (for testing purposes) in RDz. You will create a basic dispatch order Web service from `dispatchOrder.wsdl`, which can be requested by the Web service requester you created in the previous chapter.

If you do not want to create your own Web service you can use the one that is supplied with CICS in the example directory:

```
/usr/lpp/cicsts/cicsts41/samples/webservices/client/ExampleAppDispatchOrder.ear
```

In this case, import this EAR file and proceed to 7.3.2, “Implement the RDz based Web service” on page 187.

7.3.1 Create a Web service skeleton with RDz

In this example we create an RDz-based Web service by generating a JavaBean skeleton. The precondition is that your WSDL file must be either in your workspace or available through remote connection. If this is not already the case, you must transfer this file to your local machine:

```
/usr/lpp/cicsts/cicsts41/samples/webservices/wsd1/dispatchOrder.wsdl
```

Import it into a simple project.

To generate the Web service, select **File** → **New** → **Other**. Expand **Web Services** and select **Web Service**. This starts the Web service wizard.

1. On the first panel (Figure 7-9), select the Web service type (in this example, **Top down Java bean Web Service**). Use the slide bar to display the Test Service option. You also can select the **Monitor the Web service** check box to enable a TCP/IP monitor to listen to the service port. Click **Next**.

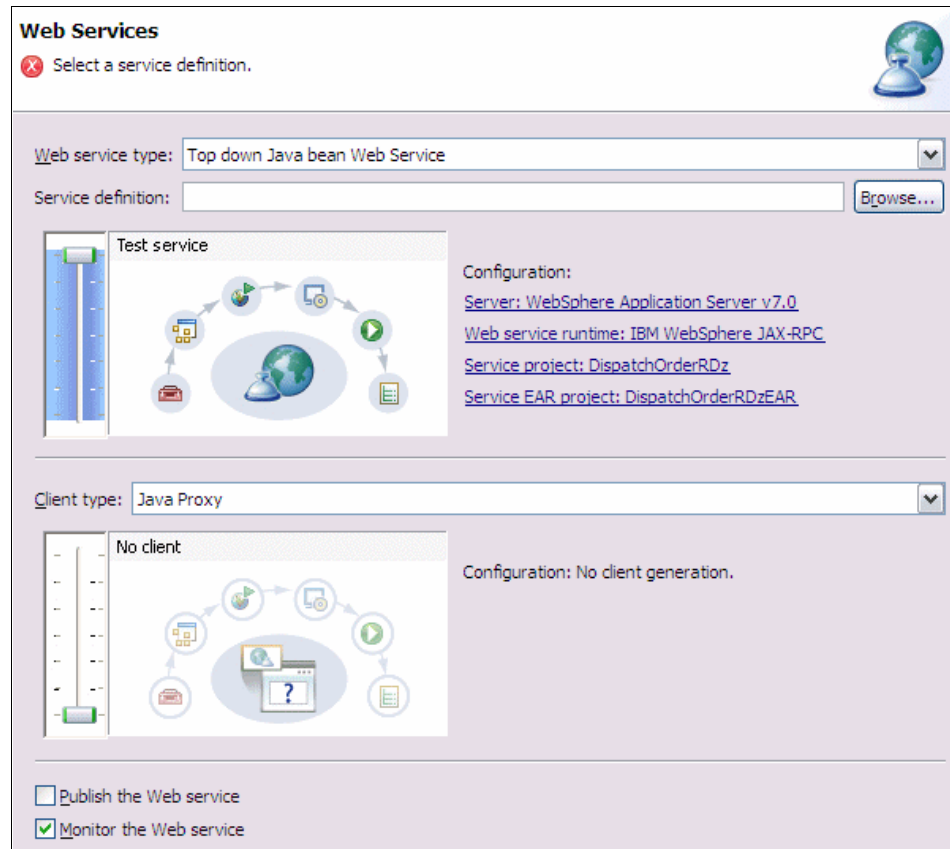


Figure 7-9 RDz Web service wizard: Web service type

2. Browse the service definition to open the second panel (Figure 7-10), locate the WSDL file using **Browse**, and click **OK**.

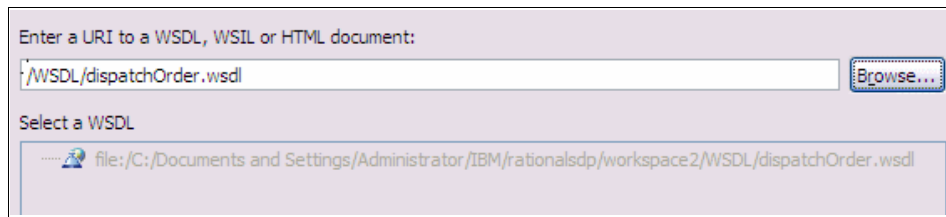


Figure 7-10 RDz Web service wizard: WSDL selection

Note: An alternative method is to right-click `dispatchOrder.wsdl` in the Project Explorer panel and then generate the Web service by selecting **File** → **New** → **Other**. Expand **Web Services** and select **Web Service**. This will preselect the Service Definition

3. From the first panel, select the service project name from the Configuration part to open the third panel. The third panel (Figure 7-11) prompts you for the name of your enterprise application (EAR project: DispatchOrderRDzEAR) and for the name of its contained Web project (Service project: DispatchOrderRDz) that will contain your business logic. Click **Next**.

If the projects do not exist already, they will be created and deployed on the internal test server.

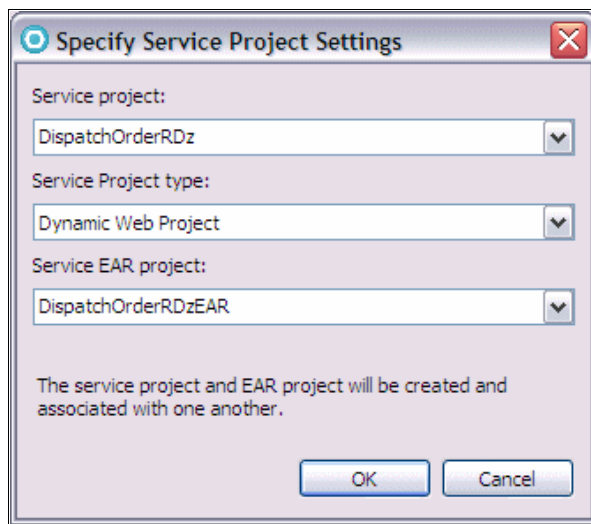
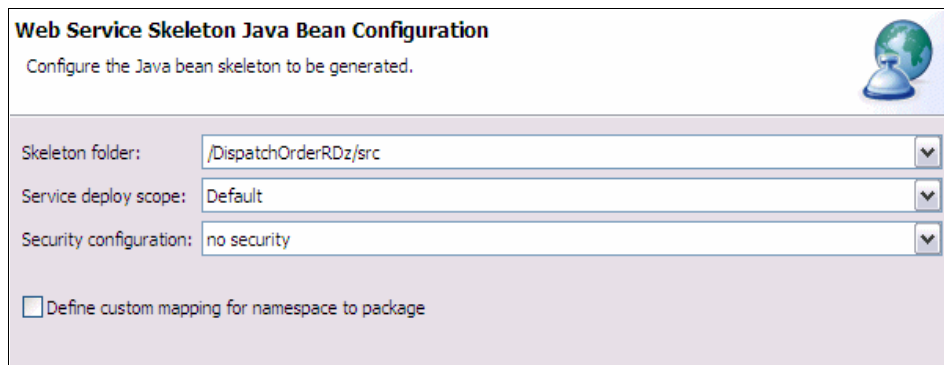


Figure 7-11 RDz Web service wizard: Specify enterprise application

4. The fourth panel (Figure 7-12) asks for the name of the source folder that will contain your Java code. Accept the default, /DispatchOrderRDz/src, and click **Next**.



Web Service Skeleton Java Bean Configuration
Configure the Java bean skeleton to be generated.

Skeleton folder: /DispatchOrderRDz/src

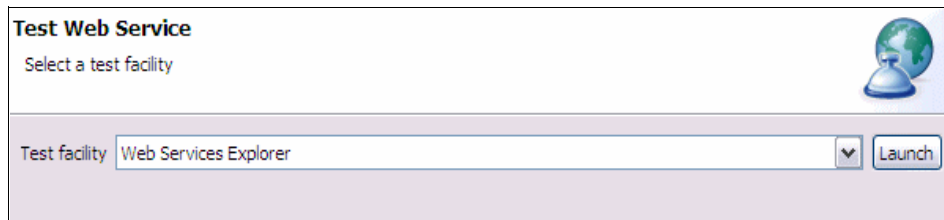
Service deploy scope: Default

Security configuration: no security

Define custom mapping for namespace to package

Figure 7-12 Rational Developer for System z Web service wizard: Skeleton folder

5. You specified on the first panel that you wanted to test your Web service. Now you choose your test facility (Figure 7-13). Choose the **Web Services Explorer**, which will start after the wizard has finished. Click **Next**.



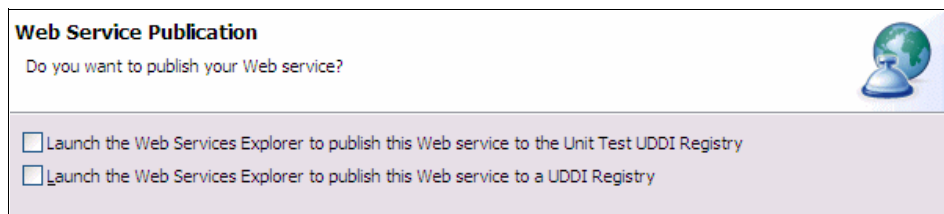
Test Web Service
Select a test facility

Test facility: Web Services Explorer

Launch

Figure 7-13 RDz Web service wizard: Test facility

6. The last panel (Figure 7-14) shows publishing details. Do not select anything for this example. Click **Finish**. You have created a simple Web service.



Web Service Publication
Do you want to publish your Web service?

Launch the Web Services Explorer to publish this Web service to the Unit Test UDDI Registry

Launch the Web Services Explorer to publish this Web service to a UDDI Registry

Figure 7-14 RDz Web service wizard - Publishing

7.3.2 Implement the RDz based Web service

Open the Java EE perspective to look at all generated projects. If you have imported the sample endpoint, it should look similar to Figure 7-15.

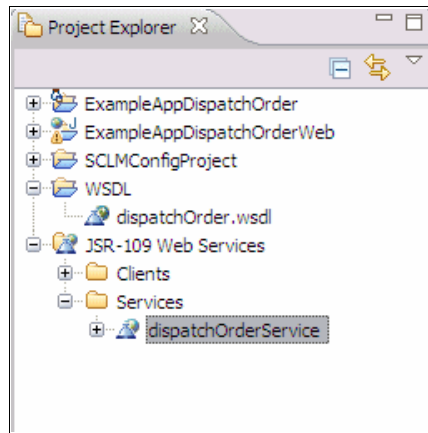


Figure 7-15 Workspace after creating a Web service

Expand the dispatchOrderService, as shown in Figure 7-16.



Figure 7-16 Expand the dispatchOrderService

You see your WSDL file on which the service is based, as well as the actual service implementation in the Service Classes folder. Open the implementation. If you have imported the sample endpoint, it will contain the dispatchOrder public method (Example 7-5 on page 188), which builds an answer object to be returned at the end.

Example 7-5 DispatchOrderSoapBindingImpl.java from sample

```
public DispatchOrderResponse dispatchOrder(DispatchOrderRequest
requestPart)
    throws java.rmi.RemoteException
{
    Confirmation confirmation = new Confirmation();
    confirmation.setValue("Order in Dispatch");
    DispatchOrderResponse response = new DispatchOrderResponse();
    response.setConfirmation(confirmation);
    return response;
}
```

If you have created the service yourself, this implementation will return null. Modify the code as shown in Example 7-6. If the response string you set is longer than 20 characters according to the WSDL, a request from CICS will fail.

Example 7-6 DispatchOrderSoapBindingImpl.java self-generated

```
package com.dispatchOrder.exampleApp.www;
import com.Response.dispatchOrder.exampleApp.www.*;
import com.Request.dispatchOrder.exampleApp.www.*;

public class DispatchOrderSoapBindingImpl implements DispatchOrderPort{
    public DispatchOrderResponse dispatchOrder(DispatchOrderRequest
request)
    throws java.rmi.RemoteException
    {
        DispatchOrderResponse response = new DispatchOrderResponse();
        response.setConfirmation("RDz dispatched order");
        return response;
    }
}
```

7.3.3 Test the Web service using RDz

The Web Services Explorer starts automatically after the Web service wizard finishes. If you closed the window or imported the Web service from the sample directory, start it manually. Right-click the WSDL file in the Web Services folder of the J2EE project explorer. Select **Test with Web Services Explorer**. Select the dispatchOrder operation.

Select the endpoint with port 9081. The complete URI is constructed similar to:
http://localhost:9081/<context_root>/services/<name_of_your_portType>

The name of your portType can be found in the `dispatchOrder.wsdl` file as shown in Example 7-7. This maps to the operation `dispatchOrder` that you will invoke.

Example 7-7 Excerpt from dispatchOrder.wsdl: portType

```
<portType name="dispatchOrderPort">
  <operation name="dispatchOrder">
    <input message="tns:dispatchOrderRequest"
      name="DFHOXODSRequest"/>
    <output message="tns:dispatchOrderResponse"
      name="DFHOXODSResponse"/>
  </operation>
</portType>
```

The context root, which is defined in your EAR file, describes which application on the server (EAR file) will be addressed with the request. In the J2EE perspective, expand your EAR file and the META-INF folder. Open `application.xml` (Example 7-8) to find your context root.

Example 7-8 Excerpts from application.xml of the self-generated endpoint

```
<application version="5"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/application_5.xsd" >
  <display-name> DispatchOrderRDzEAR</display-name>
  <module>
  <web>
  <web-uri> DispatchOrderRDz.war</web-uri>
  <context-root> /DispatchOrderRDz</context-root>
  </web>
  </module>
</application>
```

Go back to the Web Services Explorer to test your Web service. Select the **dispatchOrder** operation. You should see a window similar to Figure 7-17 on page 190 (if you use your self-created endpoint) or Figure 7-18 on page 191 (if you use the sample endpoint). Enter parameters, being sure to conform to your WSDL. Your strings in this case must be exactly eight characters long. Click **Go**. You should get the answer string you just specified in your Java program (RDz dispatched order) or, if you are using the provided sample, the answer is Order in Dispatch.

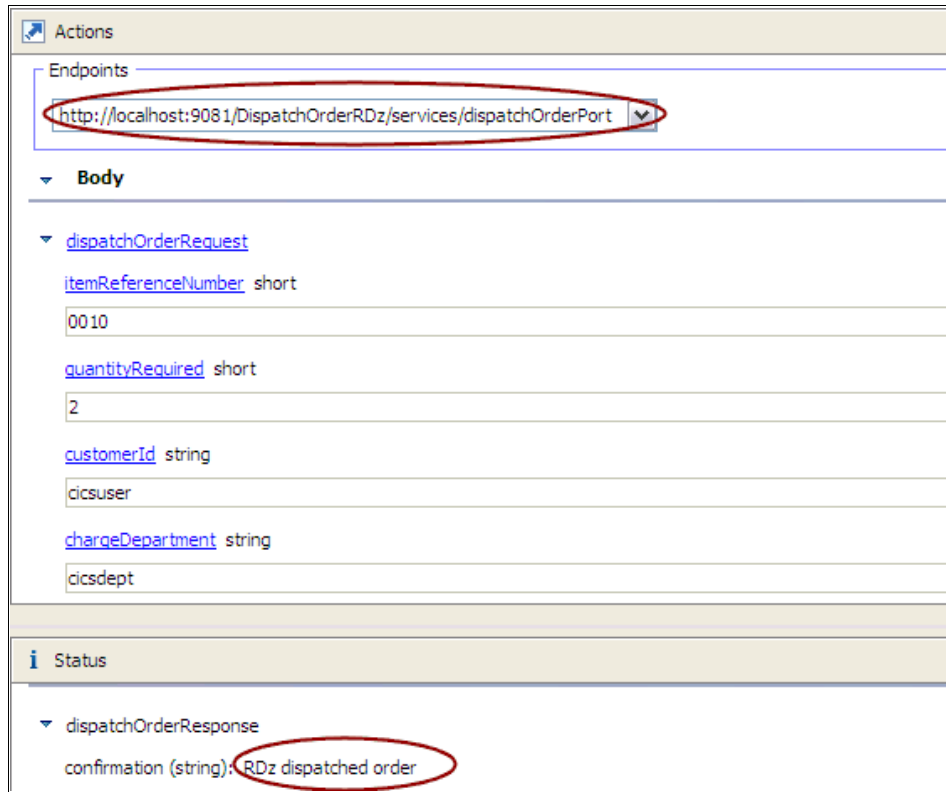


Figure 7-17 Web Services Explorer testing newly created Web service

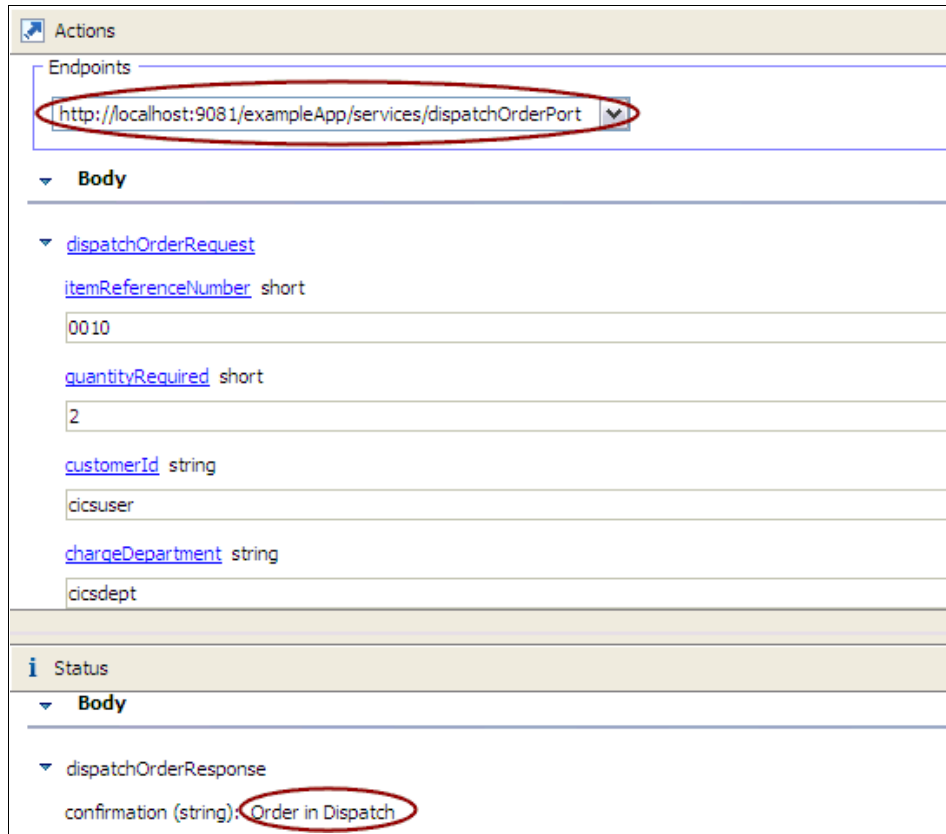


Figure 7-18 Web Services Explorer testing provided Web service

7.3.4 Test the Web service using the CICS sample application

The following steps show how to invoke this new Web service from CICS.

1. Open the configuration application of your CICS catalog manager example application (Figure 7-19). Make sure Outbound WebService is set to YES and insert the Outbound WebService URI of your end point. Use the TCP/IP address of the workstation on which you are running RDz. Refer to 7.2.3, “Test the requester application” on page 180 for further details. Press Enter to confirm your change.

Important: URIs are case sensitive. If the characters you type are transformed to upper case after saving then you should set your terminal to mixed case by typing CE0T Tra.

This capitalizes transaction IDs only.

```
CONFIGURE CICS EXAMPLE CATALOG APPLICATION

      Datastore Type ==> VSAM           STUB|VSAM
Outbound WebService? ==> YES           YES|NO
      Catalog Manager ==> DFHOXCMN
      Data Store Stub ==> DFHOXSDS
      Data Store VSAM ==> DFHOXVDS
      Order Dispatch Stub ==> DFHOXSOD
Order Dispatch WebService ==> DFHOXWOD
      Stock Manager ==> DFHOXSSM
      VSAM File Name ==> EXMPCAT
Server Address and Port ==> 9.12.4.75:30072
Outbound WebService URI ==> http://wtsc66.itso.ibm.com:9108/DispatchOrde
                        ==> rRDz/services/dispatchOrderPort
                        ==>
                        ==>
                        ==>
                        ==>
```

Figure 7-19 Configure CICS CATALOG application

2. Start the catalog manager by typing EGUI, select option 2 with, for example, element 0010, which will give you a panel similar to Figure 7-20. Insert some parameters and press Enter.

```
CICS EXAMPLE CATALOG APPLICATION - Details of your order

Enter order details, then press ENTER

Item      Description                                Cost      Stock On Order
-----
0010      Ball Pens Black 24pk                            2.90      0095 000

Order Quantity: 2
User Name: CICSUSER
Charge Dept: CICSDEPT
```

Figure 7-20 Dispatch an order in CICS

CICS tells you that your order has been placed successfully (Figure 7-21).

```
CICS EXAMPLE CATALOG APPLICATION - Main Menu

Select an action, then press ENTER

Action . . . .  1. List Items
                  2. Order Item Number
                  3. Exit

ORDER SUCCESSFULLY PLACED
```

Figure 7-21 Successful dispatched order in CICS.



Componentization

In this chapter, we consider the concept of componentization and how software components can be used to improve traditional CICS application development. We also look at some of the new features of CICS TS 4.1 that extend the benefits of Web services for ordinary applications.

8.1 CICS applications as components

Part of the value of Web services is the ability to reuse existing applications as building blocks within a wider context. The existing applications might be complicated and involve many CICS PROGRAMs, but to the outside world they are simple XML-based interfaces. A lot of complexity is hidden behind the Service interface. The associated WSDL document describes the interface to the application without any implementation complexities.

Traditional CICS application development has some similarities. CICS applications are normally made up of multiple executable modules. These modules might interact through the EXEC CICS LINK command, through COBOL call statements, and through similar mechanisms. Code can be built into libraries and shared between applications, or made into callable routines.

Applications in turn can interact with each other, typically using further EXEC CICS LINK commands. It is this application-to-application interaction that introduces a subtle problem. It can be difficult to determine the logical boundaries between applications. There is no way to know whether an EXEC CICS LINK between CICS PROGRAMs marks a boundary within an application, or between applications. This confusion is something that CICS administrators have learned to live with, but it does have an impact. It is not unusual for CICS administrators to experience significant difficulties in understanding what their PROGRAMs do, how they interact, and what impact changes will introduce. See Figure 8-1 for a typical CICS system example, and all the complexities of programs linking to each other.

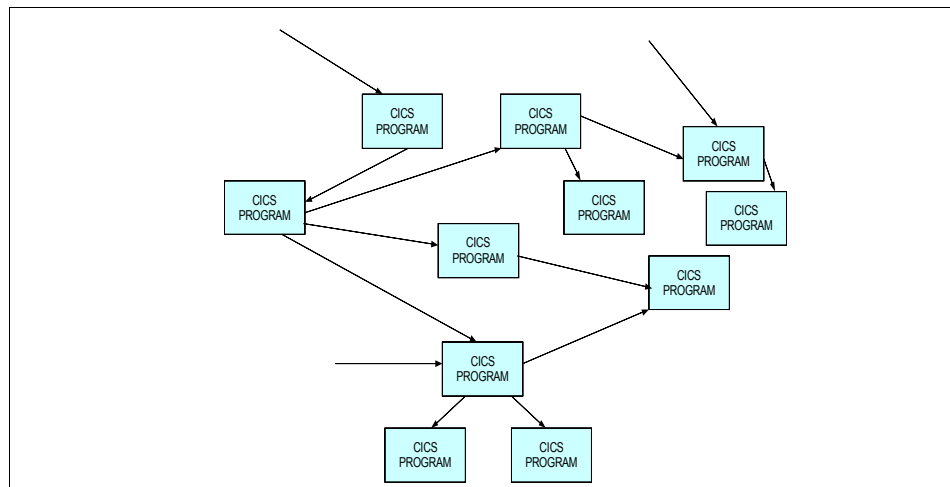


Figure 8-1 A typical CICS system with many PROGRAMs linking to each other

One way to reduce this problem is to introduce a formal boundary for application-to-application interactions. A target CICS application can be exposed as a Service, and be invoked from another application. The LINK command is used within applications, but the INVOKE command is used between applications. Ordinary CICS applications can be treated as Services, with the associated advantages and characteristics. See Figure 8-2, for an example of CICS applications being treated as services.

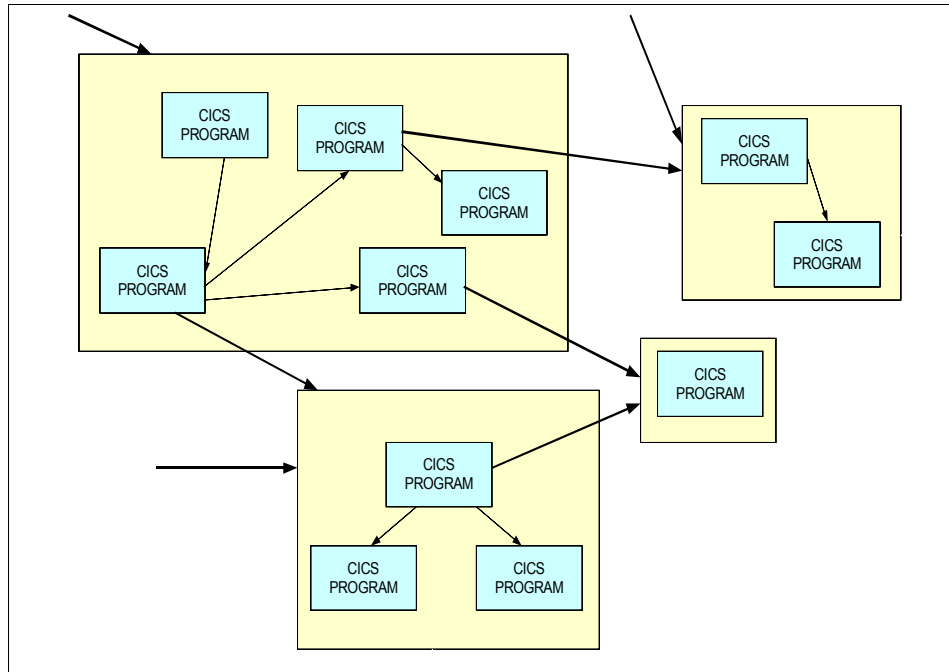


Figure 8-2 The same CICS system with component boundaries identified

In this chapter *Component* is used to describe a Service used within a CICS region.

8.2 Locally optimized Web services

One technique that can be used to formalize application boundaries within a CICS region is to deploy the target applications as Web services. This is done using the same techniques as would normally be used when exposing an existing PROGRAM as a Web service. The difference is that both the requester

and provider applications are local to the same CICS system. The client program uses the EXEC CICS INVOKE WEBSERVICE command to call the target Web service.

There is an important optimization in CICS that facilitates componentization. As part of the INVOKE processing CICS looks to see whether the named WEBSERVICE resource is hosted in a requester mode PIPELINE or a provider mode PIPELINE. For normal uses of EXEC CICS INVOKE WEBSERVICE the target is in a requester mode PIPELINE and this results in a SOAP message being sent to the remote Service. But, if the WEBSERVICE is hosted in a provider mode PIPELINE, CICS behaves differently. CICS instead issues an EXEC CICS LINK to the PROGRAM that implements the local WEBSERVICE.

An application can issue EXEC CICS INVOKE WEBSERVICE, but CICS can automatically optimize that into EXEC CICS LINK.

This results in several benefits compared to coding EXEC CICS LINK in the source application:

- ▶ CICS knows about the application boundary. The boundary can be seen in CICS monitoring, statistics, and diagnostics. CICS has the opportunity to add value to the invocation by modifying the processing based on configuration settings.
- ▶ The client application is only loosely bound to the target service. For example, the target PROGRAM can be changed by the administrator based upon the WEBSERVICE resource definition. If there is ever a requirement to move the target service to another platform, a simple administrative change in the CICS region will result in the requester making an outbound SOAP call to the target Web service without any application changes being required.
- ▶ A formal WSDL-based description of the interface exists. The target service might optionally be exposed to the outside world through the SOAP interface, but this is not a requirement. If at some point in the future there is a requirement for it to become an external Web service, no effort is required to implement it. The work has already been done.

In CICS TS 4.1 a new API command has been introduced called EXEC CICS INVOKE SERVICE. This is a synonym for EXEC CICS INVOKE WEBSERVICE. It emphasizes that a Service does not have to involve external interactions, a Service can be local.

8.3 Using WSDL to describe COBOL components

If you use the INVOKE command to call a Web service, a WSDL description of that service will exist. It might have been generated by DFHLS2WS (or RDz) from an existing set of copybooks. This is likely to be the case when existing applications are turned in to components.

For top-down development there is an opportunity to use modern software design tools as part of the development process for new CICS components.

For example, you could consider using Rational Software Architect (RSA) and the Universal Modelling Language (UML) to describe the abstract interfaces between the desired software components. After you have a UML-based description of the interfaces to the new component, you can generate a WSDL-based description from it. After you have a WSDL-based description of the interface you can generate COBOL bindings using DFHWS2LS (or RDz).

From there you can implement a CICS Service that exactly conforms to the architectural intentions encoded by the application architect in the UML document. The CICS components are just one more box in the architect's palette, with the implementation complexities abstracted away as with any other Web service.

8.4 Further Options with CICS TS 4.1

Locally-optimized Web service invocations offer the best possible performance for an EXEC CICS INVOKE call, but they do involve some limitations compared to remote Web service invocations:

- ▶ The requester and provider must share the same copybooks. This might not be a significant limitations, but it does mean that both the requester and the provider applications must be implemented in the same programming language.
- ▶ The requester and provider applications will run in the same CICS unit of work (UOW). If an abend occurs in the provider application, this will also back-out changes made in the requester application unless the application manages its own transactions.
- ▶ The CICS pipelines are not used. This means that none of the handler programs associated with the PIPELINE resource are called. None of the diagnostics normally associated with a pipeline will be available. The optimization from INVOKE WEBSERVICE to LINK completely optimizes the pipeline out of the processing.

- ▶ Many of the control containers that are normally available to the provider mode Web service on the default channel are unavailable. This includes and containers that normally hold XML data such as DFH-REQUEST, DFH-RESPONSE and DFHWS-BODY.

Note: The DFHWS-URI, DFHWS-OPERATION and DFHWS-SOAPACTION containers will all be available.

In CICS TS 4.1 there is a new option available that allows a compromise between performance and flexibility for local components. The new capability involves using a WEBSERVICE resource in a requester mode PIPELINE together with a special URI format that gives the application control over what processing should occur. This results in two new options that weren't available with earlier version of CICS:

- ▶ Use INVOKE SERVICE calls to link to a local CICS component after the requester mode pipeline processing has been performed.
- ▶ Use INVOKE SERVICE calls to link to a local CICS component after both a requester mode pipeline and a provider mode pipeline have been called, but without sending the request out to the network.

These two new scenarios are discussed in the following sections. There is a third new URI format to allow chaining of requester mode PIPELINES, but that capability is not discussed here.

8.4.1 Linking to a target PROGRAM from a requester mode PIPELINE

In this scenario the requester application calls:

```
EXEC CICS INVOKE SERVICE(servicename) OPERATION(operationName) URI(uri)
```

specifying a URI in the form:

```
cics://PROGRAM/program
```

where program is the name of the target CICS PROGRAM to which to link. CICS finds the SERVICE or WEBSERVICE identified by servicename and starts the processing through the associated requester mode PIPELINE. At the end of the pipeline processing CICS issues an EXEC CICS LINK to the program specified in the URI.

This scenario is similar to the locally optimized scenario, except that the requester mode pipeline is used and the associated pipeline handler programs are called by CICS. However, the application data passed to the target PROGRAM is the same data provided by the source application.

See Figure 8-3 for example for an example of linking to a program from a requester mode pipeline.

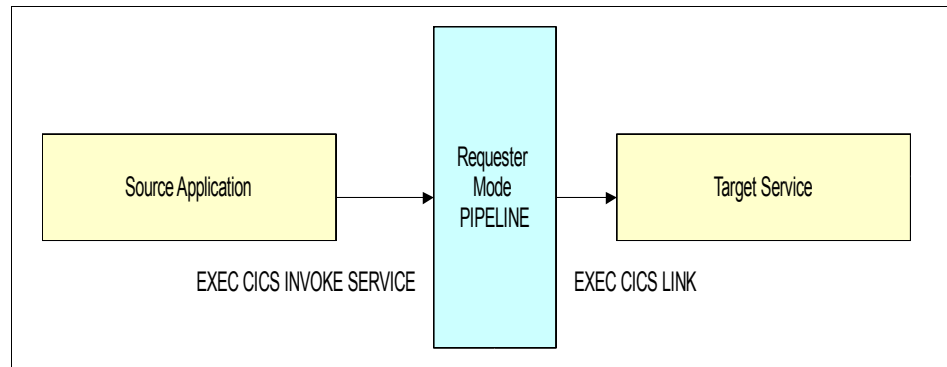


Figure 8-3 LINKING to a PROGRAM from a requester mode PIPELINE

This provides some additional flexibility that the locally optimized INVOKE doesn't offer, but there are some restrictions remain.

8.4.2 Invoking a local SERVICE from a requester mode PIPELINE

In this scenario the requester application calls:

```
EXEC CICS INVOKE SERVICE(servicename) OPERATION(operationName) URI(uri)
```

specifying a URI in the form:

```
cics://SERVICE/service?targetServiceUri=targetServiceUri
```

where *service* is the name of a provider mode CICS SERVICE (typically a WEBSERVICE) to invoke and *targetServiceUri* is the URI associated with the provider mode SERVICE.

CICS finds the SERVICE or WEBSERVICE identified by the *servicename* and starts the processing through the associated requester mode PIPELINE. At the end of the pipeline processing CICS locates the provider mode SERVICE or WEBSERVICE identified in the URI by *service*. CICS then starts the provider mode PIPELINE associated with this service using the specified *targetServiceUri*.

This scenario is similar to making an ordinary EXEC CICS INVOKE WEBSERVICE call with a URI that addresses a provider mode WEBSERVICE hosted in CICS, except that the call through the networking layer of code is optimized out of the execution path. See Figure 8-4 for example of invoking a local service from a requestor mode pipeline.

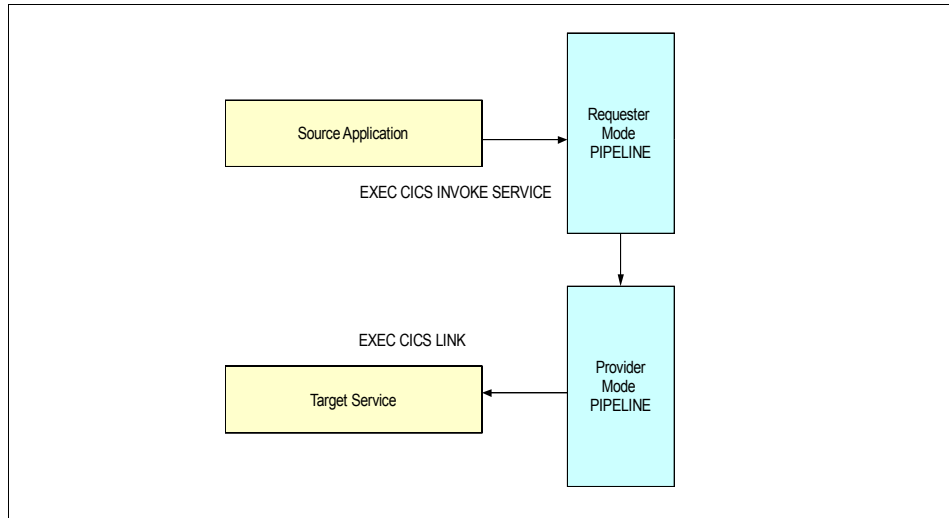


Figure 8-4 INVOKING a local service from a requestor mode PIPELINE

This scenario offers a a lot of flexibility for scenarios where both the requester and provider components are hosted in CICS, but it does involve generating XML from application data and parsing that XML back into application data. This is in addition to the cost of running both the requester mode pipeline and the provider mode pipeline.



New SOA patterns for CICS TS V4.1

9.1 Service Component Architecture

in this section we discuss the fundamentals of the Service Component Architecture (SCA), which is a new service-oriented architecture (SOA) based programming model available as part of CICS TS V4.1.

SCA has similar goals and ideals to Web services, and many of the concepts and benefits overlap, but the implementation is quite different.

SCA is based on the idea that business function is provided as a series of services (or components), which are assembled together to create solutions that serve a particular business need. SCA is both platform and programming language neutral, and is suitable for creating new business services by means of composition of both new and existing components. SCA provides a model for the composition of services and for the deployment of service components, including the reuse of existing applications.

In the following sections we discuss the concepts and terminology of SCA.

9.1.1 Introduction to SCA

In this section we introduce SCA, basic concepts, and components.

Basic concepts

SCA is designed around *components*, which encapsulate services that can be invoked. SCA components expose interfaces that define the information that must be supplied to invoke a service. Each interface can be defined using either a Java Interface, or a WSDL Port Type. The CICS implementation of SCA is based around WSDL, so that is the interface format upon which we will concentrate.

SCA components can also invoke services that are exposed by other SCA components. We say that one SCA component references the other. As with Web services, service invocation is an entirely black-box affair. It does not matter to one component how another is implemented. An interface can have one or more *operations* (also sometimes called *functions*). When a component is invoked, an operation name is specified. These operations can be one-way (in-only in WSDL) or two-way (in-out in WSDL).

The major difference between SCA and Web services is that SCA focuses on the assembly of composite services, whereas Web services focuses on wire-level interoperability between services. SCA can be bound to Web services, but it does not require an XML-based messaging system.

Figure 9-1 shows a simplified UML model of the SCA architecture.

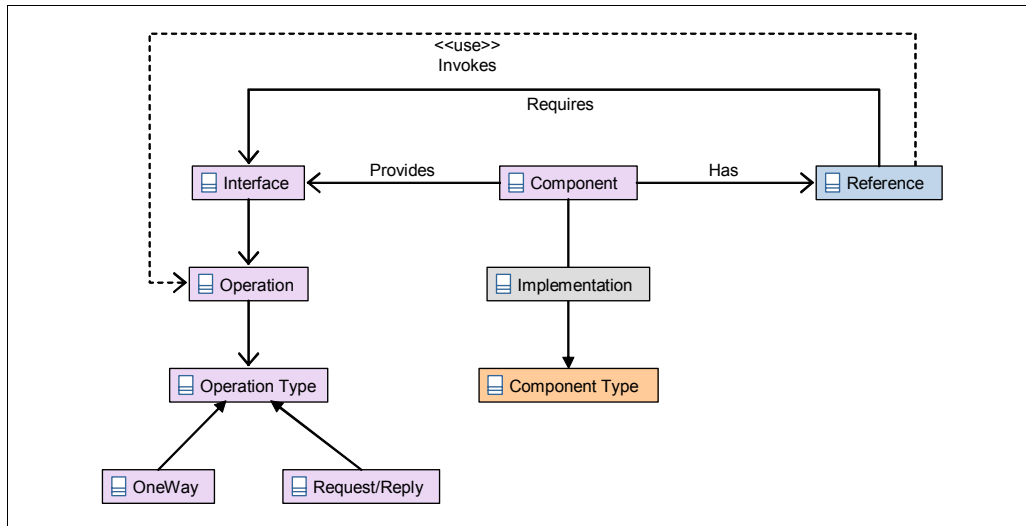


Figure 9-1 Simplified SCA component UML model

The Open Service Oriented Architecture group documents the SCA model:

<http://www.osoa.org/display/Main/Home>

Collections of components are referred to as *composites*. SCA composites are defined using Service Component Definition Language (SCDL) files. As with WSDL, SCDL is usually machine-generated and not intended for direct use by application developers. You would normally use tools such as RDz to produce the SCDL.

SCA components

A *component* is the basic unit of composition in SCA. Components are used to define which services are available and can be invoked within the SCA runtime environment.

Example 9-1 on page 206 shows a fragment of SCDL that defines a component called `SampleComponent` that is instantiated by a Java class called `MyJavaClass`. This class implements the methods that are specified in interface `MyInterface`. A property has also been specified that will be available to the SCA component at runtime. The service name is the name of business function provided by the `SampleComponent`. This example uses a Java class, therefore it is not appropriate for deployment to CICS, but could be used with other SCA compliant application environments.

Example 9-1 Component element

```
<component name="SampleComponent">
  <implementation.java class="ibm.com.MyJavaClass" />
  <property name="pname">attributes</property>
  <service name="MyService">
    <interface.java interface="ibm.com.MyInterface" />
  </service>
</component>
```

SCA services

The service element allows you to define which specific business function or services the component (or composite) provides. A *reference* can be used to describe dependencies on services provided by another component. Both service and reference elements can be further specified using the interface and binding elements.

SCA operations

SCA supports two types of operations.

- ▶ One-way operations (also called fire-and-forget) have data that flows into the SCA component from the caller, but the SCA component returns no data back to the caller.
- ▶ Two-way operations (also called request/reply) have data flowing into the component (request), and then back to the caller (reply).

Some architectural models call one-way operations *asynchronous* and two-way operations *synchronous* (Unified Modelling Language [UML] uses these terms). SCA does not impose this concept.

SCA composites

A composite is the deployable unit for SCA. It can contain components, services, references, property declarations, and the wiring that describes the connection between these elements.

A composite can be made up of a number of components that are wired together. The composite exposes an external interface as Services, and can call external services through References.

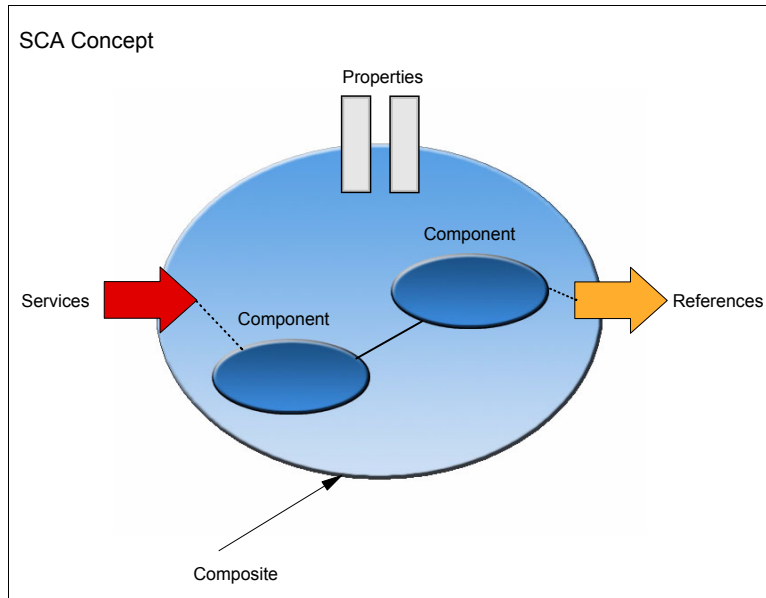


Figure 9-2 Composites and components

The SCDL fragment in Example 9-2 shows a typical composite definition. The sample shows a composite named `sampleCatalog`. The composite exposes a service called `Catalog`. The business methods are defined within a component element called `CatalogComponent`. The component contains the implementation of an interface called `ibm.com.sampleCatalog`, which specifies the available operations. The example also shows that the externally visible service is actually a Web service. The `binding.ws` element defines the Web services-based access method used to invoke the service.

Example 9-2 Composite element

```

<composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
  targetNamespace="http://ibm.com/sampleCatalog"
  name="sampleCatalog" >
  <service name="Catalog" promote="CatalogComponent">
    <interface.java interface="ibm.com.sampleCatalog"/>
    <binding.ws port="http://ibm.com/Catalog
      wsdl.endpoint(Catalog/CatalogWS)"/>
  </service>
  <component name="CatalogComponent">
    <implementation.java class="ibm.com.sampleCatalog"/>
  </component>
</composite>

```

9.2 CICS TS V4.1: Implementation of SCA

In this section we look at CICS implementation of SCA.

9.2.1 BUNDLE resources

CICS TS V4.1 introduces a new packaging format called a *bundle* for deploying some types of CICS resources, including the artifacts needed for SCA. A BUNDLE resource represents a group of artifacts that can be installed and managed together. Conceptually it is similar to a Java ARchives (JAR) file that might be familiar from Java environments.

The BUNDLE involves one or more files installed in the UNIX file system, the most important of which is the Manifest file. This is an XML file that acts as an index for identifying the rest of the contents of the bundle. A BUNDLE resource in CICS encapsulates the artifacts in the bundle directory on the UNIX file system.

Unlike traditional CICS CSD groups, the relationship between the resources installed from a bundle persist after installation. This means that you can manage all the related resources as a single composite entity. For example, if you disable a BUNDLE resource because you want to stop an application from running, CICS disables all of the related application resources for you. You can manage the contents of a bundle and its constituent parts using the CICS Explorer.

Bundles for SCA composites are normally created using RDz.

9.2.2 Creating services from existing CICS applications

You can create two types of service from your CICS applications:

- ▶ Channel-based services that are local to a CICS environment
- ▶ XML-based services that are exposed externally as Web services.

In both cases, the application program that you expose as a service is defined in the <Implementation> element of the SCDL for the SCA component.

Channel-based SCA services

Channel-based services are CICS applications that are deployed as SCA components and assembled together using a tool such as RDz. These services are available to other CICS applications that use the INVOKE SERVICE API command and pass binary data in containers on a channel.

The interface to the application program is described in WSDL. For a channel-based service, the binding is described in the `binding.cics` section of the SCDL. There are no WSBind files required in this scenario.

XML-based SCA services

XML-based services are SCA services that wrapper CICS WEBSERVICE resources and use a SOAP-based messaging protocol. XML-based services are available to CICS applications that use the `INVOKE SERVICE` API command, but they are also available to external requesters from the network. You can either create Web services using the Web Services Assistant, or you can use RDz. If you use RDz, you can also create an SCA component for your Web service. There are some advantages to creating a component from a Web service:

- ▶ You can reuse the components to develop future composite applications rapidly using RDz.
- ▶ You can use SCDL to describe the Web service, thereby moving the configuration information out of the application and WSBind file and into the SCDL. This can make it easier to implement changes without having to change the WSBind file. For example, if you want to run a Web service under different default transactions and user IDs, you can change the SCDL without having to regenerate the WSBind file.

The interface to the application program is described in WSDL. For an XML-based service, the binding is described in the `binding.ws` section of the SCDL. The bundle also includes the WSBind files that allow CICS to transform the application data to SOAP messages.

9.2.3 Deploying SCA services

In the following sections, we look at deploying SCA services.

The bundle resource

SCA bundles are created using RDz. A bundle contains the resources that are required by the service, typically the SCDL and any WSBind files used. Any system resources that the service requires can be defined as prerequisites in the bundle manifest file, but they are not included in the bundle itself.

After your bundle has been created you must deploy it to the UNIX file system as a directory structure. Then you must create and install a CICS `BUNDLE` resource that points at this directory. CICS then installs each of the SCA composites that are referenced from the bundle manifest file.

Domains

SCA domains are the runtime environments for the assembled business services. To use your application composites in CICS you have to deploy it using a bundle. Each bundle has a scope, which represents the deployment domain. By default this value is empty, but you can specify a specific BASESCOPE to act as a domain (or naming context) for the bundle if you want to do so.

9.2.4 RDz SCA tooling

The Enterprise Service Tools perspective in RDz 7.6 has been extended to provide views and wizards that allow you to develop CICS SCA projects.

You can use the RDz SCA tooling to compose SCA objects visually by wiring services and references together. A wire is a connector that passes control and data from a component to a target.

9.2.5 Creating and deploying an SCA service from an existing CICS application

The steps in Figure 9-3 on page 211 can be used to design, implement, compose, and run a CICS SCA application using RDz SCA tooling.

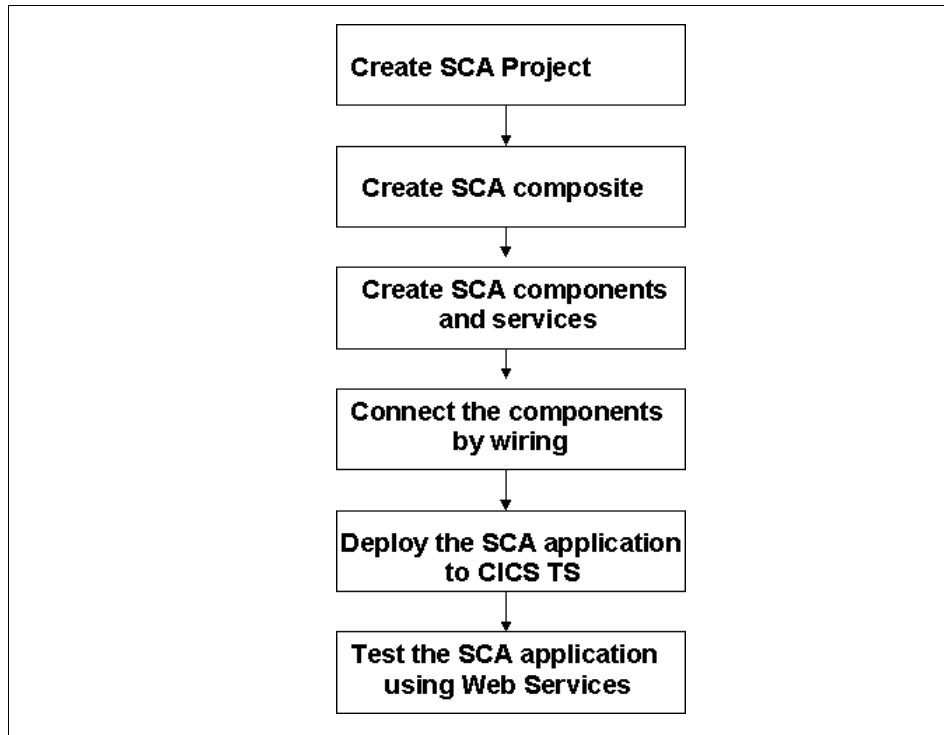


Figure 9-3 Steps to design, implement, compose and run a CICS SCA application

1. Create SCA Project

You can use the SCA project wizard to create a new CICS SCA project. The implementation type for the SCA component can be set to CICS

2. Create SCA composite

The composite wizard allows to configure the composite to be created.

3. Create SCA components and services

Create a component type from CICS application source code.

4. Connect the components by wiring

The composite editor can be used to visually connect the components.

5. Deploy the SCA application to CICS TS

The deploy bundle wizard can be used to deploy the assembled application to the UNIX file system

You can test the SCA service using the Web Services Explorer.



Hints and tips

This chapter focuses on hints and tips in the following areas:

- ▶ Custom handlers for pipelines
- ▶ SOAP fault API
- ▶ Handling variable cardinality elements
- ▶ Using WSDL generated by Rational Developer
- ▶ WSDL types not supported by WS2LS
- ▶ Problem determination
- ▶ XML parsing in CICS application

10.1 Custom handlers programs for pipelines

CICS supplies a set of special purpose SOAP header and message handler programs for use within a pipeline configuration. These can enhance CICS to implement external specifications such as WS-Security, WS-AT or WS-Addressing (at CICS TS V41).

You can also create your own handler programs to satisfy local requirements. For example, you could have a logging service that records SOAP messages, or a diagnostics service that e-mails diagnostics to an operator in the event of a failure. The advantage of using a handler program to implement these sorts of requirements is that the handler will be active for all Web services in the PIPELINE.

10.1.1 A simple example handler program

In this section we explore creating a simple handler program that uses the WEB API to discover the value of a HTTP header. We then write that value into a container on the current channel.

A problem: Operation resolution

A single Web service might implement many different operations. A common requirement in provider mode applications is the need to know which of the operations has been invoked. CICS makes the operation name available to the application in the DFHWS-OPERATION container. However, this container is populated by the CICS-supplied application handler (DFHPITP), so if you do not use the CICS-supplied application handler program, or if you have handlers that run in the pipeline before DFHPITP has executed, you might have difficulty discovering which operation is being invoked.

The official way to resolve the operation is to consider the signature of the body of the SOAP message. This involves calculating the operation based on the pattern of XML tags within the SOAP body. This is a complicated operation to perform. However, many WSDL authors offer a useful clue by ensuring that the operation name is encoded in a special HTTP header called the SOAPAction header. In many cases you can read this HTTP header and use the value to infer the operation that was called, without having to parse any XML.

CICS does this for you. You can find the value of the SOAPAction HTTP header in the DFHWS-SOAPACTION container. However, for the purposes of this example we pretend that you need to discover the SOAPAction header programmatically using the WEB API.

Example 10-1 shows an HTTP header including the SOAPAction field. In this instance a URI value is supplied. This information could be used by the service provider to determine the intent of the request.

Example 10-1 HTTP header with SOAPAction URI

```
POST /exampleApp/inquireCatalog HTTP1.1
Content-Type: text/xml; charset="UTF-8"
SOAPAction: "http://itsocatalog.org/index#MyMessage"
```

The value that is specified in the SOAPAction header need not be a fully qualified URI. A single word might be used instead of a URI (Example 10-2).

Example 10-2 HTTP header with non-URI SOAPAction value

```
POST /exampleApp/inquireCatalog HTTP1.1
Content-Type: text/xml; charset="UTF-8"
SOAPAction: "index"
```

PIPELINE states

A handler program can be written to obtain the value of the SOAPAction URI (if present) and place it into the SOAPACTION container. The application program can use this container to retrieve this value if required.

The handler program will be called twice during the normal execution of a Web service request: once during the inbound phase and once during the outbound phase. However, the handler only has to perform steps during the inbound phase of the request. At runtime, the current phase of execution is stored in container DFHFUNTION. Possible values within this container are:

- ▶ RECEIVE-REQUEST
- ▶ SEND-RESPONSE
- ▶ SEND-REQUEST
- ▶ RECEIVE-RESPONSE
- ▶ PROCESS-REQUEST
- ▶ NO-RESPONSE
- ▶ HANDLER-ERROR

More information about these states can be found in *CICS Transaction Server for z/OS Internet Guide Version 3 Release 1*, SC34-6450.

We can check the value in this container and test whether it equals the literal value RECEIVE-REQUEST. If the value in the container does not equal this value, then the message handler is not being invoked during the desired phase of execution and we can return execution back to the CICS pipeline handler. See Example 10-3 on page 216.

Example 10-3 Checking execution phase

```
*****
CHECK-INBOUND SECTION.
*CHECK WE ARE EXECUTING DURING THE RECEIVE-REQUEST PHASE
EXEC CICS GET CONTAINER('DFHFUNCTION  ')
      INTO(FUNC-BUFFER)
      RESP(RESP)
      RESP2(RESP2)
      END-EXEC.
      IF FUNC-BUFFER NOT EQUAL TO 'RECEIVE-REQUEST'
      EXEC CICS RETURN END-EXEC.
CHECK-INBOUND-END. EXIT.
*****
```

Using the WEB api to access HTTP headers

When the execution phase has been determined, the EXEC CICS WEB READ command can be used to obtain the SOAPAction header. The retrieved value can be placed in a new container on the current channel. The name of the container used to store the SOAPAction URI value must be known to both the message handler and the application program. See Example 10-4.

Example 10-4 Retrieving the SOAPAction header and storing it in a container

```
*****
RETRIEVE-SOAP-ACTION SECTION.
*RETRIEVE THE SOAPACTION HEADER
EXEC CICS WEB READ HTTPHEADER(HEADER-NAME)
      NAMELENGTH(10)
      VALUE(URI-BUFFER)
      VALULENGTH(LENGTH OF URI-BUFFER)
      RESP(RESP)
      RESP2(RESP2)
      END-EXEC.

RETRIEVE-SOAP-ACTION-END. EXIT.
*****
ADD-HEADER-TO-CONTAINER SECTION.
*ADD THE SOAPACTION HTTP HEADER TO A CONTAINER
*DID THE LAST OPERATION SUCCEED
      IF RESP EQUAL TO DFHRESP(NORMAL)
      EXEC CICS PUT CONTAINER(CONT-NAME)
            FROM(URI-BUFFER)
            FLENGTH(LENGTH OF URI-BUFFER)
            RESP(RESP)
```

```

                RESP2(RESP2)
                END-EXEC
            END-IF.
        ADD-HEADER-TO-CONTAINER-END. EXIT.
    *****

```

Setting the PIPELINE state after execution

When this message handler is called, containers DFHREQUEST and DFHRESPONSE exist on the channel. This gives the message handler the ability to either allow processing to continue to the next handler in the pipeline or to construct a response to the request and terminate any further request processing. In this case we want execution to continue, so DFHRESPONSE must be deleted from the channel. Otherwise, when the handler returned execution to CICS, both DFHREQUEST and DFHRESPONSE would still exist on the channel. This situation causes ambiguity and will cause CICS to re-call the handler for exception processing. See Example 10-5.

Example 10-5 Deleting container DFHRESPONSE

```

    *****
    DELETE-DFHRESPONSE SECTION.
    *Deleting DFHRESPONSE will ensure that the message is
    *passed to the next stage of the pipeline
        EXEC CICS DELETE CONTAINER('DFHRESPONSE    ')
            RESP(RESP)
            RESP2(RESP2)
            END-EXEC.
    DELETE-DFHRESPONSE-END. EXIT.
    *****

```

Changes to the PIPELINE configuration file

This application (SOAPACT) was integrated into the pipeline by using the pipeline configuration file shown in Example 10-6 on page 218.

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline>
  <service>
    <service_handler_list>
      <handler>
        <program>SOAPACT</program>
        <handler_parameter_list/>
      </handler>
    </service_handler_list>
    <terminal_handler>
      <cics_soap_1.1_handler/>
    </terminal_handler>
  </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

Attention: Applicable message handlers are executed in the order that they are listed within the pipeline configuration file. This is important if a handler requires another handler to have executed.

The SOAP specification 1.2 has removed the requirement for the HTTP SOAPAction header field to be present in a SOAP request.

10.1.2 Handling state information

Sometimes it is desirable to implement a stateful service. For example, a pagination service that maintains a cursor to an entry in a data stream and always returns the next 10 records. Or a service that maintains the concept of a session across multiple uses.

A method to allow stateful transactions to take place would require three steps:

1. A method to generate unique session tokens
2. An initial message to obtain a session token
3. A mechanism for subsequent messages to reference the session token.

You could embed the token in the application data. This will require the client to know to return that token on subsequent calls. You could embed the token in a SOAP header, and require the infrastructure to propagate the token. You could make use of WS-Addressing to manage session state.

In general it is best to maintain session tokens in the applications themselves as this allows the flexibility to propagate and use the tokens wherever they are needed.

To help you implement this mechanism in your Web services, a suggested mechanism for persisting state information over several transactions is mentioned in *CICS Transaction Server for z/OS Internet Guide Version 3 Release 1*, SC34-6450.

This book outlines two sample modules, DFH\$WBST and DFH\$WBSR, which provide a mechanism for managing state tokens. Functions are provided for the creation and destruction of tokens as well as storing and retrieving state data given a token value. A method is also provided for the cleanup of state information and tokens that have not been used for a set period of time.

10.1.3 Propagating user identity tokens

Often there is a requirement for a specific user of a Web service to identify themselves to CICS for authentication.

There are a number of mechanisms by which this can be done. In many scenarios the user will authenticate to an intermediate server such as WebSphere Application Server, which, in turn, communicates with CICS. In this case you are advised to use WS-Security to implement the identity propagation between WebSphere Application Server and CICS.

You can also use transport layer encryption (SSL/TLS) to secure the communication channels. Advanced deployments could make use of a WebSphere DataPower® appliance to sit between CICS and the external network.

Simpler deployments might make use of HTTP basic authentication to send a simple user ID and password to CICS.

Application developers rarely need to be concerned with these issues as the security is normally implemented in the infrastructure. Typically the systems programmer will enable identity propagation in WebSphere Application Server and in CICS, and the applications will run as before. The identity information is flowed wither as HTTP headers or as SOAP headers without requiring application changes.

For further information you are advised to refer to the CICS Information Center.

10.2 The SOAP fault API

The SOAP specification provides a mechanism by which error diagnostics can be returned from a provider to a requester. This mechanism is the SOAP fault message. CICS automatically returns fault messages to requesters in the event of an application abend or system failure, but SOAP aware provider mode applications can return application specific fault messages programmatically.

CICS supplies three EXEC CICS commands that can be used to create a SOAP FAULT message:

- ▶ SOAPFAULT CREATE
Creates a new SOAPFAULT object.
- ▶ SOAPFAULT ADD
Adds extra information to the current SOAPFAULT object.
- ▶ SOAPFAULT DELETE
Deletes the current SOAPFAULT object.

Find more information about these commands in the CICS Information Center. These CICS APIs require one of the CICS-supplied SOAP handlers to be in the execution stack at the time the API is driven. This means that the API is not available to some handler programs in the pipeline, but is available for Web services. The API also requires that the application programs were linked to using a Channel.

10.2.1 How to create a SOAP Fault in an application

Example 10-7 shows an application throwing a SOAP fault programmatically.

Example 10-7 Example of creating a SOAP fault

```
dc1 msgDetail char(*)
constant('<ati:ExampleFaultxmlns="http://www.example.org/faults"
xmlns:ati="http://www.example.org/faults">Detailed error message goes
here.</ati:ExampleFault>');
dc1 msgFaultString char(*) constant('Something went wrong');
```

```
EXEC CICS SOAPFAULT
                CREATE CLIENT
                DETAIL(msgDetail) DETAILLENGTH(length(msgDetail))
                FAULTSTRING(msgFaultString)
                FAULTSTRLEN(length(msgFaultString))
                RESP(RESP) RESP2(RESP2);
```

10.2.2 Parsing SOAP Fault messages in CICS TS V4.1

In this section we consider parsing SOAP fault messages received by a Web service requesting application in CICS. For most purposes it is sufficient to know that a fault has been returned. This is indicated by an INVREQ response from the INVOKE command with a RESP2 value of 6. If you want to access application-specific diagnostics embedded within the SOAP fault message then the following technique might be useful.

In CICS TS V3.1 and CICS TS V3.2 you have the option of reading the DFHWS-BODY container returned by CICS to access the XML representation of the SOAP fault message. You can then parse the XML data using a mechanism of your choice.

In CICS TS V4.1 there is a new XML parsing API command you can use to help in this process. Example 10-8 demonstrates the use of DFHSC2LS and the CICS TRANSFORM command. The example is included here both as an example of parsing SOAP fault messages, but also as an example of using the TRANSFORM command. You could go on to use the TRANSFORM command for other purposes.

DFHSC2LS

First, use DFHSC2LS to build COBOL bindings for SOAP faults. Start by down-loading a copy of the XML schema for SOAP Envelopes from here the following Web page:

<http://schemas.xmlsoap.org/soap/envelope/>

For example, you could save it to a location in the UNIX file system called /u/example/source/SOAP11.xsd.

Next, use DFHSC2LS to process the XML schema and create as output a set of COBOL bindings for the schema, and an XSDBind file in a bundle directory. You could use JCL similar to the following:

Example 10-8 JCL of DFHSC2LS

```
//EXAMPLE EXEC DFHSC2LS,  
  //INPUT.SYSUT1 DD *  
  MAPPING-LEVEL=3.0  
  ELEMENTS=Body,Fault  
  SCHEMA=/u/example/source/SOAP11.xsd  
  LANG=COBOL  
  PDSLIB=//EXAMPLE.COBOL.LIBRARY  
  PDSMEM=SOAP11  
  XSDBIND=SOAP11.xsdbind
```

```
BUNDLE=/u/example/output/bundle/SOAP11
LOGFILE=/u/example/output/logfile.log
*/
```

DFHSC2LS will create several COBOL language structures as shown in Example 10-9.

Example 10-9 Bindings for the 'Body' of the SOAP Envelope

```
03 Body.
   06 Body-num                PIC S9(9) COMP-5 SYNC.
   06 Body-cont               PIC X(16).
01 SOAP1101-Body.
   03 Body-xml-cont          PIC X(16).
   03 Body-xmlns-cont       PIC X(16).
```

This language structure contains bindings to allow any number of XML tags to appear within the SOAP Body. The number of tags found will be stored by CICS in the Body-num field, and information about the data will be stored by CICS in the container named by the Body-cont field. Each XML tag from the body will then have two fields associated with it that provide the XML for the tag in a container named in Body-xml-cont and the in-scope XML namespace declarations in a container named in Body-xmlns-cont. See Example 10-10.

Example 10-10 Bindings for the 'Fault' within the Body of a SOAP Envelope

```
03 Fault.
   06 faultcode-length       PIC S9999 COMP-5 SYNC.
   06 faultcode              PIC X(255).
   06 faultstring-length     PIC S9999 COMP-5 SYNC.
   06 faultstring            PIC X(255).
   06 faultactor-num         PIC S9(9) COMP-5 SYNC.
   06 faultactor.
     09 faultactor2-length   PIC S9999 COMP-5 SYNC.
     09 faultactor2         PIC X(255).
   06 detail3-num           PIC S9(9) COMP-5 SYNC.
   06 detail2.
     09 Xdetail-num         PIC S9(9) COMP-5 SYNC.
     09 Xdetail-cont       PIC X(16).
01 SOAP1102-Xdetail.
   03 detail-xml-cont       PIC X(16).
   03 detail-xmlns-cont    PIC X(16).
```

This language structure contains bindings to allow a single SOAP Fault to be parsed. It provides access to the 'faultcode', 'faultstring' and 'faultactor' fields, together with structures to map any number of XML tags found within the 'detail' section of the SOAP Fault.

Install the bundle

The next task is to install the bundle into CICS.

Create and install a BUNDLE definition such as the following:

```
BUNDLE: SOAP11
GROUP: EXAMPLE
DESCRIPTION: Bundle for mapping SOAP 1.1 SOAP Faults
BUNDLEDIR: /u/example/output/bundle/SOAP11
```

The BUNDLEDIR points to the location that was specified using the BUNDLE parameter of DFHSC2LS. If you run DFHSC2LS on a different z/OS image from the one used by CICS, you might need to copy the bundle directory to the target machine. In this case, you can use a different directory path and set the value of BUNDLEDIR accordingly. The name of the bundle is arbitrary. You can pick something other than SOAP11 if you prefer.

After installing into CICS you will have a BUNDLE resource called SOAP11 and an XMLTRANSFORM resource also called SOAP11. The XMLTRANSFORM name is derived from the value of the XSDBIND parameter of DFHSC2LS.

An Example SOAP fault message

Example 10-11 is an example SOAP fault message that might be found within the DFHWS-BODY container following an EXEC CICS INVOKE WEBSERVICE command.

Example 10-11 SOAP fault

```
<SOAP-ENV:Body>
  <SOAP-ENV:Fault xmlns="">
    <faultcode>SOAP-ENV:Server</faultcode>
    <faultstring>Conversion to SOAP failed</faultstring>
    <detail>
      <CICSFault xmlns="http://www.ibm.com/software/http/cics/WSFault">
        DFHPI1010 *** XML generation failed. A conversion error
        INVALID_PACKED_DEC) occurred when converting field 'example' for
        WEBSERVICE 'testWebservice'.
      </CICSFault>
    </detail>
  </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

This example is of a fault message created by CICS when a conversion error occurs. When this is processed by the TRANSFORM command CICS will set Body-num to 1 to indicate that there is a single XML tag within the Body tag. It will also set Body-cont to the name of a Container such as DFHPICC-00000001.

Inside container DFHPICC-00000001 CICS will place the names of two further container, for example DFHPICC-00000002 and DFHPICC-00000003.

Container DFHPICC-00000002 will contain the first tag from within the body. See Example 10-12.

Example 10-12 SOAP fault in container DFHPICC-00000002

```
<SOAP-ENV:Fault xmlns="">
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>Conversion to SOAP failed</faultstring>
  <detail>
    <CICSFault xmlns="http://www.ibm.com/software/htp/cics/WSFault">
      DFHPI1010 *** XML generation failed. A conversion error
      INVALID_PACKED_DEC) occurred when converting field 'example' for
      WEBSERVICE 'testWebservice'.
    </CICSFault>
  </detail>
</SOAP-ENV:Fault>
```

Container DFHPICC-00000003 will contain any in-scope namespace declarations. For example:

```
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

If the DFHPICC-00000002 container is then parsed through a second EXEC CICS TRANSFORM command, further output will be created by CICS. The faultcode and faultcode-length fields will be set to SOAP-ENV:Server and 15. The faultstring and faultstring-length fields will be set to Conversion to SOAP failed and 25. The faultactor-num field will be set to 0. The detail3-num field will be set to 1 to indicate that the optional detail tag is present in the fault. The detail2-num field will be set to 1 to indicate that there is one sub-tag within the optional detail tag. The detail2-cont field will be set to the name of a container, for example DFHPICC-00000004.

Container DFHPICC-00000004 will contain the names of two further containers, for example DFHPICC-00000005 and DFHPICC-00000006.

Container DFHPICC-00000005 will contain the first XML tag found within the detail section of the SOAPult. In this example it will contain the information shown in Example 10-13.

Example 10-13 CICSFault in container DFHPICC-00000005

```
<CICSFault xmlns="http://www.ibm.com/software/htp/cics/WSFault">
DFHPI1010 *** XML generation failed. A conversion error
(INVALID_PACKED_DEC) occurred when converting field 'example' for
WEBSERVICE 'testWebservice'.
</CICSFault>
```

Container DFHPICC-00000006 will contain the in-scope namespace declarations. For example:

```
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Application Code

To implement an application to parse the SOAP Fault you will have to do the following:

1. Call the TRANSFORM command to query the contents of the DFHWS-BODY container. For example:

```
EXEC CICS TRANSFORM XMLTODATA CHANNEL(channel-name)
XMLCONTAINER('DFHWS-BODY') NSCONTAINER('DFHWS-XMLNS')
ELEMNAME(element-name) ELEMNAMELEN(element-name-len) END-EXEC
```

2. If the element-name is set to Body then parse the container, if not then something went wrong. To parse the body use the following commands:

```
EXEC CICS TRANSFORM XMLTODATA CHANNEL(channel-name)
XMLTRANSFORM('SOAP11') XMLCONTAINER('DFHWS-BODY')
NSCONTAINER('DFHWS-XMLNS') DATCONTAINER('PARSEDBODY') END-EXEC
EXEC CICS GET CONTAINER('PARSEDBODY') SET(body-ptr) END-EXEC
```

3. Address the parsed data. For example:

```
SET ADDRESS OF Body TO body-ptr
```

Check Body-num to ensure there is at least one entry in the Body. Assuming that there is, read the container that lists the details. For example:

```
EXEC CICS GET CONTAINER(Body-cont) SET(body-cont-ptr) END-EXEC
SET ADDRESS OF SOAP1101-Body TO body-cont-ptr
```

4. Call TRANSFORM a second time to query the first tag from within the Body:

```
EXEC CICS TRANSFORM XMLTODATA CHANNEL(channel-name)
XMLCONTAINER(Body-xml-cont) NSCONTAINER(Body-xmlns-cont)
ELEMNAME(element-name) ELEMNAMELEN(element-name-len) END-EXEC
```

5. If the element-name is set to Fault, parse the container:

```
EXEC CICS TRANSFORM XMLTODATA CHANNEL(channel-name)
XMLTRANSFORM('SOAP11') XMLCONTAINER(Body-xml-cont)
NSCONTAINER(Body-xmlns-cont) DATCONTAINER('PARSEDFault') END-EXEC
EXEC CICS GET CONTAINER('PARSEDFault') SET(fault-ptr) END-EXEC
```

6. SET ADDRESS OF Fault TO fault-ptr

You can now query the data from the fault. For example, you might find the faultstring to be useful. If you want to parse application specific details from the detail section of the fault, you can do so by building further application-specific COBOL bindings using DFHSC2LS and issuing further TRANSFORM commands in the application.

Note: It is valid to combine steps 1 and 2 into a single TRANSFORM command. Similarly steps 5 and 6 can be combined.

Optional: Editing the Schema

Advanced users could consider editing the XML schema in such a way as to simplify the application code that is required. The XML schema currently describes a SOAP Body tag as in Example 10-14.

Example 10-14 current schema of SOAP Body

```
<xs:complexType name="Body" >
<xs:sequence>
<xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded"
processContents="lax" />
</xs:sequence>
...
</xs:complexType>
```

You could change this so that it explicitly claims to hold a single SOAP Fault as in Example 10-15 on page 227.

```
<xs:complexType name="Body" >
<xs:sequence>
<xs:element ref="tns:Fault"/>
</xs:sequence>
...
</xs:complexType>
```

If you process the edited XML schema with DFHSC2LS, you will get a simpler set of language structures created with less container-based indirection. The application code will therefore be simpler and could be written with a single call to the TRANSFORM command.

10.3 Handling variably recurring XML elements

WSDL enables a Web service to define any xsd:element as being optional. It also allows for elements to appear multiple times. Elements that might appear an undefined number of times are known as variably recurring elements. Because the number of occurrences of the data that will be present in a SOAP message is not known until the request is received by CICS, writing an application program to access all occurrences of the element can be difficult.

There are two major mechanisms for handling variably recurring data top-down. The first is to use a technique that became available at mapping level 2.1 called *in-lining*. The second is to use a mechanism based around CICS containers.

10.3.1 In-lined variably recurring data

In this scenario DFHWS2LS maps the variably recurring data into a simple array together with a num field to indicate the number of instances of the data that are actually present.

The application program can then access the instance of the data as they would a normal array, with the restriction that the num field must be used appropriately.

To use in-lining you have to set a value for the INLINE-MAXOCCURS-LIMIT parameter of DFHWS2LS. This parameter was introduced at mapping level 2.1 and it indicates which values of maxOccurs to in-line. It defaults to a value of 1, indicating that optional fields should be in-lined, but nothing else. You can increase the value to in-line a greater number of variably recurring elements.

Figure 10-1 shows a fragment of a WSDL document viewed in RDz. It includes an xsd:element called recs that can be used anywhere from one to 10 times. The data therefore can consist of at least one instance and it can have 10 instances at most.

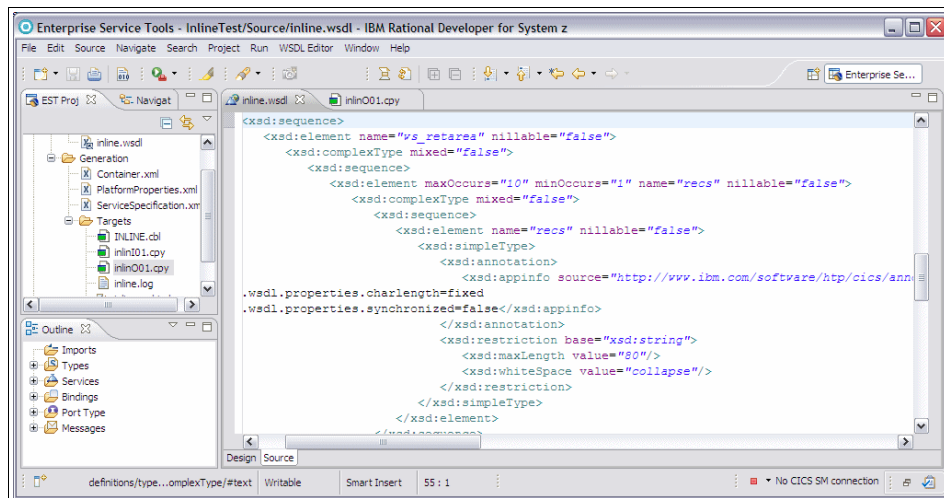


Figure 10-1 In-line mapping example - WSDL

If DFHWS2LS is used with an INLINE-MAXOCCURS-LIMIT value set to at least 10, the following COBOL language structure is generated, as shown in Figure 10-2.

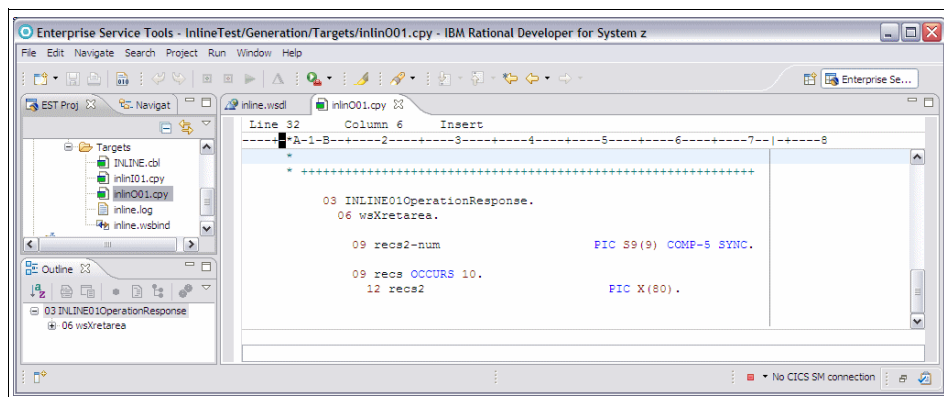


Figure 10-2 In-line mapping example - generated Cobol structure

You can see that an array has been allocated sufficient to hold 10 instances of the data. In this example, each instance of the data is only 80 bytes long. But in real WSDL individual data instances might be much longer, in which case you

should think carefully about whether in-lining the data is a good idea. If, for example, each instance of the data is a one MB in size, in-lining up to 10 instances will result in a lot of storage being allocated, even if only a single instance is normally used.

Furthermore, if the WSDL specified `maxOccurs="unbounded"` then in-lining the data is not an option. In this case, the container based strategy is used.

10.3.2 Container based variably recurring data: inbound

The fragment of WSDL in Example 10-16 defines an element of type `s:string` called `name` that optionally might appear in the SOAP message.

Example 10-16 Defining an optional element

```
<s:element name="name" type="s:string" minOccurs="0" maxOccurs="1">
```

When DFHWS2LS parses this element declaration without in-lining being active, it places two fields in the generated language structure. Example 10-17 shows the output for the C / C++ language.

Example 10-17 Output from DFHWS2LS for variable cardinality element

```
int name_num;  
char name_cont[16];
```

The first variable references how many occurrences of the element `name` were found in the input message. The second variable gives the name of the container where the occurrences of the element were stored.

Where we are just using an optional element, we can use the numeric variable to test whether the optional element was included in the input message. If the variable is set to 1, the optional element was sent and the data value was placed in a container. The second variable holds the name of this container.

Example 10-18 declares an instance of the structure generated by DFHWS2LS, then uses that to hold a local copy of the data in the container.

Example 10-18 Accessing an optional element

```
struct Name  
{  
    char name[255];  
}  
  
struct Name myName;
```

```

if(name_num == 1)
{
    EXEC CICS GET CONTAINER(name_cont)
    NODATA
    FLENGTH(length);

    EXEC CICS GET CONTAINER(name_cont)
    INTO(&name)
    FLENGTH(length);
}

sprintf(message, "value of optional element name was
%.255s",myName.name);

```

The maxOccurs and minOccurs attributes can also be used to define an array of elements that can have a maximum number of elements. The WSDL extract in Example 10-19 redefines the name element to be an unbounded array.

Example 10-19 Defining a variable cardinality element

```

<s:element name="name" type="s:string" minOccurs="0"
maxOccurs="unbounded">

```

At run time, CICS will take all instances of the element name that were sent on the request, concatenate them, and place the concatenation into a single container. The Web service then has to navigate this structure. One method of doing this is to use pointer arithmetic to access all data items in the container.

In Example 10-20, instead of declaring a single instance of the structure we declare a pointer of the type structure and set it to the address of the container that is used to store the concatenation. This enables us to navigate through the whole structure easily.

Example 10-20 Processing an optional element in C

```

struct Name
{
    char name[255];
}

struct Name*names;
int counter = 0;
char current_name[255];

if(name_num > 0)

```



```

{
    /*If a collection of names was sent process them*/
    EXEC CICS GET CONTAINER(name_cont)
    NODATA
    FLENGTH(length);

    EXEC CICS GET CONTAINER(name_cont)
    SET(names)
    FLENGTH(length);

    /*For each name sent print it out*/
    for(counter=0;counter<name_num;counter++)
    {
        memcpy(current_name,names(counter).name,255);
        sprintf(message, "value of name %.255s",names(counter).name);
    }
}

```

Example 10-21 shows a similar technique in COBOL.

Example 10-21 Processing an optional element in COBOL

```

WORKING-STORAGE SECTION.
01 W-S-VARIABLES.
03 NAME-PTR                USAGE IS POINTER.
03 X-PTR                   USAGE IS POINTER.
03 IX                      PIC S9(8) COMP-4 VALUE 1.
03 ITEM-COUNT              PIC S9(9) COMP-4 VALUE 0.
*
LINKAGE SECTION.
01 X                       PIC X(659999).
*
01 NAME.
05 productName             PIC X(255).
*
EXEC CICS GET CONTAINER(NAME-cont OF WS-STARTI)
SET(NAME-PTR)
FLENGTH(NAME-FLENGTH)
RESP(RESP)
RESP2(RESP2)
END-EXEC.
*
* Get addressability to NAME-cont
SET ADDRESS OF X TO NAME-PTR
SET ADDRESS OF NAME TO ADDRESS OF X

```

```

*
* Work through NAME-cont processing the data fields within
* each NAME record
*
  PERFORM WITH TEST AFTER
  UNTIL NAME-COUNT = NAME-num OF WS-STARTI
**
* Display productName field
  DISPLAY 'productname is now: ' productName
*
* Move to the next NAME record
  ADD LENGTH OF NAME TO IX
  SET NAME-PTR TO ADDRESS OF X(IX:1)
  SET ADDRESS OF NAME TO NAME-PTR
  ADD 1 TO NAME-COUNT
END-PERFORM.

```

10.3.3 Container based variably recurring data: outbound

When creating a Web service that will use an outbound list of elements or optional elements, all instances of the element must be concatenated and placed into a new container on the current channel. The container used to hold the concatenation must have a unique name on the channel. The name can be any 16-character string. However, it must not start with “DFH” as names beginning with these characters are reserved for CICS. After the container has been populated, the `element_cont` / `element_num` fields declared in the language structure created by the Web Services Assistant must be populated to allow CICS to parse the container data into a SOAP message. Example 10-22 shows a method that generates five instances of the name element and places them in a container.

Example 10-22 Global function to generate unique container names

```

char* generate_names()
{
    struct Name *names;
    int counter = 0;
    char name_container[16];

    /*Generate 5 name structures and place them in a container*/
    /*Allocate storage for 5 concatenated Name structures*/
    names = (struct Name*) calloc(5,sizeof(struct Name));
    /*Populate the names with data*/
    strcpy(names[0].name,"Test_User");

```

```

strcpy(names[1].name,"Another_User");
/*...etc...*/

/*Using a global function to generate a unique container name*/
memcpy(name_container,get_container_name(),16);

/*Add the concatenation to a container*/
EXEC CICS PUT CONTAINER(name_container)
FROM(names)
LENGTH(sizeof(struct Name) * 5);

/*populate the name_cont and name_num variables*/
memcpy(name_cont,name_container,16);
name_num = 5;
}

```

10.4 Handling undefined XML (xsd:any)

Some WSDL documents allow sections of arbitrary well-formed XML to be included within the application data. For example, you could embed an XHTML document within the body of the SOAP message. Where this technique is used the WSDL will use either an `xsd:any` tag, or an `xsd:anyType` data type.

Prior to mapping level 2.1, DFHWS2LS did not support these constructs. At mapping level 2.1 they are supported using a pass-through technique that allows the application to handle that subset of the SOAP directly as XML.

For example, consider the fragment of WSDL viewed in RDz shown in Figure 10-3 on page 234. It specifies an optional undefined XML tag might appear on the end of an `xsd:sequence`. This is a technique that can be used to support future evolution of the WSDL. If version 2 can add something specific to the end of the list, the resultant SOAP message will still validate with respect to the original WSDL.

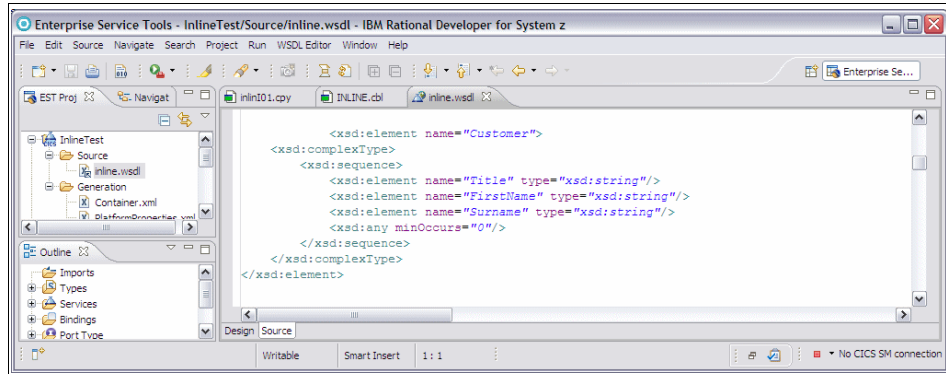


Figure 10-3 A fragment of WSDL with an optional `xsd:any`

The language structures generated by DFHWS2LS are as shown in Figure 10-4.

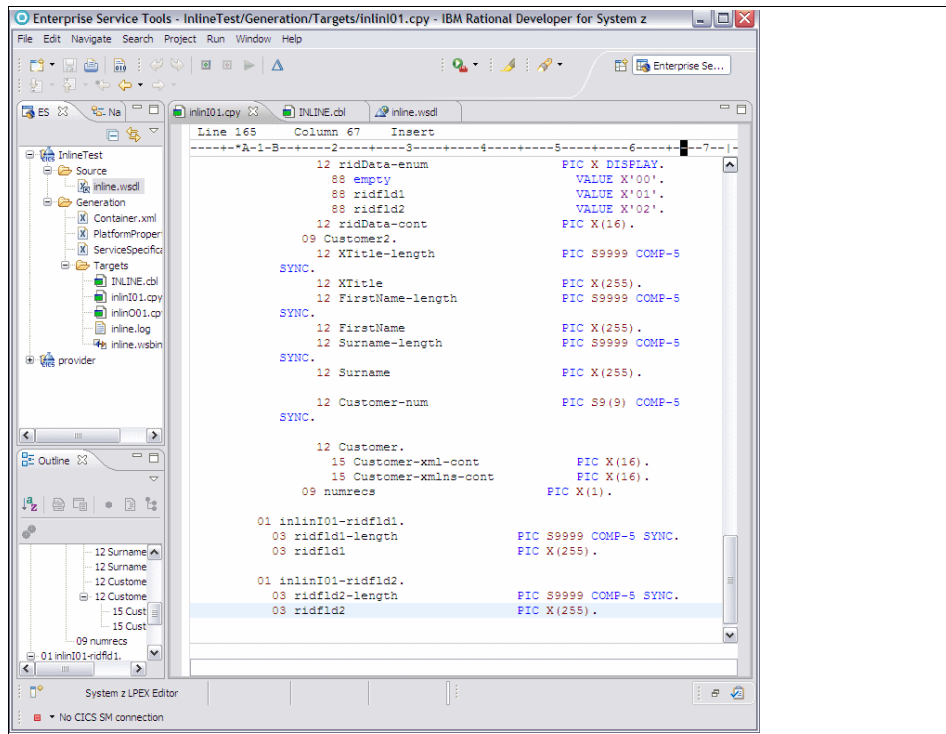


Figure 10-4 Generated language structures for `xsd:any`

In this example two significant fields have been generated:

- ▶ Customer-xml-cont PIC X(16)
This field indicates the name of the container in which the associated XML can be found.
- ▶ Customer-xmlns-cont PIC X(16).
This field indicates the name of a container in which any in-scope XML namespace prefix declarations can be found. If the XML in the first container is not self contained, you might need namespace prefixes from the second container to understand the XML.

An application that wants to understand the contents of these containers might do so with the EXEC CICS TRANSFORM command in CICS TS V4.1. It provides a mechanism that is suited to parsing or generating the XML in these containers.

10.5 Handling enumerated XML constructs

Certain constructs in the XML schema definition language are interpreted by DFHWS2LS at mapping level 2.2 and above as having enumerated content models. This means that there are a fixed number of possible options, only one of which can actually be used. For example, the `xsd:choice` construct indicates a set of options, but only one of the options can be used.

When DFHWS2LS parses `xsd:choice` constructs at mapping level 2.2 or above, it places two fields in the generated language structure. Example 10-23 shows the output for the C / C++ language.

Example 10-23 Output from DFHWS2LS for `xsd:choice` at mapping level 2.2

```
char name_enum;  
char name_cont[16];
```

The first variable indicates which of the possible options is used, the second variable gives the name of the container where the application data associated with that option can be found.

Figure 10-5 shows how we use DFHWS2LS at mapping level 2.2 to generate a COBOL language structure from an XML `<xsd:choice>` construct with two options.

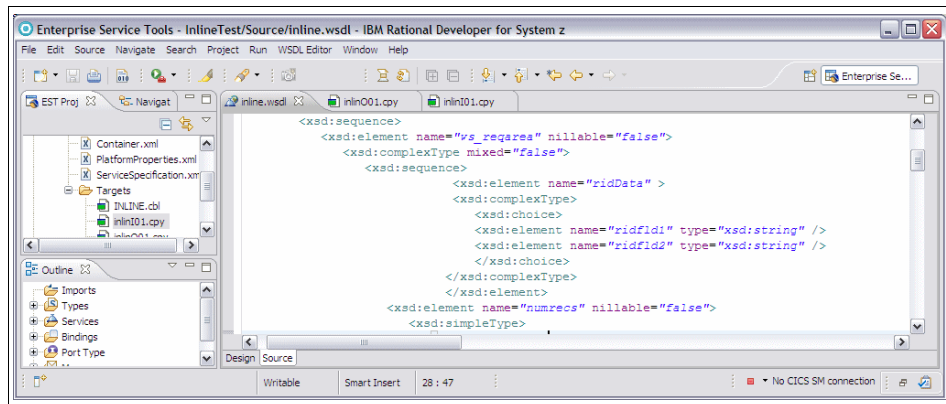


Figure 10-5 WSDL - `<xsd:choice>` element

The `ridData-enum` field indicates which option from a set of possible values is being used. The associated value is stored in the container referenced in `ridData-cont`. A value of `X'00'` indicates no content. A value of `X'01'` indicates an instance of structure `inlinI01-ridfld1`. A value of `X'02'` indicates an instance of structure `inlinI01-ridfld2`. See Figure 10-6.

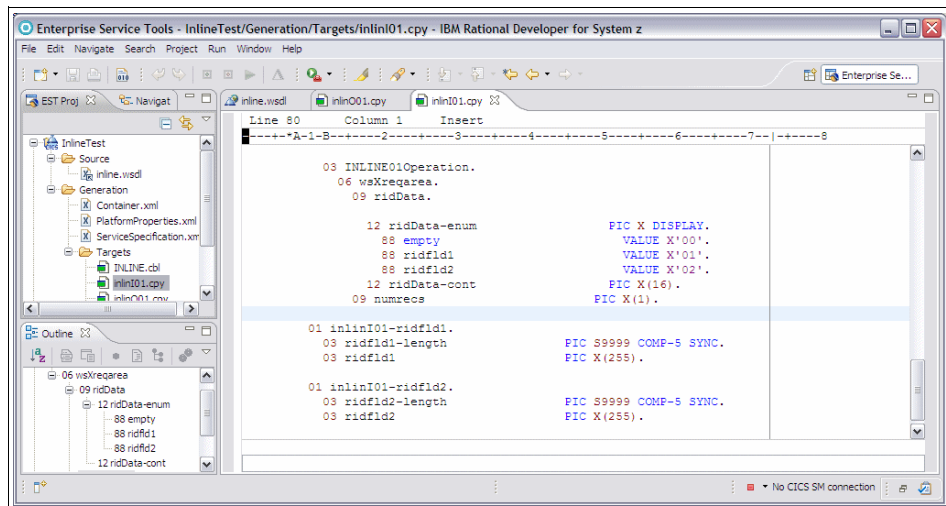


Figure 10-6 Generated COBOL language structures for `xsd:choice`

Other constructs from XML that are handled in a similar way. This includes:

- ▶ Substitution groups

This is an advanced concept that allows an `xsd:element` to be substituted with any other `xsd:element` from a specific set of options.

- ▶ Abstract data types

This is an object orientated concept where the schema references an abstract parent data type, but where one of a set of child data types will actually be used in the SOAP messages.

In all of these scenarios, DFHWS2LS generates language structures to map the individual options, and an enum field to indicate which option is actually used.

10.6 Modifying generated WSDL

If you are unsatisfied with the WSDL bindings for an application that has been processed using DFHLS2WS, a useful technique is to edit the generated WSDL until it matches your expectations. You can then reprocess that WSDL using DFHWS2LS, and potentially write a wrapper program to map data between the new generated language structures, and the original ones.

In the following example we use this technique to demonstrate how individual fields within a language structure can be mapped as `xsd:base64Binary` data rather than `xsd:string` data. This makes those fields eligible for optimization using the MTOM/XOP protocol in CICS TS V3.2.

10.6.1 Background to MTOM/XOP

If MTOM is enabled, some SOAP messages that contain binary data might be processed faster in the PIPELINE and on the network than would otherwise be the case. In standard SOAP messages, any binary data that is sent (such as an image file) is encoded using a representation called base64 encoding. This representation increases the size of the binary data and can impact transmission time.

Enabling MTOM/XOP in the pipeline reduces the size of SOAP messages that contain base64 encoded data. The SOAP Message Transmission Optimization Mechanism (MTOM) and XML-binary Optimized Packaging (XOP) specifications, (often referred to as MTOM/XOP) define a method for optimizing the transmission of large `xsd:base64Binary` data objects within SOAP messages. The MTOM specification conceptually defines a method for optimizing SOAP messages by separating out binary data that would otherwise be base64

encoded, and sending it in separate binary attachments using a MIME Multipart/Related message. This type of MIME message is called an MTOM message.

Sending the data in binary format reduces its size, optimizing the transmission of the SOAP message. The XOP specification defines an implementation for optimizing XML messages using binary attachments in a packaging format that includes, but is not limited to, MIME messages. The size of the base64binary data is reduced because the attachments are encoded in binary format. The XML in the SOAP message is converted to XOP format by replacing the base64binary data with a special <xop:Include> element that references the relevant MIME attachment using a URI.

Measurements show that sending large binary fields as MTOM/XOP attachments offers significant performance improvements in CICS compared to using ordinary xsd:base64Binary data. The size of the XML part of the data is smaller, so there is a lot less data for CICS to parse through searching for XML markup.

However, use of MTOM/XOP does require that the partner process must also understand this protocol. There are some scenarios where enabling MTOM/XOP is not advisable. Refer to the CICS Information Center for further details.

10.6.2 Support for xsd:base64Binary and MTOM/XOP

If your WSDL documents contain fields defined with type xsd:base64Binary, and if you use DFHWS2LS at mapping level 1.2 or higher, then you are eligible for the MTOM/XOP optimisations.

If you are using DFHLS2WS and want to treat all of the text fields as binary data (and thereby make them eligible for MTOM/XOP optimisation), do so by specifying CHAR-VARYING=BINARY as a parameter in DFHLS2WS. However, if you are using DFHLS2WS and only want to treat a single field as having binary content then you will have to use the following more complicated technique.

10.6.3 Mapping a single field as binary data with DFHLS2WS

If you have an application that you want to enable as a Web service and use a binary mapping for a single field, you should perform the following steps:

1. Run DFHLS2WS to generate the WSDL as normal;
2. Modify the WSDL so that the field in question specifies data type xsd:base64Binary;
3. Run DFHWS2LS on the generated WSDL to generate a WSBind file and language structures;

4. Review the generated language structures to ensure that they are compatible with the original language structures. In this scenario it is likely that they will be. In which case no further action is required.
5. If the new language structures are not compatible with the original language structures, either modify the existing program, or implement a wrapper program that maps between the new and old data formats.

The scenario in Example 10-24 demonstrates how to generate a WSBind file that can be used to interpret base64Binary data. We use the Cobol data structure as shown in the following example to generate a WSDL using DFHLS2WS. We intend to move a maximum of 60000 bytes of binary data to the imgData field.

Example 10-24 COBOL structure

```
01 ws-data.  
  03 cafld1          PIC X(15).  
  03 cafld2          PIC X(15).  
  03 cafld3          PIC X(6).  
  03 imglength      PIC X(8).  
  03 imgData        PIC X(60000).
```

After running DFHLS2WS, the generated WSDL contains the imgData element which is defined as an element of type xsd:string. See Example 10-25.

Example 10-25 Binary field imgData after running DFHLS2WS

```
<xsd:element name="imgData" nillable="false">  
  <xsd:simpleType>  
    <xsd:annotation>  
      ::::  
      ::::  
    </xsd:annotation>  
    <xsd:restriction base="xsd:string">  
      <xsd:maxLength value="60000"/>  
      <xsd:whiteSpace value="collapse"/>  
    </xsd:restriction>  
  </xsd:simpleType>  
</xsd:element>
```

To generate a WSBind file that can interpret the element as base64Binary, We modified the WSDL as shown in Example 10-26.

Example 10-26 Modified binary field imgData

```
<xsd:element name="imgData" nillable="false">
  <xsd:simpleType>
    <xsd:annotation>
      ::::
      ::::
    </xsd:annotation>
    <xsd:restriction base="xsd:base64Binary">
      <xsd:maxLength value="60000"/>
      <xsd:whiteSpace value="collapse"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

We then reprocess the WSDL using DFHWS2LS.

10.6.4 Handling variable length values and white space

A common requirement is to process variable length values as part of your SOAP messages. For example, if you have a field whose content might vary in size from zero bytes to 1 Mb, you will not want an every message to be padded to the maximum length with spaces.

There are several characteristics of WSDL that need to be considered when discussing this problem.

maxLength, minLength, and length facets

An XML Schema facet identifies a specific type of restriction to apply to a data type. The technical definition of these facets can be read in the XML Schema specification at the following Web page:

<http://www.w3.org/TR/xmlschema-2/#rf-facets>

The facets are used to restrict the range of values for a data type. If the length facet is specified for a data type, that data type is of fixed length. If the maxLength facet is supplied, there is a maximum length for the data type. Similarly, if a minLength is specified, there is a minimum length for the data type.

If none of these facets are set then a String based data type is considered to have an unbounded maximum length and a minimum length of 0. See Example 10-27 on page 241.

Example 10-27 A string with a maximum length specified

```
<xsd:element name="thing">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="3000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

In Example 10-27 an `xsd:element` called `thing` has been defined with a data type derived from `xsd:string`. A maximum length of 3000 characters has been specified.

whiteSpace facet

The `whiteSpace` facet is used to define the desired behavior with respect to white space around a data value. There are three possible values for this facet, `preserve`, `replace`, and `collapse`. The definition for this facet is available at the following Web page:

<http://www.w3.org/TR/xmlschema-2/#rf-whiteSpace>

If a value of `preserve` is used, any spaces, tabs, new lines, and so forth, within the value are considered to be deliberate. If a value of `replace` is used, tabs and new lines are replaced with an appropriate number of spaces. If a value of `collapse` is used, leading and trailing white space is removed.

A value of `preserve` is implied if the WSDL does not specify a value for this facet. See Example 10-28.

Example 10-28 A string with 'collapse' processing for white space

```
<xsd:element name="thing2">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="3000"/>
      <xsd:whiteSpace value="collapse"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

In Example 10-28, an `xsd:element` called `thing2` has been defined that has the same definition as the `thing` in Example 10-27, but the WSDL author has specified that white space within the XML tag at runtime should be collapsed.

Null characters (x'00') are not valid in XML

XML documents are not allowed to contain null values. This is a general requirement in all XML documents including SOAP messages. If at runtime CICS is asked to include a text value within a generated SOAP message that includes a null character, CICS will treat that character as the end of the string and the value will be truncated.

This is true for all values of the whiteSpace, length, minLength, and maxLength facets. Care must be taken if a variable length mapping strategy is used and the text strings might contain null characters.

How DFHWS2LS handles variable length values

The behavior of DFHWS2LS with respect to variable length values can be changed using input parameters at mapping level 1.2 or higher.

Three issues that must be considered are the default maximum string length imposed by DFHWS2LS if such a length is not specified in the WSDL, the mapping of variable length values into CICS containers, and the mapping of variable length values into fixed length character arrays.

- ▶ **The default maximum string length**

DFHWS2LS requires that all xsd:string defined in the WSDL have a maximum length. If the WSDL does not specify a maximum length, DFHWS2LS imposes one. By default this maximum length is set to 255 characters.

The default maximum length imposed by DFHWS2LS is configurable using the DEFAULT-CHAR-MAXLENGTH input parameter. If you would prefer a default maximum length of 20 characters then you could specify DEFAULT-CHAR-MAXLENGTH=20. If you would like a default maximum length of 2K then you could set DEFAULT-CHAR-MAXLENGTH=2048.

In general it is best to specify in the WSDL the precise maximum character length you would like DFHWS2LS to use as this avoids the problem of having one global default being applied to all xsd:string based data types. It also avoids the runtime problem of a partner process sending a data value to CICS which is longer than the maximum data length imposed by DFHWS2LS.

You can specify maxLength="unbounded" in the WSDL to indicate that there really is no theoretical maximum length to the String.

- ▶ Mapping variable length values into a CICS container

You can tell CICS to use a container for storing variable length xsd:string values. CICS does this automatically for xsd:strings which are known to have a maximum length greater than 32K characters (at mapping level 1.1 and above). CICS containers are a convenient way to address long variable length values.

You can specify the threshold at which this container based mapping is used by setting a value for the CHAR-VARYING-LIMIT parameter. For example, if you want all variable length xsd:strings with maximum lengths of 255 or greater to be mapped into CICS containers then you would do so by specifying CHAR-VARYING-LIMIT=255. You can, for example, combine this parameter with the DEFAULT-CHAR-MAXLENGTH parameter to specify that all xsd:strings with an unspecified maximum length are mapped into CICS containers.

If the container mapping is used then a language structure is created by DFHWS2LS with a field for the container name to be stored.

The container used must always be read from and written to in BIT mode.

- ▶ Fixed Length mappings for variable length Strings

There are three different mechanisms available that DFHWS2LS can use for mapping variable length values to fixed length character arrays. These mechanisms are most appropriate where the maximum length of the xsd:string is known to be relatively short, therefore causing low overhead in terms of wasted space in storage.

These mechanisms are: basic fixed length character arrays; null terminated character arrays and 'varying' character arrays. You can select which one is used by setting a value for the CHAR-VARYING parameter.

- Basic fixed length character arrays

These are often the default at lower mapping levels. DFHWS2LS allocates a field within the language structure based on the maximum length of the xsd:string (as defaulted using the DEFAULT-CHAR-MAXLENGTH parameter). At runtime CICS will pad the value that arrives on the wire with spaces to fill this field. If the value that arrives on the wire is too large for the field then a conversion error is reported.

For outbound communication the application should place a value into the character array and either null terminate the value or pad it with spaces. If the value of the whiteSpace facet was 'collapse' then CICS will remove any trailing white space. If the field was null terminated then CICS will truncate the value at the first null.

You can specifically request this variable length mapping strategy by specifying the CHAR-VARYING=NO input parameter.

- Null terminated character arrays

In this scenario DFHWS2LS behaves in a similar way as for fixed length character arrays, but CICS will add a null terminator to the end of the data in any character array it populates. The application program can therefore recognize the end of the significant data. For example, the application can detect white space that is deliberately present due to the use of the `whiteSpace="preserve"` facet.

For outbound communication the application must null terminate any character arrays it populates.

You can specifically request this variable length mapping strategy by specifying the `CHAR-VARYING=NULL` input parameter.

- Varying character arrays

DFHWS2LS can produce character arrays that are prefixed with an explicit length field that is used to identify the significant characters from the fixed length buffer. This format of representation is particularly common in PL/I.

You can specifically request this variable length mapping strategy by specifying the `CHAR-VARYING=YES` input parameter.

If this mapping strategy is used then CICS will generate SOAP messages that contain the requested number of characters for the field. This mapping strategy usually results in the best performance as there is no need to scan the text fields to identify the significant characters.

The choice of which variable length mapping strategies to use is mostly a matter of application development strategy. In general it is a good idea to specify both the `whiteSpace` and `maxLength` facets in your WSDL documents for each `xsd:element` as the defaults might not be appropriate, and to set a value for the `CHAR-VARYING-LIMIT` input parameter.

How DFHLS2WS handles variable length values

DFHLS2WS has fewer options for processing variable length values. The most common technique in COBOL for defining variable length values is to use the `OCCURS DEPENDING ON` data type, but this is not supported by DFHLS2WS. By default CICS treats all character arrays as fixed length. As with the equivalent DFHWS2LS scenario, if the application null terminates any character arrays used, CICS will truncate values at the null character rather than including any padding characters in the outbound SOAP messages.

You can specify `CHAR-VARYING=NULL` under DFHLS2WS so that CICS will always treat character arrays as being null terminated. If you use this option, the maximum length of the field is effectively one character less than specified as the null terminator takes up one character.

There is a further options available at mapping level 2.1. You can specify a value of CHAR-VARYING=COLLAPSE. This tells CICS to remove automatically any trailing spaces from the end of character arrays when generating XML. This is the default value of the CHAR-VARYING option at mapping level 2.1 for all programming languages other than C and C++.

10.7 WSDL types not supported by DFHWS2LS

Although the Web Services Assistant will accept most WSDL documents, some elements of WSDL are not accepted or are ignored. This section will introduce several techniques that can be used if DFHWS2LS rejects your WSDL.

Start by validating the WSDL. Sometimes DFHWS2LS rejects a document because there is something subtly wrong with it. RDz and Eclipse are both good at doing WSDL validation.

Make sure that the most recent mapping level is being used. On CICS TS V3.1 this means mapping level 1.2. On CICS TS V3.2 this means mapping level 2.2. For CICS TS V4.1 this means mapping level 3.0. Many elements that are not supported in CICS TS V3.1 are supported in CICS TS V3.2 or CICS TS V4.1, so in some cases, an upgrade to a new version of CICS might be advisable.

Look at modifying a local copy of the WSDL to work around problematic constructs. In CICS TS V3.2 and above a good technique is to replace unsupported constructs with `xsd:any` or `xsd:anyType` fields. That passes the problem of parsing and generating the problematic XML to the application, but at least the application only has to parse the subset of the XML that CICS does not support. Example 10-29 show an element which is not supported.

Example 10-29 minOccurs and maxOccurs in <xsd:sequence> element

```
<xsd:element name="testElement1">
<xsd:complexType>
  <xsd:sequence minOccurs="2" maxOccurs="5">
    <xsd:element name="shipTo" type="xsd:string"/>
    <xsd:element name="billTo" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

The use of minOccurs and maxOccurs attributes are not supported for the <xsd:sequence> element. The exceptions to this rule are when minOccurs="0" and maxOccurs="1" or minOccurs="1" and maxOccurs="1". In Example 10-30, we can use <xsd:anytype> to replace the unsupported constructs.

Example 10-30 Valid WSDL

```
<xsd:element name="testElement1" type="xsd:anyType" />
</xsd:element>
```

Another approach is rewriting the WSDL using supported elements. Example 10-31 show a nested <xsd:choice> that is not supported by DFHWS2LS.

Example 10-31 nested choice

```
<xsd:choice>
  <xsd:element name="name1" type="string" />
  <xsd:choice>
    <xsd:element name="name2a" type="string" />
    <xsd:element name="name2b" type="string" />
  </xsd:choice>
</xsd:choice>
```

We can change the WSDL as shown in Example 10-32. These two WSDL fragments are equivalent.

Example 10-32 valid WSDL

```
<xsd:choice>
  <xsd:element name="name1" type="string" />
  <xsd:element name="name2a" type="string" />
  <xsd:element name="name2b" type="string" />
</xsd:choice>
```

You can also use this approach to add restrictions that would not otherwise have been present such as setting the maxLength for xsd:strings to something sensible.

If modifying the WSDL is not acceptable or is not possible, you should consider writing applications that work directly with the XML. For example, you can create your own XML-aware Web service applications.

For the detailed information about how to do this, refer to the following Web page:

https://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.webservices.doc/tasks/dfhws_creating_xmlapps.html

An important variant of this concept is to use Java in CICS to handle the XML. This idea is discussed on the following Web page:

<https://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp?topic=/com.ibm.cics.ts.webservices.doc/concepts/javacics.html>

If these are not acceptable, it might be necessary to host a transformative technology off-platform that can map between the original WSDL and something CICS can support. This could involve having a simpler Web service hosted in CICS, and a mapping technology in one of the broker products such as WESB, WMB, or DataPower.

Note: In scenarios where CICS is a requester and the response comes back with large volumes of superfluous data that would otherwise be ignored in CICS, this is a good technique for stripping the bloat from the XML before passing the useful data on to CICS.

10.8 Problem determination

While preparing material for this publication, we discovered user faults that caused errors. In this section we outline problems that you might discover and the relevant solutions.

10.8.1 Problems using DFHWS2LS and DFHLS2WS

When using the Web Services Assistants DFHWS2LS and DFHLS2WS, several errors might occur. This section includes some of the most common errors and solutions to fix the problem.

If OMVSEX fails with a return code of 127 and the following error is seen:

FSUM7351 not found

The IBM LookAt tool can be used to get further diagnostic information from this error message. The LookAt tool can be found at the following Web page:

<http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/>

In this case the definition in Example 10-33 is given in response to the error code.

Example 10-33 Response from LookAt tool

Explanation: You attempted to execute a command that could not be found.

User Response: Ensure that the command exists and that the PATH environment variable is valid.

This error states that the PATH environment variable is invalid. Because the Web Services Assistant is a Java program, more paths have to be correct for the program to run.

In some situations, the line numbers along the right column of the JCL job card can be passed to the Java program, which will attempt to use them as part of the input parameters. This underlying fault can result in different error codes. Sometimes JES will truncate the characters after column 72. In this situation the job will run and only result in a return code of 4 in the OMVSEX step. Another result that stems from the same problem is if the Java program throws a `java.lang.IllegalArgumentException` when attempting to parse a parameter that has line numbers attached to the end of the parameter. This problem can be resolved by setting `NUMBERS OFF` in your JCL profile.

The user ID under which the Web Services Assistant job runs must be defined to OMVS and have read access to HFS and write access to the directory specified on the LOGFILE parm. Because the assistants are Java programs, the user ID must have a large enough storage allocation to run Java.

Neither of the Web Services Assistant programs lock the temporary HFS files that they create. Therefore a batch of these jobs cannot be run in parallel and must be run sequentially. Failure to do this can cause undocumented errors.

10.8.2 Using the execution diagnostic facility to debug Web services

CEDX can be used to debug a request as it is processed by CICS. To turn on the execution diagnostic facility to debug Web services, use the `CEDX CPIH,ON` command.

The CPIH transaction is the CICS inbound HTTP inbound routing transaction, so it is run when CICS is servicing an inbound Web service over the HTTP transport. After the command above has been issued, any inbound Web service request can be debugged. CEDX will debug all EXEC CICS commands in any custom message handlers that are executed and the terminal application

program. Although only EXEC CICS commands are shown during the debug session, this is useful to see the flow of execution through message handlers.

When you have finished debugging your Web service, turn off CEDX using CEDX CPIH,OFF.

10.8.3 Debugging CICS SFR applications

When using Service Flow Modeler-deployed flows in CICS Integrator Adapter runtime, the following debugging points might help provide diagnostic information.

The CICS SFR properties file contains information relating to each adapter deployed within the CICS SFR environment. This file is updated each time a new adapter is deployed through a JCL job run by JES that executes DFHMAMUP, which adds a record to the properties file (DFHMAMPF). The contents of this file can be used to ensure that the parameters used to run the deployed service are correct. JCL job DFHMAMPD dumps the contents of the properties file. Each record in the file corresponds to a particular service adapter that has been deployed within the CICS SFR environment. Figure 10-7 shows the output from job DFHMAMPD.

```
Property type: R (Request properties)      Name: MAIVPREQ      Version: 2
Request type: 1 (Sync)                    Nav/Init name: DFHMAIP1  Nav/Init transid: CMA5  Type: 0 (Navigator)
Persistence: 1 (Yes)                      XML parse name:        Deployment: 2 (COMPLEX)

Property type: R (Request properties)      Name: NCDPLAA      Version: 2
Request type: 0 (Async)                   Nav/Init name: NCDPLAN  Nav/Init transid: NDAN  Type: 0 (Navigator)
Persistence: 1 (Yes)                      XML parse name:        Deployment: 2 (COMPLEX)
```

Figure 10-7 Sample CICS SFR property file dump

The CICS Integrator Adapter Error Listener load module (DFHMAERQ) is triggered whenever an error message is written to intrapartition transient data queue CMAQ and writes to the CICS SFR error file (DFHMAERF). This file contains information about the cause of the error and the specific error code. An error is written only if CICS SFR has encountered an error running the deployed adapter. An error will not be written if the deployed service ran correctly but the data was unexpected. The contents of the error file can be dumped by using sample JCL job DFHMAMED, which runs module DFHMAEUP. To ensure that all data has been flushed to the error file before running the dump JCL, it is recommended that you close and re-open the error file definition inside CICS. Figure 10-8 on page 250 shows a dump from the CICS SFR error file.

```

Processed: Date: 07/07/05   Time: 11:14:54:   PutApplid:           PutTranid:
Error: CIA08002E   Normal processing

  Userid: CICSUSER   Applid: IYK3ZMY1   Tranid: CKBP   Eibtaskn: 0000271   AbsTime: 003329723694270
Request: MAIVPREQ   Mode: Sync   Program: DFHMADPL   Type: System
Activity:           Node Name:
Event:           Event type: None   Step: MAIN
Proctype: DFHMAINA   Process: CICSUSER0000271003329723694270
Failed Processtype:   Failed Process:
ReplyToQ:           ReplyToQMgr:
MQ MsgId:           MQ CorrelId:

Error detail: Application

```

Figure 10-8 Sample CICS SFR error file dump

The error field is a key area of interest in this dump file. This field defines the CICS SFR error code that has caused the deployed flow to fail. The full list of CICS SFR error codes can be found in the *CICS Integrator Adapter for z/OS Run Time User's Guide and Reference*, SC34-5899-05. The program field shows which program CICS SFR was executing when the failure happened.

Some of the most common error messages are:

► CIA01001E

This designates a VSAM file read error. The most common reason for this is that DFHMADPL has attempted to read the properties file (DFHMAMPF) for a misspelled request name. This can be verified by checking that the program field in the error dump reads DFHMADPL and the file being read is DFHMADPL. The error dump also shows the value used in the file read operation.

► CIA03001E

This error message means that an EXEC CICS LINK to the resource being modelled in the deployed flow has failed. A common reason is an incorrect SYSID value in the property file. Check the TYPE=2 PARM02 value in the property file to ensure that it is set to the system ID that is hosting the DPL application.

Because CICS SFR uses Business Transaction Services to process requests, the BTS audit level feature can be used to provide further diagnostic information. Each CICS SFR request runs under a BTS process type. This is defined at runtime in the CICS SFR DFHMAH header field DFHMAH-PROCESSTYPE. The process type defined in the header also must have been defined to CICS. This can be done using CEDA DEFINE PROCESSTYPE.

Within the CICS definition an audit level variable is set. Possible values are:

- ▶ Activity
- ▶ Full
- ▶ Process
- ▶ None

If an audit level other than none is specified, audit log records are written to an MVS logstream by the CICS Log Manager. You can read the records offline using the CICS audit trail utility program (DFHATUP). A sample job, DFHMABAP, is provided to run this program.

You might want to consider defining a specific process type for debug use and another for use within an production environment.

Another way to debug a deployed flow is to check the use counts of all generated programs before and after flow execution. This will enable you to track the execution of the flow and establish where in the flow the error is happening. When the failing module is known, it can be useful to run that application using the debug facility CEDF for further diagnostic information.

10.8.4 Runtime SOAP validation

Each Web service that is deployed onto CICS through the Web Services Assistant performs simple validation of each SOAP request before it is parsed and the data transformed into the language structure. This validation checks that the SOAP request is a well-formed XML document. Any SOAP request that fails this validation will be refused by CICS. It is possible for CICS to validate each SOAP request against the WSDL schema. Because the WSDL schema defines the syntax of a valid SOAP request, such validation ensures that the request contains all of the necessary elements. Such validation does incur a significant overhead to the processing of a Web service request and is not recommended to be used in a production environment. However, when used in a testing environment, runtime validation can be useful to ensure that SOAP requests reaching your Web service are correct.

To turn validation on, use the CEMT S WEBSERVICE(WEBSERVICENAME) VALIDATION command.

(*WEBSERVICENAME* is the name of the Web service definition you want to debug.)

Also, ensure that the WSDLFILE attribute of the Web service definition is set to the path for the WSDL file that you want SOAP validation to be performed against.

Tip: If you need to turn validation on a Web service that has a mixed case name, be sure to activate mixed-case mode on your terminal by issuing the CEOT TRANIDONLY command:

Note: Having a TCP/IP port mismatch will result in a similar scenario

10.9 XML parsing in CICS application

XML allows you to tag data in a way that is similar to how you tag data when creating an HTML file. XML incorporates many of the successful features of HTML, but was also developed to address some of the limitations of HTML. XML tags might be user-defined through a schema for later validation, which can either be a Document Type Definition (DTD) or a document written in the XML Schema language. In addition, namespaces can help ensure you have unique tags for your XML document. The syntax of XML has more restrictions than HTML, but this results in faster and cheaper browsing. The ability to create your own tagging structure gives you the power to categorize and structure data for both ease of retrieval and ease of display. XML is already being used for publishing, as well as for data storage and retrieval, data interchange between heterogeneous platforms, data transformations, and data displays. As it evolves and becomes more powerful, XML might allow for single-source data retrieval and data display.

The benefits of using XML vary but, overall, marked-up data and the ability to read and interpret that data provide the following benefits:

- ▶ With XML, applications can more easily read information from a variety of platforms. The data is platform-independent, so now the sharing of data between you and your customers can be simplified.
- ▶ Companies that work in the business-to-business (B2B) environment are developing DTDs and schemas for their industry. The ability to parse standardized XML documents gives business products an opportunity to be exploited in the B2B environment.
- ▶ XML data can be read even if you do not have a detailed picture of how that data is structured. Your clients will no longer need to go through complex processes to update how to interpret data that you send to them because the DTD or schema gives the ability to understand the information.

- ▶ Changing the content and structure of data is easier with XML. The data is tagged so you can add and remove elements without impacting existing elements. You will be able to change the data without having to change the application.

However, despite all the benefits of using XML, there are some things of which to be aware. First of all, working with marked up data can be additional work when writing applications because it physically requires more pieces to work together. Given the benefits of using XML, this additional work can reduce the amount of work needed to make a change in the future. Second, although it is a recommendation developed by the World Wide Web Consortium (W3C), XML (along with its related technologies and standards including Schema, XPath, and DOM/SAX APIs) is still a developing technology.

There are many methods we can use if we want to parse or generate XML in CICS application. In this section, we will introduce three methods which can be used to parse and generate XML.

- ▶ XML Toolkit for z/OS including Java edition and C edition
- ▶ COBOL High Speed XML parser
- ▶ CICS API - EXEC CICS TRANSFORM

10.9.1 XML Toolkit for z/OS

The XML Toolkit for z/OS provides the base infrastructure to integrate vertical and industry-specific data formats, structures, schemas, and metadata to ensure industry compliance of data representation and content. Some of its key uses include categorizing and tagging data for exchange in disparate environments, as well as transforming ad hoc unstructured data to XML records, enabling you to search, cross-reference, and share records. The toolkit includes the XML Parser, C++ Edition and the XSLT Processor, C++ Edition

XML Parser, C++ Edition allows an application to take advantage of the z/OS XML System Services component. A set of z/OS-specific parser classes have been implemented in the XML Parser, C++ Edition to provide this ability. These classes were created to mimic the existing SAX2 and DOM interfaces. They allow many applications to exploit the improved cost and performance characteristics of the z/OS XML System Services component with minimal changes to their code.

The toolkit supports applications running on both z/OS UNIX System Services and MVS environments.

For detailed information about the XML Toolkit for z/OS, you can refer to the following Web site:

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

10.9.2 COBOL High Speed XML parser

In this section we will look the COBOL high speed XML parser, looking into processing XML input, and producing XML output.

Processing XML input

You can process XML input in a COBOL program by using the XML PARSE statement.

The XML PARSE statement is the COBOL language interface to either of two high-speed XML parsers. You use the XMLPARSE compiler option to select the appropriate parser for your application:

- ▶ XMLPARSE(XMLSS) selects the z/OS XML System Services parser.
This option provides enhanced features such as namespace processing, validation of XML documents with respect to an XML schema, and conversion of text fragments to national character representation (Unicode UTF-16).
- ▶ XMLPARSE(COMPAT) selects the XML parser that is built into the COBOL library.
This option provides compatibility with XML parsing in Enterprise COBOL Version 3.

Processing XML input involves passing control between the XML parser and a processing procedure in which you handle parser events. See Figure 10-9.

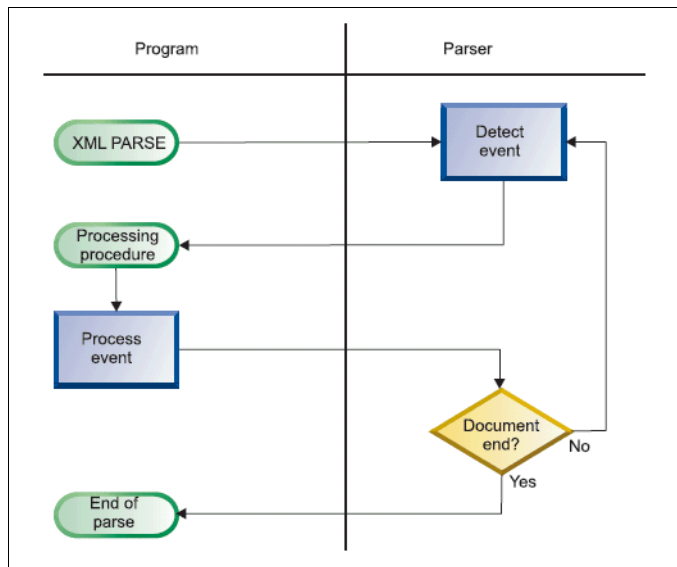


Figure 10-9 control between XML Parser and COBOL program

Producing XML output

You can produce XML output from a COBOL program by using the XML GENERATE statement.

To transform COBOL data to XML, use the XML GENERATE statement as in Example 10-34.

Example 10-34 XML GENERATE statement sample

```
XML GENERATE XML-OUTPUT FROM SOURCE-REC
COUNT IN XML-CHAR-COUNT
ON EXCEPTION
DISPLAY 'XML generation error ' XML-CODE
STOP RUN
NOT ON EXCEPTION
DISPLAY 'XML document was successfully generated.'
END-XML
```

In the XML GENERATE statement, identify the data item (XML-OUTPUT in the example above) that is to receive the XML output. Next, identify the source data item that is to be transformed to XML format (SOURCE-REC in the example). Optionally, you can code the COUNT IN phrase to obtain the number of XML character encoding units that are filled during generation of the XML output.

10.9.3 CICS API: EXEC CICS TRANSFORM

You can write application programs to transform application binary data into XML and vice versa. CICS supports a number of high-level languages and provides an XML assistant to map how the data is transformed during runtime processing. CICS uses the same technology for mapping application data to XML in SOAP messages, as part of the Web services support.

The advantage of using this approach to transform application data to and from XML is that CICS goes beyond the capabilities offered by an XML parser. CICS can interpret the XML and perform record-based conversions of the application data. Therefore, it is easier and faster for you to create applications that work with XML using this approach.

The steps to use CICS TRANSFORM API are as follows:

1. Create the mappings using the XML assistant.
The CICS XML assistant is a supplied utility that helps you to create the required artifacts to transform application binary data to XML or transform XML to application binary data. The XML assistant can create the artifacts in a bundle directory or another specified location on z/OS UNIXÆ.
2. Create the resources in CICS to make the mappings available.
3. Create or update an application program to use the TRANSFORM API command. The application must use a channel-based interface.
4. Run the application to test that the transformation works as you intended. You can turn on validation to check that CICS converts the data correctly.

In 10.2.2, “Parsing SOAP Fault messages in CICS TS V4.1” on page 221, there is an example that shows the steps necessary to use CICS API TRANSFORM to parse XML. In that example, XML is used as a soap fault in a soap message.



COBOL samples

In this chapter we provide a series of COBOL samples to demonstrate calling Web services from a CICS transaction. We show how COBOL programs can be written to handle several different XML constructs that can be found in a WSDL file. In particular we demonstrate the use of the following:

- ▶ The <any> tag
- ▶ The <choice> tag
- ▶ The minOccurs and maxOccurs tags

11.1 Introduction

The development strategy for all of the examples included in this chapter has been broadly similar.

1. Create a WSDL file to describe the Web service we have created.
2. Use this WSDL file in RDz to create a Web service skeleton.
3. Edit the Web service implementation code in the skeleton to do some meaningful processing and pass pack some data to the client (which in our case is a CICS transaction).
4. Run the WSDL file through the Web Services Assistant to produce the input and output language structures and a CICS bind file that can be used in defining the Web service to CICS.
5. Create a COBOL program that calls the Web service. We used RDz to create a skeleton COBOL program that is updated to initialize the language structures that are passed to the service and to process the reply received.

The goal is to provide working examples that can be implemented. We provide the following information for each example:

- ▶ The WSDL file
- ▶ The input and output language structures
- ▶ The COBOL source for the client transaction
- ▶ An EAR file that can be deployed to an application server that contains the Java source for the Web service.

11.2 Example 1: The <xsd:any> tag

In the first example we demonstrate the use of the XML <any> tag, which when used in a WSDL file indicates that at this point in the data supplied to the Web service there will be a section of XML which at this point is undefined. This then allows the client application to insert any piece of well formed XML into the data passed to the Web service. The client application is responsible for creating this XML and the Web service receiving it is responsible for parsing it and processing the data within it. In our example we pass a small piece of XML that is defined in a working storage location as follows:

```
Move '<whatever>.....</whatever>' to WS-CUST-XML
```

The Web service will echo this back to the client.

11.2.1 The WSDL

The WSDL used for this example contained the following section, which represents a customer. The extract can be seen in Example 11-1

Example 11-1 WSDL extract showing xsd:any

```
<xsd:complexType abstract="false" block="#all" final="#all"
mixed="false" name="ProgramInterface">
  <xsd:sequence>
    <xsd:element name="ws_reqarea" nillable="false">
      <xsd:complexType mixed="false">
        <xsd:sequence>
          <xsd:element name="Customer">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="Title"
type="xsd:string" />
                <xsd:element name="FirstName"
type="xsd:string" />
                <xsd:element name="Surname"
type="xsd:string" />
                <xsd:any minOccurs="0" />
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

The section of WSDL shows that we have three fields: Title, FirstName and Surname, which are all string fields. This is followed by a field described using an xsd:any tag that indicates the position a section of data that is undefined.

The complete WSDL can be seen in Appendix B, “Sample COBOL programs” on page 307

Now that we have the WSDL describing our Web service, we need to produce language structures (in our case COBOL copy books) that can be used in or client program along with a bind file that will be used to create the CICS resources need to call the Web service.

The tooling with RDz includes the CICS Web Services assistant which will produce these artifacts. In addition RDz will also produce a skeleton COBOL program which is an excellent starting point for creating a client CICS transaction from which we can call the Web service.

Web Services Assistant

The process for running the Web Services Assistant with RDz is simple. After RDz is running, switch to the Enterprise Service Tools Perspective. From this perspective you should then create a new Web Services for CICS Project, as shown in Figure 11-1

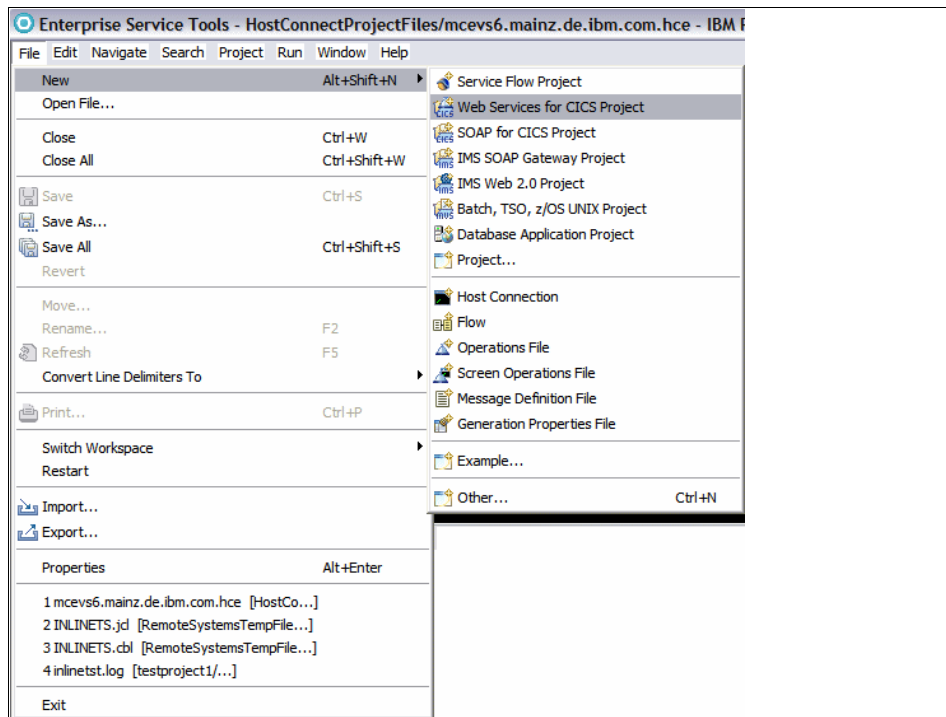


Figure 11-1 Create new Web Services for CICS Project

When you select this option a window will open where you should:

- ▶ Give the project a name
- ▶ Select **Create New Service Implementation (Top Down)**
- ▶ Select **Service Requester**
- ▶ Select **Interpretive XML Conversion** (the only option)

The New Web Services for CICS Project window will display, as in Figure 11-2.

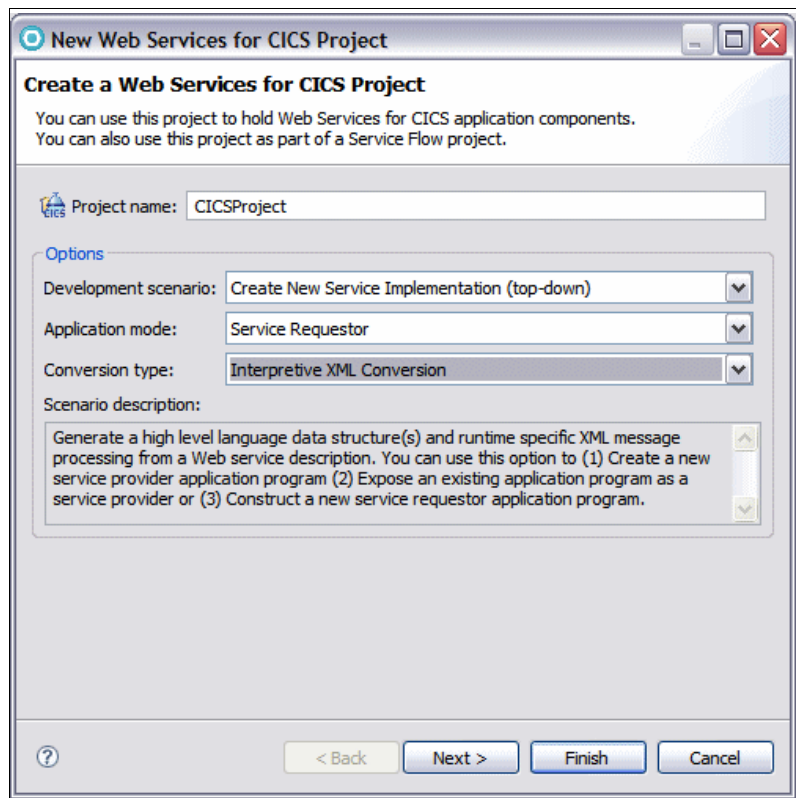


Figure 11-2 New Web Services for CICS Project window

Click **Next**. You will be presented with the window from which you will import the WSDL to be used for this project. See Figure 11-3. This window gives a choice of three locations from which you can import the WSDL file.

- ▶ The local file system
- ▶ Another workspace in RDz
- ▶ A file on a remote z/OS system

Locate the WSDL file by clicking the appropriate button. In our case the WSDL file was stored on a local hard drive, so the **File System** button was used.

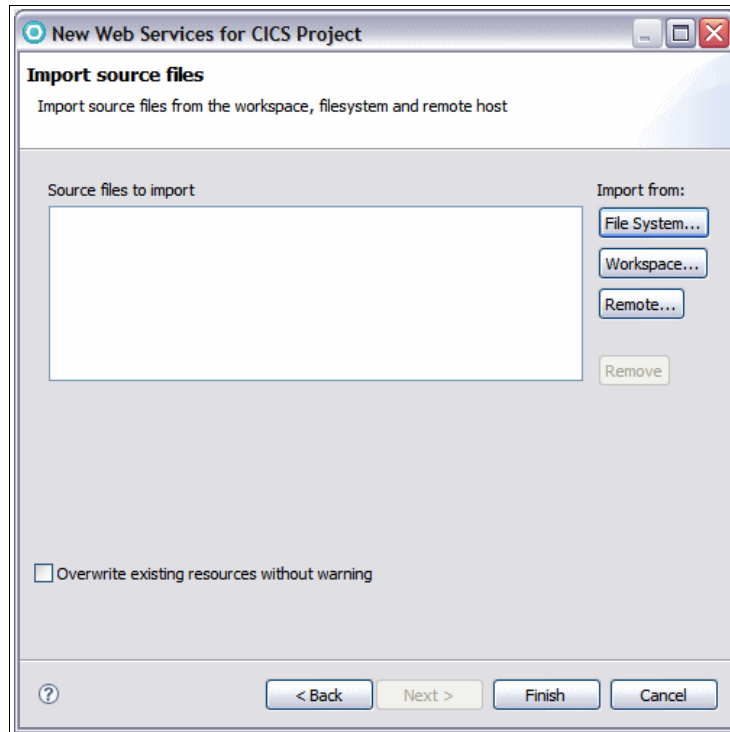


Figure 11-3 Import WSDL file

Click **Finish**, RDz displays the “Web Services for CICS - Create New Implementation” window. See Figure 11-4. We allowed all the fields and options presented in this window to default. We did, however ensure that the mapping level was set to at least 2.1 by pressing the **Change WSBIND Preferences** button and selecting mapping level 2.1 from the drop-down list.

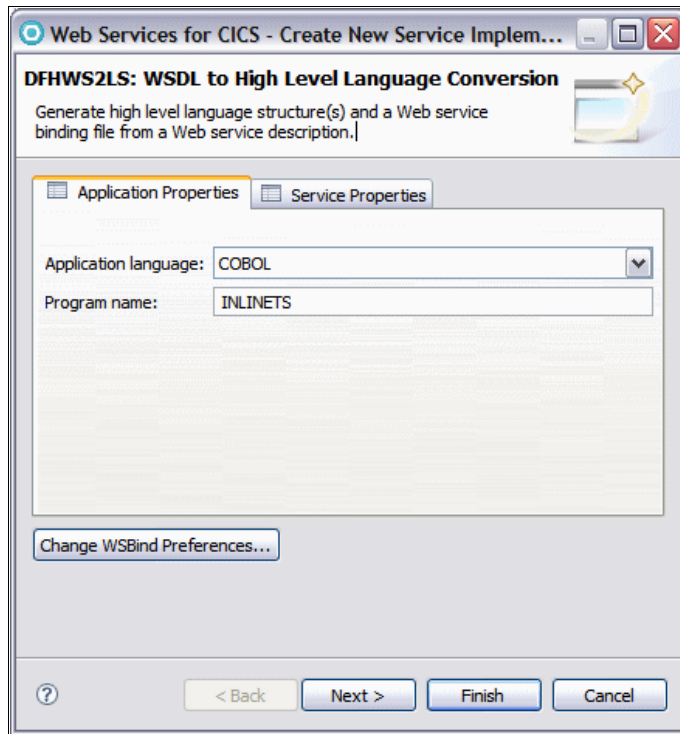


Figure 11-4 Create New Service Implementation

When the **Finish** button is clicked, RDz invokes the Web Services Assistant and several artifacts are created in your project. These include:

- ▶ Skeleton COBOL program
- ▶ Input copybook
- ▶ Output copybook
- ▶ Web services assistant log file
- ▶ wsbind file

These can be seen in Figure 11-5.

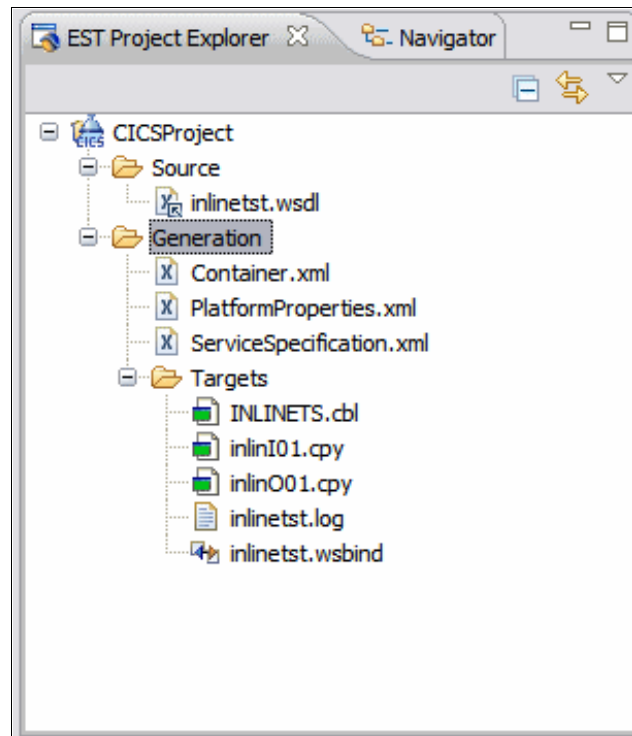


Figure 11-5 Artifacts

At this point we now have a COBOL program, two copybooks, and a wsbind file. The COBOL program is a skeleton and is discussed in more detail in 11.2.3, “The COBOL program” on page 265.

The wsbind file should be copied to UNIX Systems Services on the z/OS system where the CICS transaction will be run. This will be used when we install the pipeline which will be discussed in 11.2.4, “CICS resource definitions” on page 277.

11.2.2 Web Services Assistant: z/OS

The Web Services Assistant can also be run on z/OS by executing a batch job. In this example we would expect the copybooks produced to be identical to that of the Web Services Assistant in RDz. An example of the JCL used to run DFHWS2LS is included in Example 11-2.

A key difference between what the Web services assistant will produce in RDz to that on z/OS is that there is no skeleton COBOL program produced.

Example 11-2 DFHWS2LS

```
//WS2LS      EXEC DFHWS2LS,REGION=0M,
//           PATHPREF=' ',
//           TMPDIR='/tmp',
//           USSDIR='cicsts41',
//*          JAVADIR='java142s/J1.4/'
//           JAVADIR='java/J6.0'
//INPUT.SYSUT1 DD *
PDSLIB=//WAFITZ.U.COPY
LANG=COBOL
REQMEM=inlinI
RESPMEM=inlinO
LOGFILE=/u/wafitz/inline/inlinetst.log
WSBIND=/u/wafitz/inline/inlinetst.wsbind
WSDL=/u/wafitz/inline/inlinetst.wsdI
MAPPING-LEVEL=2.2
/*
//
```

11.2.3 The COBOL program

The RDz tooling was able to produce a skeleton COBOL program which we used as a starting point for our client program which will call our service.

The skeleton program in Example 11-3 on page 266 performs the following tasks.

- ▶ Sets up the container, channel and Web service names
- ▶ Leaves an open section for the programmer to set up the input language structure
- ▶ Puts the DFHWS-DATA container into the SERVICE-CHANNEL
- ▶ Invoke the Web service
- ▶ Retrieves the DFHWS-DATA container from the SERVICE-CHANNEL
- ▶ Leaves an open section to process the results of the service call

Example 11-3 Skeleton Program

```
IDENTIFICATION DIVISION.
*Begin Identification Division
PROGRAM-ID. 'INLINETS'.
AUTHOR. WD4Z.
INSTALLATION. 9.1.200.V200903111338.
DATE-WRITTEN. 17/09/09 13:47.
*End Identification Division
DATA DIVISION.
*Begin Data Division
WORKING-STORAGE SECTION.
*Begin Working-Storage Section
* *****
*           Operations Available On The Remote Web Service
* *****
1 OPERATION-NAME-1.
2 PIC X(17) USAGE DISPLAY
  VALUE 'INLINE01operation'.
*End Working-Storage Section
LOCAL-STORAGE SECTION.
*Begin Local-Storage Section
* *****
*           Program Work Variables
* *****
1 SOAP-PIPELINE-WORK-VARIABLES.
2 WS-WEBSERVICE-NAME PIC X(32).
2 WS-OPERATION-NAME PIC X(255).
2 WS-CONTAINER-NAME PIC X(16).
2 WS-CHANNEL-NAME PIC X(16).
2 COMMAND-RESP PIC S9(8) COMP.
2 COMMAND-RESP2 PIC S9(8) COMP.
*Specify A URI To Override The Web Service Description
1 URI-RECORD-STRUCTURE.
2 FILLER PIC X(10).
2 WS-URI-OVERRIDE PIC X(255).
* *****
*           Language Structures
* *****
1 LANG-INLINI01.
  COPY inlinI01.
1 LANG-INLIN001.
  COPY inlin001.
*End Local-Storage Section
LINKAGE SECTION.
```

```

*Begin Linkage Section
*End Linkage Section
*End Data Division
PROCEDURE DIVISION
.
*Begin Procedure Division
MAINLINE SECTION.
* -----
*           Initialize Work Variables
* -----
INITIALIZE SOAP-PIPELINE-WORK-VARIABLES.
INITIALIZE URI-RECORD-STRUCTURE.
* -----
* Container DFHWS-DATA must be present when a service requeste
* r program issues an EXEC CICS INVOKE WEBSERVICE command. Whe
* n the command is issued, CICS converts the language structur
* e that is in the container into a SOAP request. When the soa
* p response is received, CICS converts it into another langua
* ge structure that is returned to the application in the same
* container.
* -----
MOVE 'DFHWS-DATA'
TO WS-CONTAINER-NAME
* -----
*           Channel Passed To The Web Service Call
* -----
MOVE 'SERVICE-CHANNEL'
TO WS-CHANNEL-NAME
* -----
*           WEBSERVICE resource installed in this CICS region
* -----
MOVE 'inlinetst'
TO WS-WEBSERVICE-NAME
* -----
*           Operation To Invoke On The Remote Web Service
* -----
MOVE OPERATION-NAME-1
TO WS-OPERATION-NAME
* -----
*           Populate Request Language Structure
* -----
INITIALIZE LANG-INLINIO1
* .....
* .....
* .....

```

```

* -----
*       Put Request Language Structure Into SOAP Container
* -----
      EXEC CICS PUT CONTAINER(WS-CONTAINER-NAME)
          CHANNEL(WS-CHANNEL-NAME)
          FROM(LANG-INLINIO1)
      END-EXEC
      PERFORM CHECK-CONTAINER-COMMAND
* -----
*                               Invoke The Web Service
* -----
      EXEC CICS INVOKE WEBSERVICE(WS-WEBSERVICE-NAME)
          CHANNEL(WS-CHANNEL-NAME)
*     URI(WS-URI-OVERRIDE)
          OPERATION(WS-OPERATION-NAME)
          RESP(COMMAND-RESP) RESP2(COMMAND-RESP2)
      END-EXEC
      PERFORM CHECK-WEBSERVICE-COMMAND
* -----
*                               Receive Response Language Structure
* -----
      EXEC CICS GET CONTAINER(WS-CONTAINER-NAME)
          CHANNEL(WS-CHANNEL-NAME)
          INTO(LANG-INLINO01)
      END-EXEC
      PERFORM CHECK-CONTAINER-COMMAND
* -----
*                               Process Response Language Structure
* -----
*     ....
*     ....
*     ....
* -----
*                               Finished
* -----
      EXEC CICS RETURN
      END-EXEC
      .
CHECK-CONTAINER-COMMAND.
      EVALUATE COMMAND-RESP
          WHEN DFHRESP(CCSIDERR)
*     ....
          CONTINUE
          WHEN DFHRESP(CONTAINERERR)
*     ....

```

```

        CONTINUE
    WHEN DFHRESP(INVREQ)
*     ....
        CONTINUE
    WHEN DFHRESP(LENGERR)
*     ....
        CONTINUE
    END-EVALUATE
.
CHECK-WEBSERVICE-COMMAND.
    EVALUATE COMMAND-RESP
    WHEN DFHRESP(INVREQ)
*     ....
        CONTINUE
    WHEN DFHRESP(NOTFND)
*     ....
        CONTINUE
    END-EVALUATE
.
*End Procedure Division
END PROGRAM 'INLINETS'.

```

Note: The version of RDz we used generates an INVOKE WEBSERVICE command. From CICS TS 4.1 this command is now INVOKE SERVICE. However INVOKE WEBSERVICE is retained as a synonym of the INVOKE SERVICE command and is provided for compatibility with existing Web service requester applications

To change this skeleton into a working program we need to add code into the two empty sections of code: one before we make the service call and one after.

Populate Request Language Structure

Prior to making the service call we must populate the input language structure, which is in copy book inlinI01. We need to populate the Title, Firstname, and Surname fields and set up the XML data we will be passing to the service.

The first three fields require just simple moves of data into the appropriate working storage areas. In addition we need to set the length of each of these fields as seen in Example 11-4.

Example 11-4 Set up the request language structure

```
Move 'MR' TO XTitle of wsXreqarea
Move 2 to XTitle-length of wsXreqarea

Move 'Tony' TO FirstName of wsXreqarea
Move 4 to FirstName-length of wsXreqarea

Move 'Fitzgerald' TO Surname of wsXreqarea
MOVE 10 to Surname-length of wsXreqarea
```

The XML data that will be sent, is the undefined data area, which is defined by the <xsd:any> tag in the WSDL is slightly more complicated.

The <xsd:any> tag results in the Web services assistant generating two working storage fields of the form

- ▶ elementName-xml-cont PIC X(16)
- ▶ elementName-xmlns-cont PIC X(16)

in our case the generated language structures are

```
12 Customer-num          PIC S9(9) COMP-5 SYNC.
12 Customer.
    15 Customer-xml-cont      PIC X(16).
    15 Customer-xmlns-cont   PIC X(16).
```

The first field, Customer-xml-cont, must be set to the name of a container that holds the XML and the second Customer-xmlns-cont contains the name of a container that holds any namespace prefix declarations that are in scope.

In our example we have moved a short simple piece of XML into the container. This will be echoed back by the service.

The section of code that does this can be seen in Example 11-5.

Example 11-5 Populate the XML container

```
Move 1          to Customer-num      of wsXreqarea
MOVE 'cust-xml-cont' TO Customer-xml-cont of wsXreqarea
*              --- the XML ---
Move '<Whatever>.....</Whatever>' to WS-CUST-XML

EXEC CICS PUT CONTAINER(Customer-xml-cont of wsXreqarea)
```



```
CHANNEL(WS-CHANNEL-NAME)
FROM(WS-CUST-XML)
DATATYPE(DFHVALUE(CHAR))
END-EXEC
```

After the containers have been populated the service is called using an EXEC CICS INVOKE SERVICE command. We uncommented the URI parameter so that we could set the URI of the Web service being called. The URI parameter specifies a data area containing the URI of the service to be invoked. If specified, this option supersedes any URI specified in the WEBSERVICE resource definition. If you omit this option, the WEBSERVICE binding file associated with the resource definition must include either a provider URI or a provider application name.

Additionally we added some code to the CHECK-CONTAINER-COMMAND and CHECK-WEBSERVICE-COMMAND sections of the program

Processing the response language structure

Having called the Web service we must process the results that have been sent back to our program. In our example we have displayed the results of the data areas in the response structure.

The complete final program can be seen in Example 11-6.

Example 11-6 The Final Program

```
IDENTIFICATION DIVISION.
*Begin Identification Division
PROGRAM-ID. 'INLINETS'.
AUTHOR. WD4Z.
INSTALLATION. 9.1.200.V200903111338.
DATE-WRITTEN. 09/09/09 15:16.
*End Identification Division
DATA DIVISION.
*Begin Data Division
WORKING-STORAGE SECTION.
*Begin Working-Storage Section
* *****
*           Operations Available On The Remote Web Service
* *****
1 OPERATION-NAME-1.
2 PIC X(17) USAGE DISPLAY
  VALUE 'INLINE010peration'.
*End Working-Storage Section
LOCAL-STORAGE SECTION.
```

```

*Begin Local-Storage Section
* *****
*
*           Program Work Variables
* *****
1 SOAP-PIPELINE-WORK-VARIABLES.
2 WS-WEBSERVICE-NAME PIC X(32).
2 WS-OPERATION-NAME PIC X(255).
2 WS-CONTAINER-NAME PIC X(16).
2 WS-CHANNEL-NAME PIC X(16).
2 COMMAND-RESP PIC S9(8) COMP.
2 COMMAND-RESP2 PIC S9(8) COMP.
*Specify A URI To Override The Web Service Description
1 URI-RECORD-STRUCTURE.
2 FILLER PIC X(10).
2 WS-URI-OVERRIDE PIC X(255).

1 WS-XML-PASSTHRU-DATA.
2 WS-CUST-XML PIC X(255).
2 WS-CUST-XMLns PIC X(255).

1 WS-DFHWS-BODY PIC x(400).

* *****
*
*           Language Structures
* *****
1 LANG-INLINI01.
  COPY inlinI01.
1 LANG-INLIN001.
  COPY inlin001.
*End Local-Storage Section
LINKAGE SECTION.
*Begin Linkage Section
*End Linkage Section
*End Data Division
PROCEDURE DIVISION
.
*Begin Procedure Division
MAINLINE SECTION.
* -----
*
*           Initialize Work Variables
* -----
INITIALIZE SOAP-PIPELINE-WORK-VARIABLES.
INITIALIZE URI-RECORD-STRUCTURE.
* -----
* Container DFHWS-DATA must be present when a service requeste

```

```

* r program issues an EXEC CICS INVOKE WEBSERVICE command. Whe
* n the command is issued, CICS converts the language structur
* e that is in the container into a SOAP request. When the soa
* p response is received, CICS converts it into another langua
* ge structure that is returned to the application in the same
* container.

```

```

* -----
      MOVE 'DFHWS-DATA'
      TO WS-CONTAINER-NAME

```

```

* -----
*           Channel Passed To The Web Service Call

```

```

* -----
      MOVE 'SERVICE-CHANNEL'
      TO WS-CHANNEL-NAME

```

```

* -----
*           WEBSERVICE resource installed in this CICS region

```

```

* -----
      MOVE 'inlinetst'
      TO WS-WEBSERVICE-NAME

```

```

* -----
*           Operation To Invoke On The Remote Web Service

```

```

* -----
      MOVE OPERATION-NAME-1
      TO WS-OPERATION-NAME

```

```

* -----
*           Populate Request Language Structure

```

```

* -----
      INITIALIZE LANG-INLINI01

```

```

      Move 'MR' TO XTitle of wsXreqarea
      Move 2 to XTitle-length of wsXreqarea

```

```

      Move 'Tony' TO FirstName of wsXreqarea
      Move 4 to FirstName-length of wsXreqarea

```

```

      Move 'Fitzgerald' TO Surname of wsXreqarea
      MOVE 10 to Surname-length of wsXreqarea

```

```

      INITIALIZE WS-XML-PASSTHRU-DATA

```

```

* -----
*           Put the "any" XML data into the channel

```

```

* -----

```

```

Move 1          to Customer-num      of wsXreqarea
MOVE 'cust-xml-cont' TO Customer-xml-cont of wsXreqarea

*           --- the XML ---
Move '<Whatever>.....</Whatever>' to WS-CUST-XML

EXEC CICS PUT CONTAINER(Customer-xml-cont of wsXreqarea)
      CHANNEL(WS-CHANNEL-NAME)
      FROM(WS-CUST-XML)
      DATATYPE(DFHVALUE(CHAR))
END-EXEC
PERFORM CHECK-CONTAINER-COMMAND

* -----
* Put the "any" XMLns data into the channel
* -----
MOVE 'cust-xmlns-cont'          to Customer-xmlns-cont
                                of wsXreqarea
* Move 'xmlns:ns1="http://myNS"' to WS-CUST-XMLns

* -----
* Put Request Language Structure Into SOAP Container
* -----

EXEC CICS PUT CONTAINER(WS-CONTAINER-NAME)
      CHANNEL(WS-CHANNEL-NAME)
      FROM(LANG-INLINIO1)
END-EXEC
PERFORM CHECK-CONTAINER-COMMAND

* -----
* Invoke The Web Service
* -----
Move 'http://9.173.198.188:9080/RedbookWS4/INLINE01Service'
to WS-URI-OVERRIDE

EXEC CICS INVOKE SERVICE(WS-WEBSERVICE-NAME)
      CHANNEL(WS-CHANNEL-NAME)
      URI(WS-URI-OVERRIDE)
      OPERATION(WS-OPERATION-NAME)
      RESP(COMMAND-RESP) RESP2(COMMAND-RESP2)
END-EXEC
PERFORM CHECK-WEBSERVICE-COMMAND

```

```

* -----
*           Receive Response Language Structure
* -----
EXEC CICS GET CONTAINER(WS-CONTAINER-NAME)
      CHANNEL(WS-CHANNEL-NAME)
      INTO(LANG-INLINO01)
END-EXEC
PERFORM CHECK-CONTAINER-COMMAND

* -----
*           "Process" the Response Language Structure
* -----
DISPLAY 'XTitle data returned   = ' XTitle of wsXretarea
DISPLAY 'FirstName data returned = ' FirstName of wsXretarea
DISPLAY 'Surname data returned  = ' Surname of wsXretarea

INITIALIZE WS-XML-PASSTHRU-DATA.

EXEC CICS GET CONTAINER(Customer-xmlns-cont of wsXretarea)
      CHANNEL(WS-CHANNEL-NAME)
      INTO(WS-CUST-XMLns)
END-EXEC
PERFORM CHECK-CONTAINER-COMMAND

EXEC CICS GET CONTAINER(Customer-xml-cont of wsXretarea)
      CHANNEL(WS-CHANNEL-NAME)
      INTO(WS-CUST-XML)
END-EXEC
PERFORM CHECK-CONTAINER-COMMAND

DISPLAY 'Customer-xml-cont data = ' WS-CUST-XML
DISPLAY 'Customer-xmlns-cont data = ' WS-CUST-XMLns

* -----
*                               Finished
* -----
EXEC CICS RETURN
END-EXEC
.

CHECK-CONTAINER-COMMAND.
EVALUATE COMMAND-RESP
  WHEN DFHRESP(CCSIDERR)
    EXEC CICS ABEND ABCODE('C001') END-EXEC

```

```

        CONTINUE
    WHEN DFHRESP(CONTAINERERR)
        EXEC CICS ABEND ABCODE('C002') END-EXEC
        CONTINUE
    WHEN DFHRESP(INVREQ)
        EXEC CICS ABEND ABCODE('C003') END-EXEC
        CONTINUE
    WHEN DFHRESP(LENGERR)
        EXEC CICS ABEND ABCODE('C004') END-EXEC
        CONTINUE
END-EVALUATE
.

CHECK-WEBSERVICE-COMMAND.
EVALUATE COMMAND-RESP
    WHEN DFHRESP(INVREQ)
        PERFORM INVREQ-PROCESSING
        EXEC CICS ABEND ABCODE('WS01') END-EXEC
        CONTINUE
    WHEN DFHRESP(NOTFND)
        EXEC CICS ABEND ABCODE('WS02') END-EXEC
        CONTINUE
END-EVALUATE
.

INVREQ-PROCESSING.
IF EIBRESP2 = 6 THEN
*   ** An EIBRESP2 of 6 indicates a SOAP fault **
*   ** has been returned in DFHWS-BODY          **
    EXEC CICS
        GET CONTAINER('DFHWS-BODY')
        CHANNEL(WS-CHANNEL-NAME)
        INTO(WS-DFHWS-BODY)
    END-EXEC
    DISPLAY WS-DFHWS-BODY
END-IF
.

*End Procedure Division
END PROGRAM 'INLINETS'.

```

11.2.4 CICS resource definitions

A minimum of two CICS resource definitions are required to run the example program:

- ▶ A transaction Definition
- ▶ A pipeline Definition

If program auto install is not used, a program definition will be required.

The transaction definition should point to the name of the requester program we have created. The program should have been compiled and linked into a load library which is in the DFHRPL concatenation.

For the pipeline definition we set the Configfile to point to the supplied basicsoap11requester.xml file. The Wsdir value must be set to point to the directory where the wsbind file, created by the Web Services Assistant, has been stored, as seen in Figure 11-6.

```
COnfigfile ==>
/usr/lpp/cicsts/cicsts41/samples/pipelines/basicsoap11requ
(Mixed Case) ==> ester.xml
==>
==>
==>
S Helf      ==> /var/cicsts/
(Mixed Case) ==>
==>
==>
==>
W sdir      : /wafitz/wsbind
```

Figure 11-6 Pipeline definition

Results of calling the service

The Web service has been designed to echo back the contents of the fields sent to it, including the XML text sent to it in the WS-CUST-XML field. The CICS job log should show the resultant displayed data which can be seen in Example 11-7.

Example 11-7 Results of <xsd:any> program

XTitle data returned = you said: MR

FirstName data returned = you said: Tony

Surname data returned = you said: Fitzgerald

Customer-xml-cont data = <Whatever
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">.....</Whate
ver>

Customer-xmlns-cont data =
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns="http://www.INLINE01.REQY.Request.com"
xmlns:ns2="http://www.INLINE01.RESPY.Response.com"

11.3 Example 2: The <choice> tag

This example demonstrates how a COBOL program can handle the use of the <xsd:choice> tag in a WSDL describing a Web service to be called by a CICS application. The <xsd:choice> tag indicates that only one of the options listed can be used.

11.3.1 The WSDL

The part of the WSDL used in our example that describes the choice data used can be seen in Example 11-8 on page 279.

Example 11-8 WSDL extract showing <xsd:choice>

```
<xsd:complexType abstract="false" block="#all" final="#all" mixed="false"
name="ProgramInterface">
  <xsd:sequence>
    <xsd:element name="ws_retarea" nillable="false">
      <xsd:complexType mixed="false">
        <xsd:sequence>
          <xsd:element name="choiceData">
            <xsd:complexType>
              <xsd:choice>
                <xsd:element name="firstchoice" type="xsd:string" />
                <xsd:element name="secondchoice" type="xsd:string" />
              </xsd:choice>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

In our example we have an `xsd:choice` element called `choiceData`, which contains two elements: one called `firstchoice` and the second called `secondchoice`. The Web service when called should be supplied with only one of the two elements.

The complete WSDL can be seen in “WSDL <xsd:choice>” on page 330.

11.3.2 Generation of COBOL and CICS artifacts

We generated the following artifacts from the Web services assistant in RDz.

- ▶ COBOL skeleton program
- ▶ Input and output copybooks
- ▶ wsbind file

11.3.3 The COBOL program

The COBOL skeleton program produced will be similar to the skeleton produced for the `<xsd:any>` example and shown in Example 11-3 on page 266. In the same way as we did for that example, we must add code to populate the request language structure before calling the service and also to process the results returned by the service.

Setting up the Request language structure

In our example we are going to send data in the firstchoice field to the service.

Set firstchoice to **true**. When the language structure is generated from the WSDL the <xsd:choice> tag results in a COBOL working storage area, which contains, in addition to the data fields to be sent to the service, two further data fields. The first is a flag field that indicates which of the choice variables is being sent to the service. The second field is the name of the container that will contain the data area being sent to the service. In our example this part of the language structure is as follows:

```
03 INLINE010peration.
    06 wsXreqarea.
        09 choiceData.

            12 choiceData-enum                PIC X DISPLAY.
                88 empty                      VALUE X'00'.
                88 firstchoice                VALUE X'01'.
                88 secondchoice              VALUE X'02'.
            12 choiceData-cont                PIC X(16).
```

This is followed by two data areas one representing each of the two possible data areas which can be sent to the service. Only one of these will be set and then placed into the container named in choiceData-cont.

```
01 choicI01-firstchoice.
    03 firstchoice-length                    PIC S9999 COMP-5 SYNC.
    03 firstchoice                          PIC X(255).

01 choicI01-secondchoice.
    03 secondchoice-length                  PIC S9999 COMP-5 SYNC.
    03 secondchoice                        PIC X(255).
```

The COBOL code to populate the request language structure can be seen in Example 11-9.

Example 11-9 Populate request language structure

```
INITIALIZE LANG-CHOICI01

    DISPLAY 'data is being sent in the firstchoice field'
*
* The WSDL specifies that only one of the two fields can
* be sent to the service
*   EITHER firstchoice or secondchoice
*
    move 'first choice data' to firstchoice
      of choicI01-firstchoice
```

```

move 18 to firstchoice-length
      of choicI01-firstchoice

DISPLAY 'data to be sent is ==>' firstchoice
      of choicI01-firstchoice

set firstchoice of wsXreqarea to true

move 'CHOICE-CONT' to choiceData-cont of wsXreqarea

EXEC CICS PUT CONTAINER(choiceData-cont of wsXreqarea)
      CHANNEL(WS-CHANNEL-NAME)
      FROM(choicI01-firstchoice)
END-EXEC
PERFORM CHECK-CONTAINER-COMMAND

```

Processing the Response language structure

On return from the service call the response language structure will contain the result of the call to the service. Our example consists of a COBOL EVALUATE statement that tests which of the data areas have been sent back to the requester and then displays the data sent back. This can be seen in Example 11-10.

Example 11-10 Processing the response language structure

```

EVALUATE TRUE
  when empty of wsXretarea
    display 'nothing returned'

  when firstchoice of wsXretarea
    display 'data was returned in the firstchoice field'

    EXEC CICS GET CONTAINER(choiceData-cont of wsXretarea)
          CHANNEL(WS-CHANNEL-NAME)
          INTO(choic001-firstchoice)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

    display 'data returned is ==>'
          firstchoice of choic001-firstchoice

  when secondchoice of wsXretarea
    display 'data was returned in the secondchoice field'

    EXEC CICS GET CONTAINER(choiceData-cont of wsXretarea)
          CHANNEL(WS-CHANNEL-NAME)

```

```
        INTO(choic001-secondchoice)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

    display 'data returned is ==>'
        secondchoice of choic001-secondchoice

END-EVALUATE
```

The complete COBOL program can be seen in “<?xml version="1.0"?>” on page 314.

11.3.4 CICS Resource Definitions

As for the <xsd:any> example we required two CICS resources to be defined to run the service.

- ▶ Transaction Definition
- ▶ Pipeline Definition

See 11.2.4, “CICS resource definitions” on page 277 for full details.

Results of calling the service

The service is designed to receive one of the two choice fields and echo the data back in the other of the two choice fields. The output written to the CICS joblog should look something like Example 11-11.

Example 11-11 Results of choice test

```
data is being sent in the firstchoice field
data to be sent is ==>first choice data
```

```
data was returned in the secondchoice field
data returned is ==>first choice data
```

11.4 Example 3: minoccurs and maxoccurs

This example demonstrates how a COBOL program can handle the use of the minoccurs and maxoccurs tag in a WSDL describing a Web service to be called by a CICS application. See Example 11-12 for WSDL showing minoccurs and max occurs

Example 11-12 WSDL Showing minoccurs and max occurs

```
<xsd:complexType abstract="false" block="#all" final="#all" mixed="false"
name="ProgramInterface">
  <xsd:sequence>
    <xsd:element name="ws_retarea" nillable="false">
      <xsd:complexType mixed="false">
        <xsd:sequence>
          <xsd:element maxOccurs="10" minOccurs="1" name="recs"
nillable="false">
            <xsd:complexType mixed="false">
              <xsd:sequence>
                <xsd:element name="recs" nillable="false">
                  <xsd:simpleType>
                    <xsd:annotation>
                      <xsd:appinfo
source="http://www.ibm.com/software/hcp/cics/annotations">
                        com.ibm.cics.wsd1.properties.charlength=fixed
com.ibm.cics.wsd1.properties.synchronized=false</xsd:appinfo>
                    </xsd:annotation>
                    <xsd:restriction base="xsd:string">
                      <xsd:maxLength value="80" />
                      <xsd:whiteSpace value="collapse" />
                    </xsd:restriction>
                  </xsd:simpleType>
                </xsd:element>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

In our example we have a single text field defined which must occur at least once (minOccurs=1) but no more than 10 times (maxOccurs=10). The complete WSDL can be seen in "WSDL - minOccurs/maxOccurs" on page 344.

11.4.1 Generation of COBOL and CICS artifacts

As in our previous examples we have generated the following from RDz

- ▶ COBOL skeleton program
- ▶ Input and output copybooks
- ▶ wsbind file

11.4.2 The COBOL Program

The COBOL skeleton program requires us to set up the request language structure prior to calling the service and then to add code to process the results.

Setting up the Request language structure

For our example we have a field called recs which is 80 bytes long and can occur between 1 and 10 times. The Web services assistant has generated a copy book which contains a working storage variable for the 80 byte record and two further variables to define the number of records being sent and also the name of the container which will contain the records. The working storage variables concerned are as follows:

```
03 INLINE010peration.  
    06 wsXreqarea.  
        09 recs-num                PIC S9(9) COMP-5 SYNC.  
        09 recs-cont                PIC X(16).
```

We need to write the data records to the container using a loop in which we put the data records into an array, which is then written to the container. The code to set up the request language structure can be seen in Example 11-13. In our example we have chosen to send four records to the Web service.

Example 11-13 minOccurs/maxOccurs request language structure setup

```
INITIALIZE LANG-REDBO101  
  
*--- we are going to send 4 records  
    move 4 to recs-num of wsXreqarea  
    DISPLAY " "  
    DISPLAY "=====  
    DISPLAY "Sending " recs-num of wsXreqarea " records"  
  
*--- populate our array with our data  
    Perform recs-num of wsXreqarea times  
        add 1 to ws-count  
        Move ws-record-data to recs2 of redboI01-recs
```

```

        move redboI01-recs to WS-RECORD(ws-count)
    END-Perform

*--- calculate how long the data is
    compute records-length =
        length of redboI01-recs * recs-num of wsXreqarea

*--- store the name of our data container in the
*--- request language structure
    move "RECS-CONTAINER" to recs-cont of wsXreqarea

*--- put the array into the container
    EXEC CICS PUT CONTAINER(recs-cont of wsXreqarea)
        CHANNEL(WS-CHANNEL-NAME)
        FROM(WS-RECORDS-ARRAY)
        FLENGTH(records-length)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

```

Processing the Response language structure

As in previous examples, the response language structure will contain the data returned from the service. Our processing will be to display the results using COBOL DISPLAY. The number of data records returned will be found in the recs-num field of the response language structure. This allows the program to loop around the required number of times to extract the data.

The response processing can be seen in Example 11-14.

Example 11-14 minOccurs/maxOccurs response processing

```

*--- get the returned data which is in the container
*--- named in the response language structure
    EXEC CICS GET CONTAINER(recs-cont of wsXretarea)
        CHANNEL(WS-CHANNEL-NAME)
        INTO(WS-RECORDS-ARRAY)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

    DISPLAY '===== '
    DISPLAY recs-num of wsXretarea ' records returned'

*--- Display each of the returned records.
*--- The number of records returned is in recs-num
*--- which has been extracted from the response container

```

```

*--- into the response language structure
  move 1 to ws-count
  PERFORM recs-num of wsXretarea times
    move ws-count to ws-returned-rec-num
    MOVE ws-record(ws-count) to ws-returned-rec-data
    DISPLAY ws-record-returned
    add 1 to ws-count

  END-PERFORM

```

The complete COBOL program can be seen in “Program to call minOccurs/maxOccurs example service” on page 338.

11.4.3 CICS Resource Definitions

As for the previous examples we defined two CICS resources to call the service.

- ▶ Transaction definition
- ▶ Pipeline Definition

See 11.2.4, “CICS resource definitions” on page 277.

11.4.4 Results of calling the service

Our program calls the Web service passing it four data records. The Web service is designed to send back the number of records we have sent plus two more.

The display output from a successful call to the service should look something such as the following example output in Example 11-15.

Example 11-15 Results of the minOccurs/maxOccurs test

```

=====
Sending 0000000004 records
=====
0000000006 records returned
  returned record number 01====> string array number 0
  returned record number 02====> string array number 1
  returned record number 03====> string array number 2
  returned record number 04====> string array number 3
  returned record number 05====> string array number 4
  returned record number 06====> string array number 5

```

We also ran a test to show what happens if you try to send more data to the service than is allowed by the WSDL definition, In this case the maximum number of records allowed is 10. We ran the program changing the number to 14. The call to the service then failed and message DFHPI1008 was written to MSGUSR.

```
DFHPI1008 10/19/2009 19:09:09 IV3A66A2 00186 XML generation failed
because of incorrect input (INPUT_ARRAY_TOO_LARGE recs2) for
WEBSERVICE redbookWS6.
```

The pipeline has validated the number of records being sent against the WSDL definition and has indicated through the message that the input array is too large. The EXEC CICS INVOKE SERVICE call will return an INVREQ condition with an EIBRESP2 value of 13. The full error message is also available to the program in the DFH-XML-ERRORMSG container. We have extracted this and displayed it to the console to demonstrate how to access the container when this condition is encountered. See Example 11-16.

Example 11-16 Error handling example

```

*      ** An EIBRESP2 of 13 indicates an input error **
*      ** has been detected a message is returned **
*      ** in DFH-XML-ERRORMSG **
IF EIBRESP2 = 13 THEN
  EXEC CICS
    GET CONTAINER('DFH-XML-ERRORMSG')
    CHANNEL(WS-CHANNEL-NAME)
    INTO(WS-XML-ERRORMSG)
  END-EXEC
  DISPLAY WS-XML-ERRORMSG
```



A

Sample Web services

In Example A-1 on page 294 through Example A-3 on page 302 we list the sample Web services that we use to demonstrate the COBOL requester programs in Chapter 11, “COBOL samples” on page 257.

Preparation of your RDz environment

Make sure that your copy of the IBM Rational Developer for System z is open and shows up in the Java EE perspective. If not, you can easily switch your perspective by clicking **Window** → **Open Perspective** → **Other** from the menu bar, then select **Java EE**. Your IDE should look like Figure A-1.

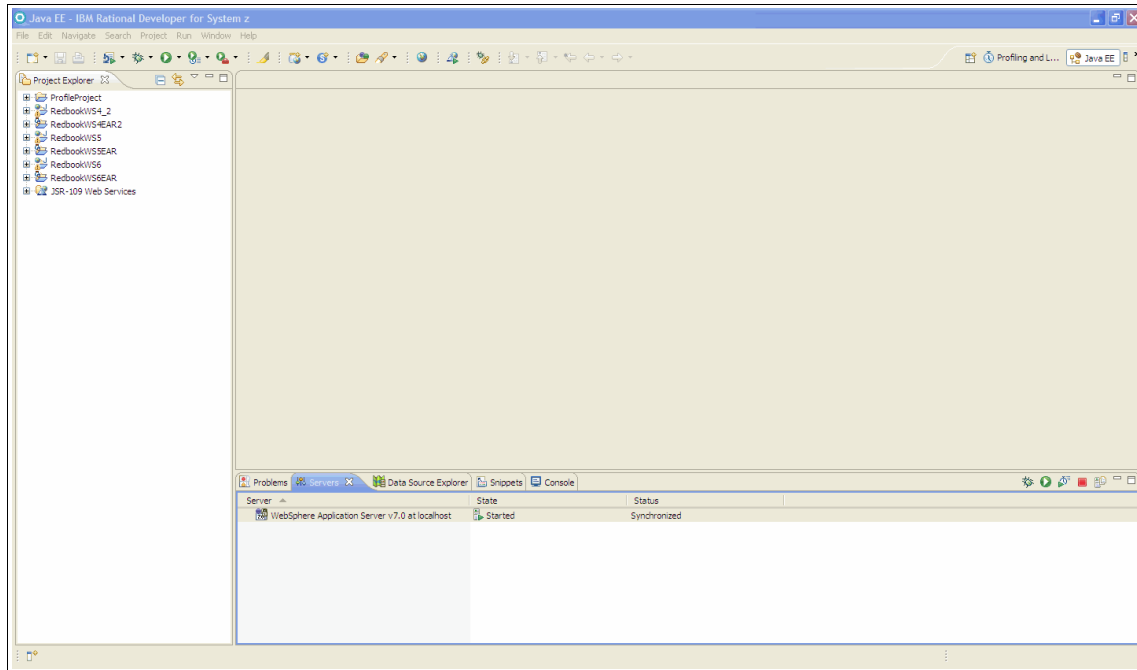


Figure A-1 RDZ Java EE perspective

You are ready to load the .ear files which contain the web service examples.

Loading an .ear file into a new or existing project

Perform the following steps to load an .ear file in a new or existing project:

1. With your IDE in Java EE perspective, choose **File** → **Import** from the menu bar, and select **EAR file** from the Java EE folder, as shown in Figure A-2.

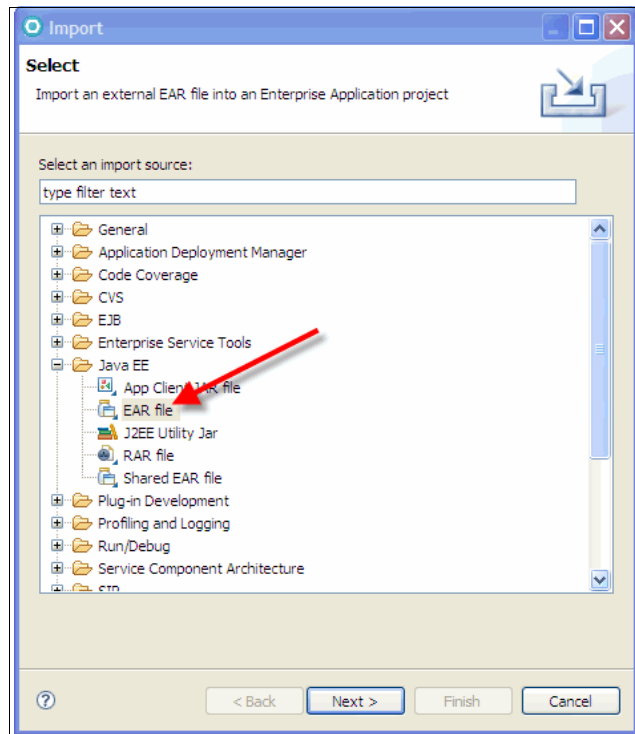


Figure A-2 RDZ import selection box

2. Click **Next**.
3. Specify the location of the .ear file in the following window, as shown in Figure A-3. Specify the Project's new name and the target runtime. We recommend WebSphere Application Server v7.0.

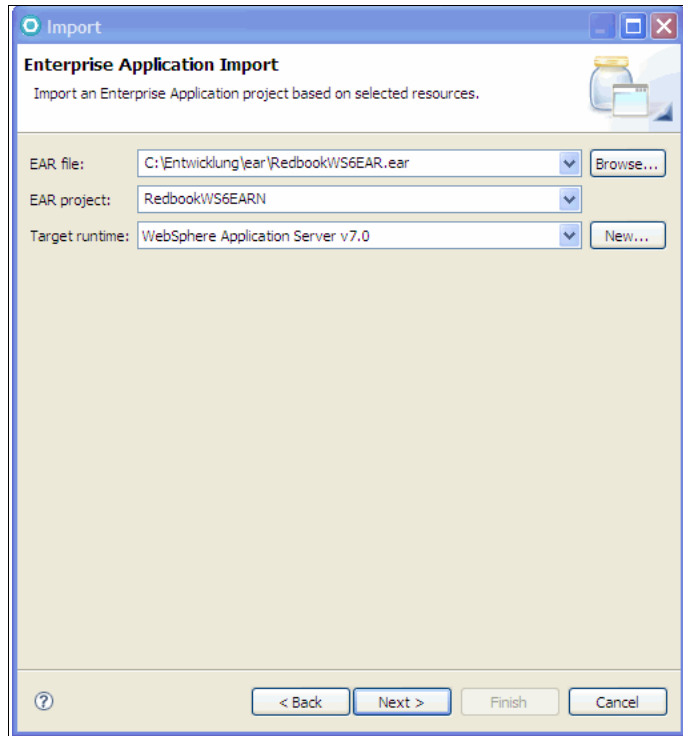


Figure A-3 RDZ Enterprise Application Import Box

4. Click **Next**. Another “Enterprise Application Import” window displays. You have the opportunity to include your existing projects to use them with the example. Each of them get a subdirectory in the new project. You can choose the modules from the .ear file to import into the new project. In our case, there is just one module.

5. Click **Finish** in the box shown in Figure A-4. IBM Rational Developer imports the .ear file.

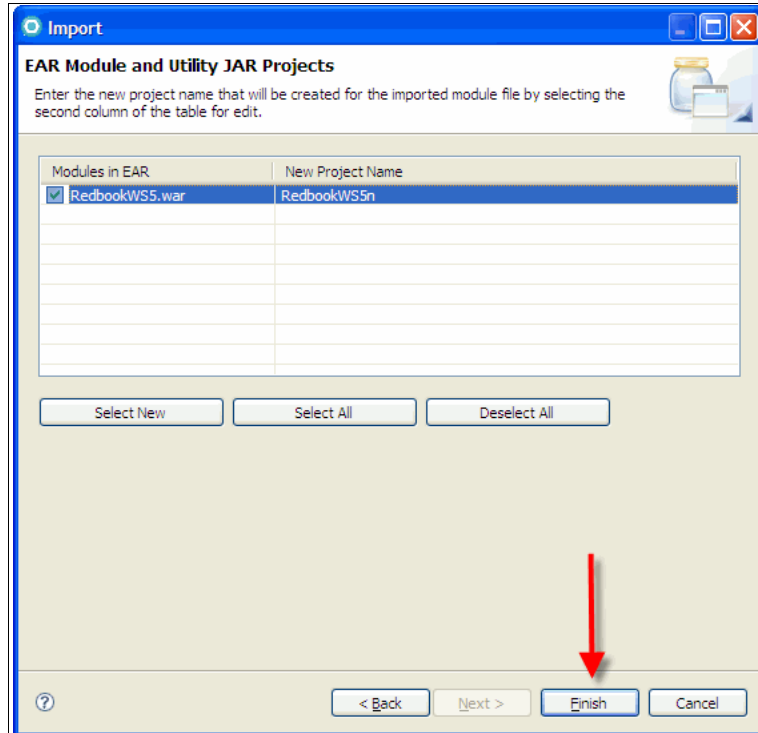


Figure A-4 RDZ Module and Utility box

Note: To use the examples for testing, you also have to deploy the program on the WebSphere server.

Description of examples A1–A3

The following pages contain a short description about what these programming examples do, plus their coding and a testing manual for each one.

The XML any passthrough Web service example

This example shows how a Web service can transmit XML code within a SOAP message without interpreting the foreign code, so that the requester gets XML code out of the envelope. You can install the file by following the steps in the

“Preparation of your RDz environment” on page 290 and “Loading an .ear file into a new or existing project” on page 291.

Coding

The code given in Example A-1 is, for the most part, automatically derived from the *.wsdl file. You only have to add the problems' solution to the Java skeleton.

Example A-1 Code of the XML any passthrough web service example

```
package com.reqy.inline01;

@javax.jws.WebService(endpointInterface="com.reqy.inline01.INLINE01Port",
    targetNamespace="http://www.INLINE01.REQY.com",
    serviceName="INLINE01Service", portName="INLINE01Port")

public class INLINE01HTTPSoapBindingImpl{

    public com.response.respy.inline01.ProgramInterface
    inline01Operation(com.request.reqy.inline01.ProgramInterface
    requestPart){
        // Here starts the actual problem solution
        // Allocating references for the exchange
        com.response.respy.inline01.ProgramInterface respref = new
        com.response.respy.inline01.ProgramInterface();
        com.response.respy.inline01.ProgramInterface.WsRetarea wsarearef = new
        com.response.respy.inline01.ProgramInterface.WsRetarea();
        com.response.respy.inline01.ProgramInterface.WsRetarea.Customer custref
        = new
        com.response.respy.inline01.ProgramInterface.WsRetarea.Customer();

        com.request.reqy.inline01.ProgramInterface reqref =new
        com.request.reqy.inline01.ProgramInterface();
        com.request.reqy.inline01.ProgramInterface.WsReqarea reqrefws = new
        com.request.reqy.inline01.ProgramInterface.WsReqarea();
        com.request.reqy.inline01.ProgramInterface.WsReqarea.Customer reqrefcus
        = new com.request.reqy.inline01.ProgramInterface.WsReqarea.Customer();
        // Processing the input
        reqrefws = requestPart.getWsReqarea();
        reqrefcus = reqrefws.getCustomer();
        // Composition of the answer
        custref.setTitle("you said: " + reqrefcus.getTitle());
        custref.setFirstName("you said: " + reqrefcus.getFirstName());
        custref.setSurname("you said: " + reqrefcus.getSurname());
        custref.setAny(reqrefcus.getAny());
        wsarearef.setCustomer(custref);
    }
}
```



```

respref.setWsRetarea(wsarearef);
    return respref;
}
}

```

Testing

With the IBM Rational Developer for System z, you can easily test the function of every .wsdl file using the Web services explorer. For this example, follow these steps:

1. Open the Services subdirectory in your project's folder, then right-click **INLINE01Service**, and choose **Test with Web Services Explorer**, as shown in Figure A-5.

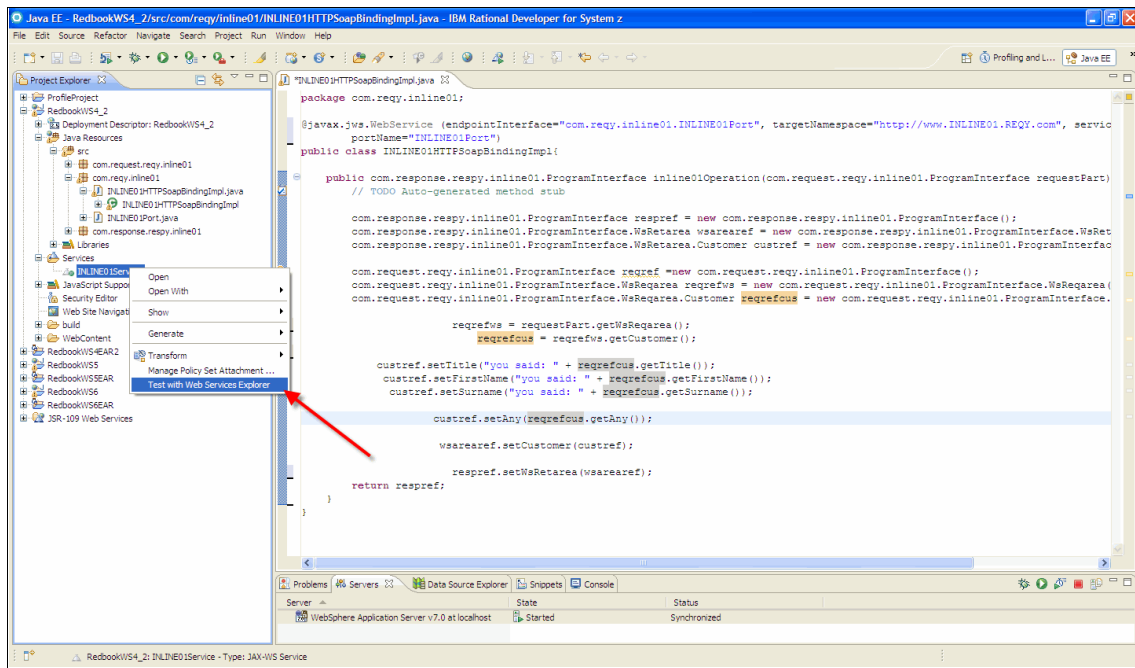


Figure A-5 Location of the Web Services Explorer in RDZ

2. Maximize the new window, the Web Services Explorer, by double-clicking on its tab. Confirm the standard endpoint and click **Go**.
3. Invoke a new WSDL Operation. Type in strings for **Title**, **FirstName**, **Surname** and an arbitrary regular XML statement in the `::inputRequestPart::0::0::1::0::0::1::0::0::4` box, as shown in Figure A-6.

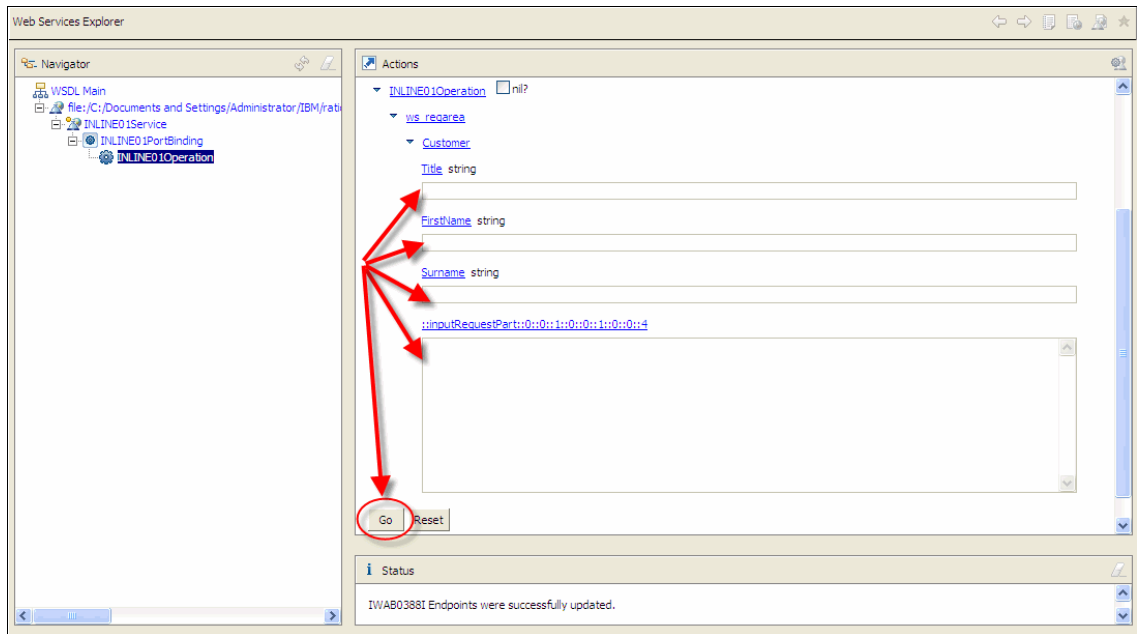


Figure A-6 XML any passthrough web service data input

4. Click **Go** and check for the results in the window below. The XML statement is delivered without disturbing its own SOAP envelope, as shown in Figure A-7.

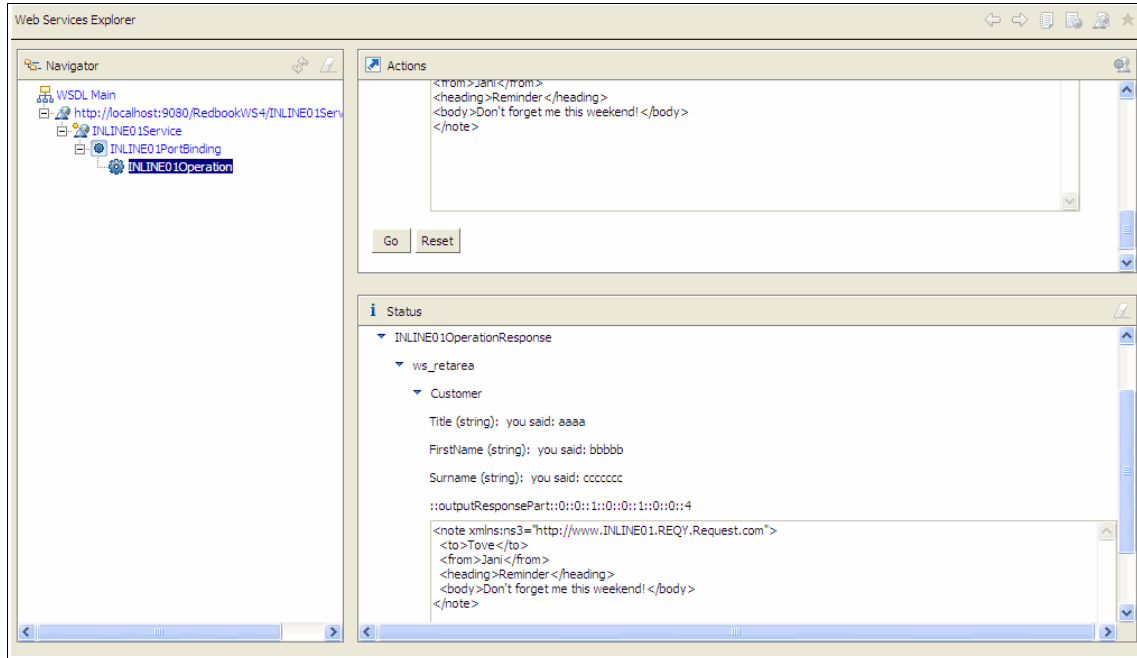


Figure A-7 XML any passthrough web service test result

The XML choice Web service example

This example of a Web service shows how you can deal with the XML choice statement when using SOAP messages for data transfer. The program simply interchanges the users' choice, returning the other possibility. You can install the file by following the steps in “Preparation of your RDz environment” on page 290 and “Loading an .ear file into a new or existing project” on page 291.

Coding

The code given in Example A-2 is, for the most part, automatically derived from the *.wsdl file. You only have to add the problems' solution to the Java skeleton.

Example A-2 Code of the XML choice web service example

```
package com.requ.inline01;

@javax.jws.WebService
(endpointInterface="com.requ.inline01.INLINE01Port",
targetNamespace="http://www.INLINE01.REQY.com",
serviceName="INLINE01Service", portName="INLINE01Port")
public class INLINE01HTTPSoapBindingImpl{

public com.response.respy.inline01.ProgramInterface
inline01Operation(com.request.requ.inline01.ProgramInterface
requestPart) {
// Here starts the actual problem solution
// Allocating references for the exchange
com.response.respy.inline01.ProgramInterface respreference = new
com.response.respy.inline01.ProgramInterface();
com.response.respy.inline01.ProgramInterface.WsRetarea respwsarea = new
com.response.respy.inline01.ProgramInterface.WsRetarea();
com.response.respy.inline01.ProgramInterface.WsRetarea.ChoiceData
respchoice = new
com.response.respy.inline01.ProgramInterface.WsRetarea.ChoiceData();
com.request.requ.inline01.ProgramInterface reqreference =new
com.request.requ.inline01.ProgramInterface();
com.request.requ.inline01.ProgramInterface.WsReqarea reqwsarea = new
com.request.requ.inline01.ProgramInterface.WsReqarea();
com.request.requ.inline01.ProgramInterface.WsReqarea.ChoiceData
reqchoice = new
com.request.requ.inline01.ProgramInterface.WsReqarea.ChoiceData();
// Processing the input
reqwsarea = requestPart.getWsReqarea();
reqchoice = reqwsarea.getChoiceData();
// Changing the users' choice
respchoice.setFirstchoice(reqchoice.getSecondchoice());
respchoice.setSecondchoice(reqchoice.getFirstchoice());
// Composition of the answer
respwsarea.setChoiceData(respchoice);
respreference.setWsRetarea(respwsarea);return respreference;
}
}
```

Testing

With the IBM Rational Developer for System z, you can test the function of every .wsdl file using the web services explorer. For this example, follow these steps:

1. Open the Services subdirectory in your project's folder, then right-click **INLINE01Service**, and choose **Test with Web Services Explorer**, as shown in Figure A-8.

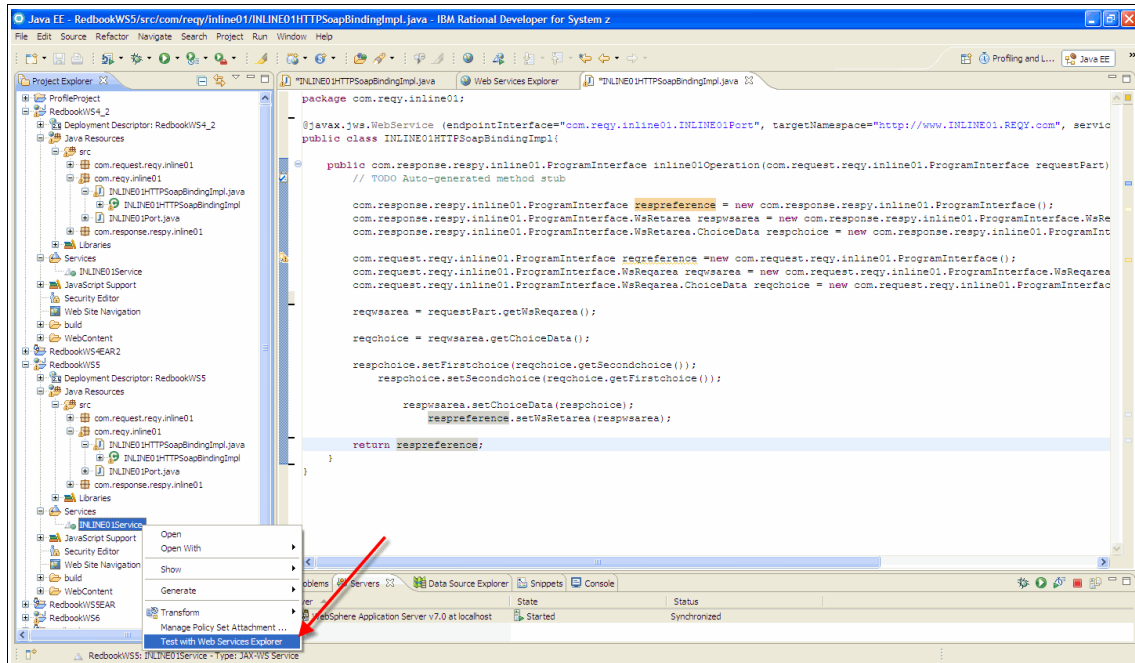


Figure A-8 Location of the Web Services Explorer in RDZ

2. Maximize the new window, **Web Services Explorer**, by double-clicking its tab.
3. Confirm the standard endpoint and click **Go**.

- Invoke a new WSDL Operation. Mark either the **firstchoice** or **secondchoice** checkbox before the **values** field, as shown in Figure A-9.

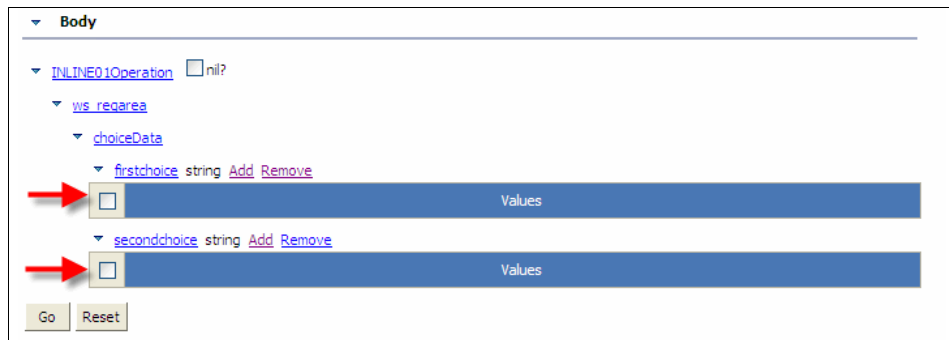


Figure A-9 XML choice web service data input 1/3

- Click **Add** in the corresponding definition of your choice as shown in Figure A-10.

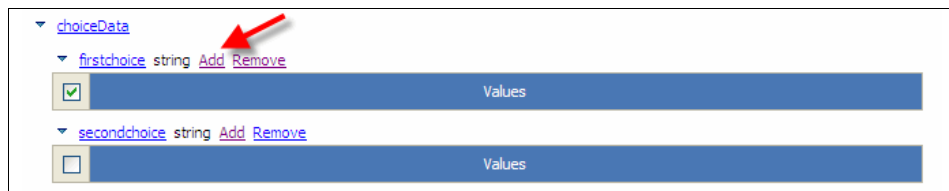


Figure A-10 XML choice web service data input 2/3

- Type in the value of your choice in the new line and select its checkbox. Assure that the second value is not empty, so add a line there, too. The windows should now look like Figure A-11.

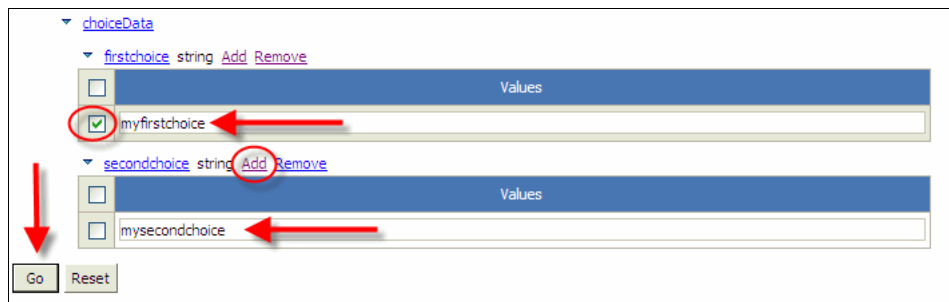


Figure A-11 XML choice web service data input 3/3

7. Click **Go** and check for the results in the window below. The choices are interchanged, as shown in Figure A-12.

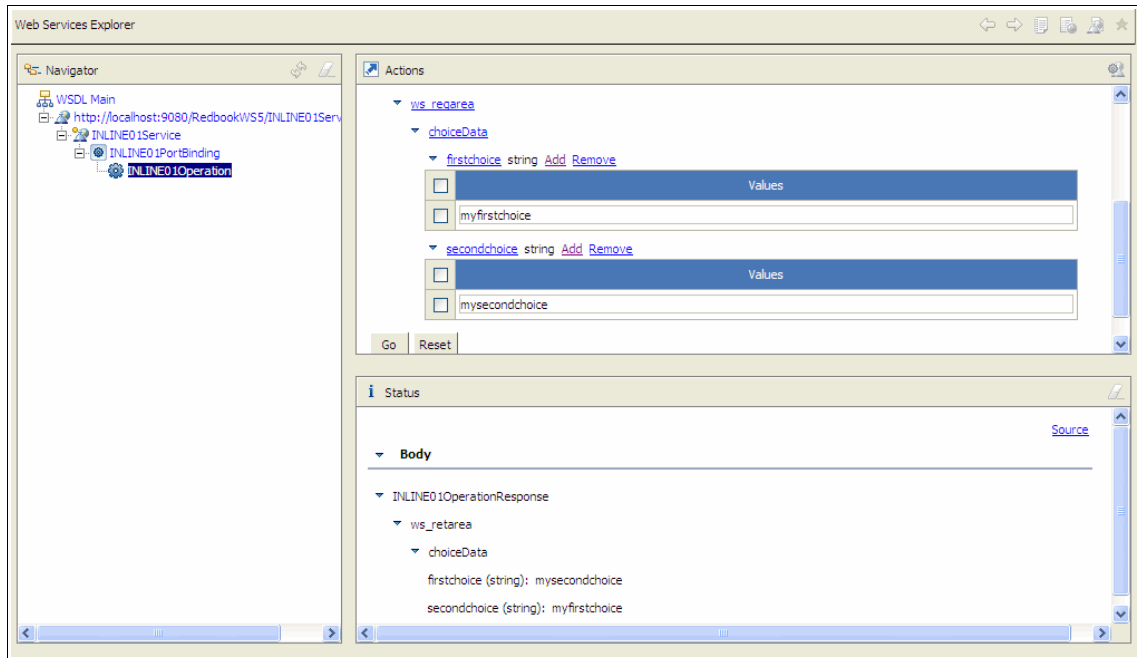


Figure A-12 XML choice web service test result

The XML occurs Web service example

This Web service demonstrates how you can deal with the both XML minOccurs- and maxoccurs-statements if you are using SOAP messages to transfer variable arrays between a modern and an old (non-dynamic, COBOL) program. It returns the data structure references by the XML statement and add two additional instances to the array, these are added beneath the maxoccurs border.

Coding

The code given in Example A-3 is, for the most part, automatically derived from the *.wsdl file. You only have to add the problems' solution to the Java skeleton.

Example A-3 Code of the XML occurs web service example

```
package com.reqy.inline01;

import java.util.ArrayList;
import java.util.List;
import com.request.reqy.inline01.ProgramInterface;

@javax.jws.WebService
(endpointInterface="com.reqy.inline01.INLINE01Port",
targetNamespace="http://www.INLINE01.REQY.com",
serviceName="INLINE01Service", portName="INLINE01Port")
public class INLINE01HTTPSoapBindingImpl{

public com.response.respy.inline01.ProgramInterface
inline01Operation(com.request.reqy.inline01.ProgramInterface
requestPart) {
// Here starts the actual problem solution
// Allocating space for reference
com.response.respy.inline01.ProgramInterface resp = new
com.response.respy.inline01.ProgramInterface();
com.response.respy.inline01.ProgramInterface.WsRetarea respwsretarea =
new com.response.respy.inline01.ProgramInterface.WsRetarea();
com.response.respy.inline01.ProgramInterface.WsRetarea.Recs resprescs =
new com.response.respy.inline01.ProgramInterface.WsRetarea.Recs();
com.request.reqy.inline01.ProgramInterface.WsReqarea reqwsarea = new
com.request.reqy.inline01.ProgramInterface.WsReqarea();
// Setting occur-borders
int location; int maxlocation;
System.out.println("Occurs Test...");
// Processing input data
reqwsarea = requestPart.getWsReqarea();
// Assure that field size per construct is correct
maxlocation = reqwsarea.getRecs().size(); maxlocation = maxlocation +2;
// Build new constructs
for (location = 0; location < maxlocation;)
{
resprescs = new
com.response.respy.inline01.ProgramInterface.WsRetarea.Recs();
resprescs.setRecs("string array number " +location);
respwsretarea.getRecs().add(location, resprescs);
}
```



```

location++;
}
esp.setWsRetarea(respwsretarea);
return resp;
}

```

Testing

With the IBM Rational Developer for System z, you can test the function of every .wsdl file using the web services explorer. For this example, follow these steps:

1. Open the Services subdirectory in your project's folder, then right-click **INLINE01Service**, and choose **Test with Web Services Explorer**. See Figure A-13.

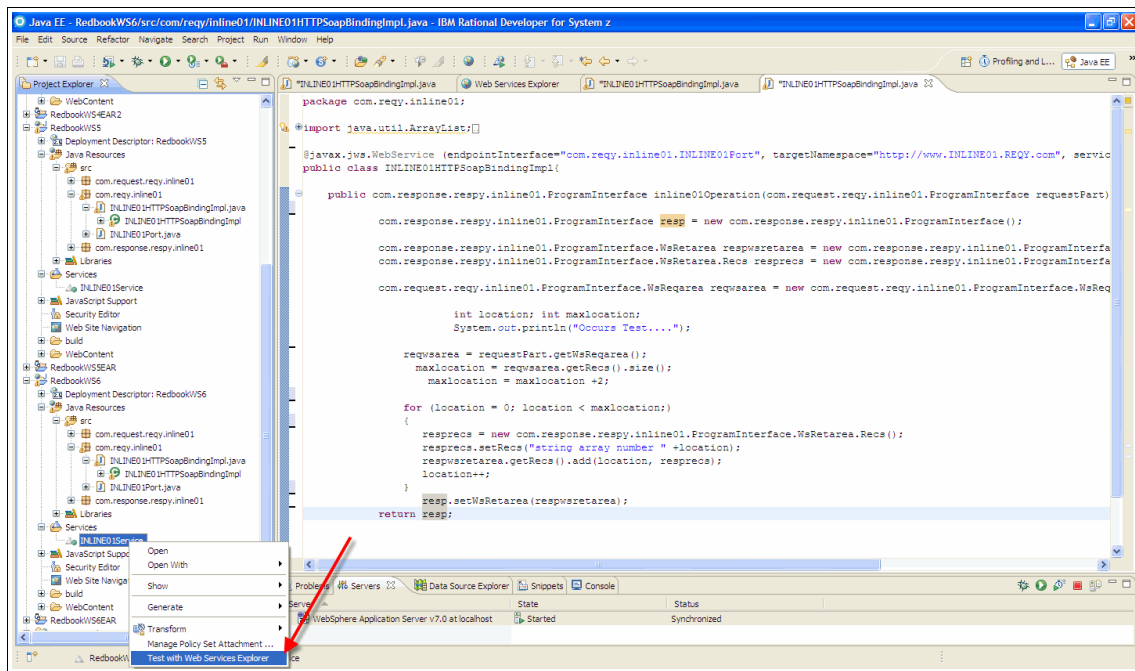


Figure A-13 Location of the Web Services Explorer in RDZ

2. Maximize the new window, the **Web Services Explorer**, by double-clicking on its tab.
3. Confirm the standard endpoint and click **Go**.
4. Invoke a new WSDL operation. Add as many as content boxes you want by clicking **Add** and check them all by selecting the **Content** cell check box.
5. Type in random values for your boxes. Your window should look like Figure A-14.

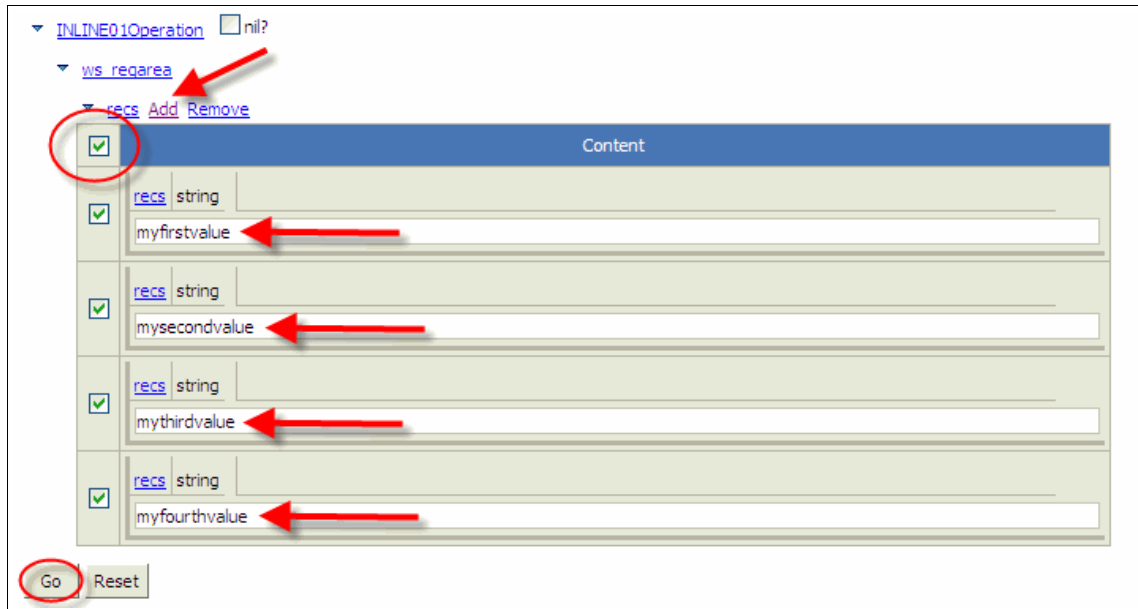


Figure A-14 XML accurs web service data input

- Click **Go** and look for the results in the window below. You receive the same data constructs (that is, the boxes), but additionally, two dynamically added extra constructs, as shown in Figure A-15.

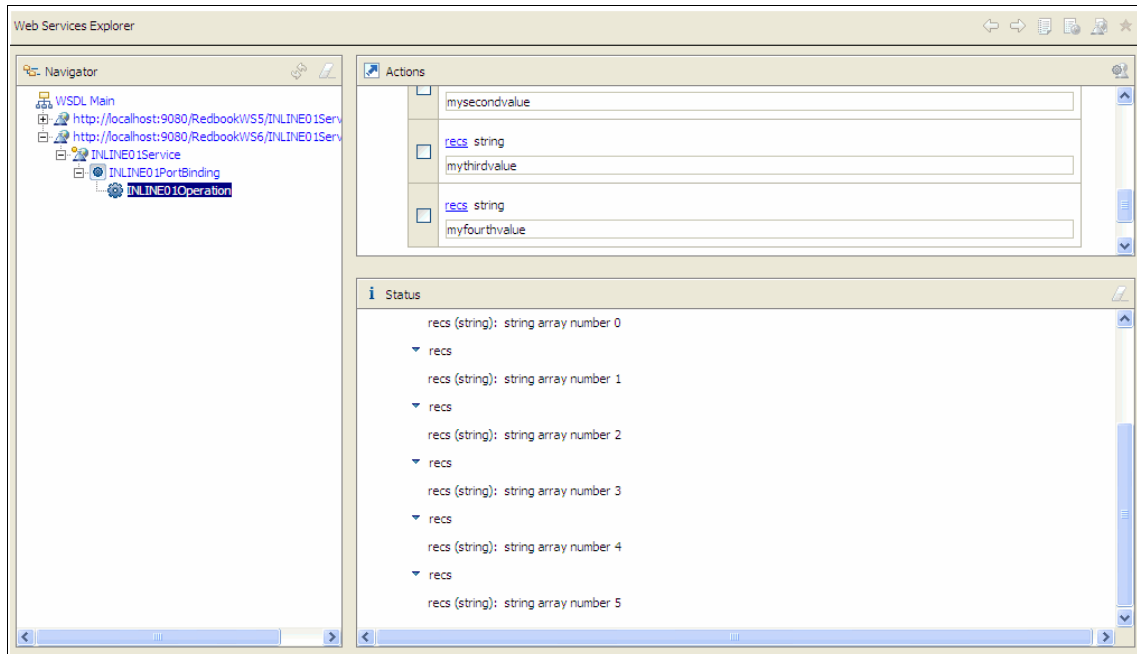


Figure A-15 XML occurs web service test result



B

Sample COBOL programs

In this appendix we list the COBOL programs used to call the sampleweb services listed in Appendix A, “Sample Web services” on page 289.

We also list the WSDL that was used provided by the service provider, which we used to generate the COBOL copybooks and the skeleton COBOL programs.

Program to call <xsd:any> example service

```
PROCESS CICS,NODYNAM,NSYMBOL(NATIONAL),TRUNC(STD)
* *****
* *****RDz**7.5*****
* *****
*           New CICS TS 3.x Web Service Requester
* *****RDz**7.5*****
*           PROCESS CICS,NODYNAM,NSYMBOL(NATIONAL),TRUNC(STD)
* *****
* *****RDz**7.5*****
* *****
*           New CICS TS 3.x Web Service Requester
* *****RDz**7.5*****
* *****

IDENTIFICATION DIVISION.
*Begin Identification Division
  PROGRAM-ID. 'INLINETS'.
  AUTHOR. WD4Z.
  INSTALLATION. 9.1.200.V200903111338.
  DATE-WRITTEN. 09/09/09 15:16.
*End Identification Division

DATA DIVISION.
*Begin Data Division
  WORKING-STORAGE SECTION.
*Begin Working-Storage Section
* *****
*           Operations Available On The Remote Web Service
* *****
  1 OPERATION-NAME-1.
  2 PIC X(17) USAGE DISPLAY
    VALUE 'INLINE01Operation'.
*End Working-Storage Section

LOCAL-STORAGE SECTION.
*Begin Local-Storage Section
* *****
*           Program Work Variables
* *****
  1 SOAP-PIPELINE-WORK-VARIABLES.
  2 WS-WEBSERVICE-NAME PIC X(32).
  2 WS-OPERATION-NAME PIC X(255).
  2 WS-CONTAINER-NAME PIC X(16).
```

```

2 WS-CHANNEL-NAME    PIC X(16).
2 COMMAND-RESP       PIC S9(8) COMP.
2 COMMAND-RESP2      PIC S9(8) COMP.
*Specify A URI To Override The Web Service Description
1 URI-RECORD-STRUCTURE.
2 FILLER              PIC X(10).
2 WS-URI-OVERRIDE     PIC X(255).

1 WS-XML-PASSTHRU-DATA.
2 WS-CUST-XML         PIC X(255).
2 WS-CUST-XMLns      PIC X(255).

1 WS-DFHWS-BODY      PIC x(400).

* *****
*                               Language Structures
* *****

1 LANG-INLINI01.
  COPY inlinI01.
1 LANG-INLIN001.
  COPY inlin001.
*End Local-Storage Section
LINKAGE SECTION.
*Begin Linkage Section
*End Linkage Section
*End Data Division
PROCEDURE DIVISION
.
*Begin Procedure Division
MAINLINE SECTION.
* -----
*                               Initialize Work Variables
* -----
  INITIALIZE SOAP-PIPELINE-WORK-VARIABLES.
  INITIALIZE URI-RECORD-STRUCTURE.
* -----
* Container DFHWS-DATA must be present when a service requeste
* r program issues an EXEC CICS INVOKE WEBSERVICE command. Whe
* n the command is issued, CICS converts the language structur
* e that is in the container into a SOAP request. When the soa
* p response is received, CICS converts it into another langua
* ge structure that is returned to the application in the same
* container.
* -----
  MOVE 'DFHWS-DATA'

```

```

        TO WS-CONTAINER-NAME
* -----
*           Channel Passed To The Web Service Call
* -----
        MOVE 'SERVICE-CHANNEL'
          TO WS-CHANNEL-NAME
* -----
*           WEBSERVICE resource installed in this CICS region
* -----
        MOVE 'inlinetst'
          TO WS-WEBSERVICE-NAME
* -----
*           Operation To Invoke On The Remote Web Service
* -----
        MOVE OPERATION-NAME-1
          TO WS-OPERATION-NAME
* -----
*           Populate Request Language Structure
* -----
        INITIALIZE LANG-INLINIO1

        Move 'MR' TO XTitle of wsXreqarea
        Move 2 to XTitle-length of wsXreqarea

        Move 'Tony' TO FirstName of wsXreqarea
        Move 4 to FirstName-length of wsXreqarea

        Move 'Fitzgerald' TO Surname of wsXreqarea
        MOVE 10 to Surname-length of wsXreqarea

        INITIALIZE WS-XML-PASSTHRU-DATA
* -----
*           Put the "any" XML data into the channel
* -----

        Move 1           to Customer-num       of wsXreqarea
        MOVE 'cust-xml-cont' TO Customer-xml-cont of wsXreqarea

*           --- the XML ---
        Move '<Whatever>.....</Whatever>' to WS-CUST-XML

        EXEC CICS PUT CONTAINER(Customer-xml-cont of wsXreqarea)

```



```

        CHANNEL(WS-CHANNEL-NAME)
        FROM(WS-CUST-XML)
        DATATYPE(DFHVALUE(CHAR))
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

* -----
*   Put the "any" XMLns data into the channel
* -----
        MOVE 'cust-xmlns-cont'          to Customer-xmlns-cont
                                           of wsXreqarea
*   Move 'xmlns:ns1="http://myNS"' to WS-CUST-XMLns

* -----
*   Put Request Language Structure Into SOAP Container
* -----

    EXEC CICS PUT CONTAINER(WS-CONTAINER-NAME)
        CHANNEL(WS-CHANNEL-NAME)
        FROM(LANG-INLINIO1)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

* -----
*                               Invoke The Web Service
* -----
        Move 'http://9.173.198.188:9080/RedbookWS4/INLINE01Service'
            to WS-URI-OVERRIDE

    EXEC CICS INVOKE SERVICE(WS-WEBSERVICE-NAME)
        CHANNEL(WS-CHANNEL-NAME)
        URI(WS-URI-OVERRIDE)
        OPERATION(WS-OPERATION-NAME)
        RESP(COMMAND-RESP) RESP2(COMMAND-RESP2)
    END-EXEC
    PERFORM CHECK-WEBSERVICE-COMMAND

* -----
*                               Receive Response Language Structure
* -----

    EXEC CICS GET CONTAINER(WS-CONTAINER-NAME)
        CHANNEL(WS-CHANNEL-NAME)
        INTO(LANG-INLINO01)
    END-EXEC

```

PERFORM CHECK-CONTAINER-COMMAND

```
* -----
*           "Process" the Response Language Structure
* -----
  DISPLAY 'XTitle data returned   = ' XTitle of wsXretarea
  DISPLAY 'FirstName data returned = ' FirstName of wsXretarea
  DISPLAY 'Surname data returned  = ' Surname of wsXretarea

  INITIALIZE WS-XML-PASSTHRU-DATA.

  EXEC CICS GET CONTAINER(Customer-xmlns-cont of wsXretarea)
    CHANNEL(WS-CHANNEL-NAME)
    INTO(WS-CUST-XMLns)
  END-EXEC
  PERFORM CHECK-CONTAINER-COMMAND

  EXEC CICS GET CONTAINER(Customer-xml-cont of wsXretarea)
    CHANNEL(WS-CHANNEL-NAME)
    INTO(WS-CUST-XML)
  END-EXEC
  PERFORM CHECK-CONTAINER-COMMAND

  DISPLAY 'Customer-xml-cont data = ' WS-CUST-XML
  DISPLAY 'Customer-xmlns-cont data = ' WS-CUST-XMLns

* -----
*                               Finished
* -----
  EXEC CICS RETURN
  END-EXEC
  .

CHECK-CONTAINER-COMMAND.
  EVALUATE COMMAND-RESP
    WHEN DFHRESP(CCSIDERR)
      EXEC CICS ABEND ABCODE('C001') END-EXEC
      CONTINUE
    WHEN DFHRESP(CONTAINERERR)
      EXEC CICS ABEND ABCODE('C002') END-EXEC
      CONTINUE
    WHEN DFHRESP(INVREQ)
      EXEC CICS ABEND ABCODE('C003') END-EXEC
      CONTINUE
    WHEN DFHRESP(LENGERR)
```

```

        EXEC CICS ABEND ABCODE('C004') END-EXEC
        CONTINUE
    END-EVALUATE
    .

CHECK-WEBSERVICE-COMMAND.
    EVALUATE COMMAND-RESP
        WHEN DFHRESP(INVREQ)
            PERFORM INVREQ-PROCESSING
            EXEC CICS ABEND ABCODE('WS01') END-EXEC
            CONTINUE
        WHEN DFHRESP(NOTFND)
            EXEC CICS ABEND ABCODE('WS02') END-EXEC
            CONTINUE
    END-EVALUATE
    .

INVREQ-PROCESSING.
    IF EIBRESP2 = 6 THEN
*   ** An EIBRESP2 of 6 indicates a SOAP fault **
*   ** has been returned in DFHWS-BODY      **
        EXEC CICS
            GET CONTAINER('DFHWS-BODY')
            CHANNEL(WS-CHANNEL-NAME)
            INTO(WS-DFHWS-BODY)
        END-EXEC
        DISPLAY WS-DFHWS-BODY
    END-IF
    .

*End Procedure Division
END PROGRAM 'INLINETS'.

```

WSDL - <xsd:any>

```
<?xml version="1.0"?>
<!--This document was generated using 'DFHLS2WS' at mapping level '2.2'. -->
<definitions targetNamespace="http://www.INLINE01.REQY.com"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:reqns="http://www.INLINE01.REQY.Request.com"
xmlns:resns="http://www.INLINE01.RESPY.Response.com"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.INLINE01.REQY.com">
  <types>
    <xsd:schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://www.INLINE01.REQY.Request.com"
xmlns:tns="http://www.INLINE01.REQY.Request.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:annotation>
        <xsd:documentation source="http://www.ibm.com/software/htp/cics/annotations">
          This schema was generated by the CICS Web services
          assistant.</xsd:documentation>
        </xsd:annotation>
        <xsd:annotation>
          <xsd:appinfo source="http://www.ibm.com/software/htp/cics/annotations">
            com.ibm.cics.wsdl.properties.mappingLevel=2.2</xsd:appinfo>
          </xsd:annotation>
          <xsd:complexType abstract="false" block="#all" final="#all"
mixed="false" name="ProgramInterface">
            <xsd:sequence>
              <xsd:element name="ws_reqarea" nillable="false">
                <xsd:complexType mixed="false">
                  <xsd:sequence>
                    <xsd:element name="Customer">
                      <xsd:complexType>
                        <xsd:sequence>
                          <xsd:element name="Title"
type="xsd:string" />
                          <xsd:element name="FirstName"
type="xsd:string" />
                          <xsd:element name="Surname"
type="xsd:string" />
                          <xsd:any minOccurs="0" />
                        </xsd:sequence>
                      </xsd:complexType>
                    </xsd:element>
```

```

        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="INLINE01Operation" nillable="false"
type="tns:ProgramInterface" />
</xsd:schema>
<xsd:schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://www.INLINE01.RESPY.Response.com"
xmlns:tns="http://www.INLINE01.RESPY.Response.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation source="http://www.ibm.com/software/htp/cics/annotations">
            This schema was generated by the CICS Web services
            assistant.</xsd:documentation>
        </xsd:annotation>
    <xsd:annotation>
        <xsd:appinfo source="http://www.ibm.com/software/htp/cics/annotations">
            com.ibm.cics.wsd1.properties.mappingLevel=2.2</xsd:appinfo>
        </xsd:annotation>
    <xsd:complexType abstract="false" block="#all" final="#all"
mixed="false" name="ProgramInterface">
        <xsd:sequence>
            <xsd:element name="ws_retarea" nillable="false">
                <xsd:complexType mixed="false">
                    <xsd:sequence>
                        <xsd:element name="Customer">
                            <xsd:complexType>
                                <xsd:sequence>
                                    <xsd:element name="Title"
type="xsd:string" />
                                    <xsd:element name="FirstName"
type="xsd:string" />
                                    <xsd:element name="Surname"
type="xsd:string" />
                                    <xsd:any minOccurs="0" />
                                </xsd:sequence>
                            </xsd:complexType>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:schema>

```

```

        </xsd:complexType>
        <xsd:element name="INLINE01operationResponse"
            nillable="false" type="tns:ProgramInterface" />
    </xsd:schema>
</types>
<message name="INLINE01operationResponse">
    <part element="resns:INLINE01operationResponse"
        name="ResponsePart" />
</message>
<message name="INLINE01operationRequest">
    <part element="reqns:INLINE01operation" name="RequestPart" />
</message>
<portType name="INLINE01Port">
    <operation name="INLINE01operation">
        <input message="tns:INLINE01operationRequest"
            name="INLINE01operationRequest" />
        <output message="tns:INLINE01operationResponse"
            name="INLINE01operationResponse" />
    </operation>
</portType>
<binding name="INLINE01HTTPSoapBinding" type="tns:INLINE01Port">
    <!-- This soap:binding indicates the use of SOAP 1.1 -->
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="INLINE01operation">
        <soap:operation soapAction="" style="document" />
        <input name="INLINE01operationRequest">
            <soap:body parts="RequestPart" use="literal" />
        </input>
        <output name="INLINE01operationResponse">
            <soap:body parts="ResponsePart" use="literal" />
        </output>
    </operation>
</binding>
<service name="INLINE01Service">
    <port binding="tns:INLINE01HTTPSoapBinding"
        name="INLINE01Port">
        <!-- This soap:address indicates the location of the Web service over HTTP.
            Please replace "my-server" with the TCPIP host name of your CICS
region.
            Please replace "my-port" with the port number of your CICS
TCPIP SERVICE. -->
        <soap:address location="http://my-server:my-port/inline/test" />
        <!-- This soap:address indicates the location of the Web service over HTTPS.
-->

```

```
      <!-- <soap:address location="https://my-server:my-port/inline/test"/> -->
      <!-- This soap:address indicates the location of the Web service over WebSphere
MQSeries.
           Please replace "my-queue" with the appropriate queue name. -->
      <!-- <soap:address
location="jms:/queue?destination=my-queue&connectionFactory=()&targetService=
/inline/test&initialContextFactory=com.ibm.mq.jms.NoJndi" /> -->
      </port>
    </service>
  </definitions>
```

Request Language Structure - inlinl01

```
* ++++++
* This file contains the generated request language structure(s)
* for WSDL operation 'INLINE01operation'.
* The response message for this WSDL Operation may be replaced
* with a SOAP Fault message.
* This structure was generated using 'DFHWS2LS' at mapping level
* '2.1'.
*
*
*      03 INLINE01operation.
*          06 wsXreqarea.
*              09 Customer1.
*
* Comments for field 'XTitle':
* This field represents the value of XML element
* '/INLINE01operation/ws_reqarea/Customer/Title'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
*      12 XTitle-length          PIC S9999 COMP-5
* SYNC.
*      12 XTitle                 PIC X(255).
*
* Comments for field 'FirstName':
* This field represents the value of XML element
* '/INLINE01operation/ws_reqarea/Customer/FirstName'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
*      12 FirstName-length      PIC S9999 COMP-5
* SYNC.
*      12 FirstName            PIC X(255).
*
* Comments for field 'Surname':
* This field represents the value of XML element
* '/INLINE01operation/ws_reqarea/Customer/Surname'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
```



```

*          12 Surname-length          PIC S9999 COMP-5
* SYNC.
*          12 Surname                  PIC X(255).
*
*
* Array 'Customer' contains a variable number of instances of
* XML element
* '/INLINE010operation/ws_reqarea/Customer/Customer'. The number
* of instances present is indicated in field 'Customer-num'.
* There should be at least '0' instance(s).
* There should be at most '1' instance(s).
*          12 Customer-num            PIC S9(9) COMP-5
* SYNC.
*
*
*          12 Customer.
*
* Comments for field 'Customer-xml-cont':
* XML data type: 'any'.
* This field contains the name of a CONTAINER which in turn
* holds the XML data for an xsd:any or xsd:anyType. The
* CONTAINER must be read from and written to in CHAR mode.
*          15 Customer-xml-cont      PIC X(16).
*
* Comments for field 'Customer-xmlns-cont':
* XML data type: 'any'.
* This field contains the name of a CONTAINER which in turn
* holds namespace prefix definitions that may be used in the
* XML. The CONTAINER must be read from in CHAR mode.
*          15 Customer-xmlns-cont   PIC X(16).
*
*
* ++++++

```

```

03 INLINE010operation.
  06 wsXreqarea.
    09 Customer1.
      12 XTitle-length          PIC S9999 COMP-5
    SYNC.
      12 XTitle                  PIC X(255).
      12 FirstName-length       PIC S9999 COMP-5
    SYNC.
      12 FirstName              PIC X(255).
      12 Surname-length         PIC S9999 COMP-5
    SYNC.

```

```
12 Surname PIC X(255).  
12 Customer-num PIC S9(9) COMP-5  
SYNC.  
12 Customer.  
15 Customer-xml-cont PIC X(16).  
15 Customer-xmlns-cont PIC X(16).
```

Response Language Structure - inlinO01

```
* ++++++
* This file contains the generated response language
* structure(s) for WSDL operation 'INLINE010operation'.
* The response message for this WSDL Operation may be replaced
* with a SOAP Fault message.
* This structure was generated using 'DFHWS2LS' at mapping level
* '2.1'.
*
*
*      03 INLINE010operationResponse.
*          06 wsXretarea.
*              09 Customer1.
*
* Comments for field 'XTitle':
* This field represents the value of XML element
* '/INLINE010operationResponse/ws_retarea/Customer/Title'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
*      12 XTitle-length          PIC S9999 COMP-5
* SYNC.
*      12 XTitle                PIC X(255).
*
* Comments for field 'FirstName':
* This field represents the value of XML element
* '/INLINE010operationResponse/ws_retarea/Customer/FirstName'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
*      12 FirstName-length      PIC S9999 COMP-5
* SYNC.
*      12 FirstName            PIC X(255).
*
* Comments for field 'Surname':
* This field represents the value of XML element
* '/INLINE010operationResponse/ws_retarea/Customer/Surname'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
```

```

*          12 Surname-length          PIC S9999 COMP-5
* SYNC.
*          12 Surname                  PIC X(255).
*
*
* Array 'Customer' contains a variable number of instances of
* XML element
* '/INLINE01OperationResponse/ws_retarea/Customer/Customer'.
* The number of instances present is indicated in field
* 'Customer-num'.
* There should be at least '0' instance(s).
* There should be at most '1' instance(s).
*          12 Customer-num            PIC S9(9) COMP-5
* SYNC.
*
*
*          12 Customer.
*
* Comments for field 'Customer-xml-cont':
* XML data type: 'any'.
* This field contains the name of a CONTAINER which in turn
* holds the XML data for an xsd:any or xsd:anyType. The
* CONTAINER must be read from and written to in CHAR mode.
*          15 Customer-xml-cont       PIC X(16).
*
* Comments for field 'Customer-xmlns-cont':
* XML data type: 'any'.
* This field contains the name of a CONTAINER which in turn
* holds namespace prefix definitions that may be used in the
* XML. The CONTAINER must be read from in CHAR mode.
*          15 Customer-xmlns-cont     PIC X(16).
*
*
* ++++++
03 INLINE01OperationResponse.
  06 wsXretarea.
    09 Customer1.
      12 XTitle-length              PIC S9999 COMP-5
    SYNC.
      12 XTitle                    PIC X(255).
      12 FirstName-length          PIC S9999 COMP-5
    SYNC.
      12 FirstName                  PIC X(255).
      12 Surname-length            PIC S9999 COMP-5

```

```
SYNC.  
  12 Surname PIC X(255).  
  
  12 Customer-num PIC S9(9) COMP-5  
SYNC.  
  
  12 Customer.  
    15 Customer-xml-cont PIC X(16).  
    15 Customer-xmlns-cont PIC X(16).
```

Program to call <xsd:choice> example service

```
PROCESS CICS,NODYNAM,NSYMBOL(NATIONAL),TRUNC(STD)
* *****
* *****RDz**7.5*****
* *****
*           New CICS TS 3.x Web Service Requester
* *****RDz**7.5*****
* *****

IDENTIFICATION DIVISION.
*Begin Identification Divsion
  PROGRAM-ID. 'CHOICETE'.
  AUTHOR. WD4Z.
  INSTALLATION. 9.1.200.V200903111338.
  DATE-WRITTEN. 21/09/09 11:45.
*End Identification Divsion
DATA DIVISION.
*Begin Data Divsion
WORKING-STORAGE SECTION.
*Begin Working-Storage Section
* *****
*           Operations Available On The Remote Web Service
* *****
  1 OPERATION-NAME-1.
  2 PIC X(17) USAGE DISPLAY
    VALUE 'INLINE01operation'.
*End Working-Storage Section
LOCAL-STORAGE SECTION.
*Begin Local-Storage Section
* *****
*           Program Work Variables
* *****
  1 SOAP-PIPELINE-WORK-VARIABLES.
  2 WS-WEBSERVICE-NAME PIC X(32).
  2 WS-OPERATION-NAME PIC X(255).
  2 WS-CONTAINER-NAME PIC X(16).
  2 WS-CHANNEL-NAME PIC X(16).
  2 COMMAND-RESP PIC S9(8) COMP.
  2 COMMAND-RESP2 PIC S9(8) COMP.
*Specify A URI To Override The Web Service Description
  1 URI-RECORD-STRUCTURE.
  2 FILLER PIC X(10).
  2 WS-URI-OVERRIDE PIC X(255).
```

```

1 WS-DFHWS-BODY      PIC x(400).
* *****
*                               Language Structures
* *****
1 LANG-CHOICI01.
  COPY choicI01.
1 LANG-CHOIC001.
  COPY choic001.
*End Local-Storage Section
LINKAGE SECTION.
*Begin Linkage Section
*End Linkage Section
*End Data Division
PROCEDURE DIVISION
.
*Begin Procedure Division
MAINLINE SECTION.
* -----
*                               Initialize Work Variables
* -----
  INITIALIZE SOAP-PIPELINE-WORK-VARIABLES.
  INITIALIZE URI-RECORD-STRUCTURE.
* -----
* Container DFHWS-DATA must be present when a service requeste
* r program issues an EXEC CICS INVOKE WEBSERVICE command. Whe
* n the command is issued, CICS converts the language structur
* e that is in the container into a SOAP request. When the soa
* p response is received, CICS converts it into another langua
* ge structure that is returned to the application in the same
* container.
* -----
  MOVE 'DFHWS-DATA'
  TO WS-CONTAINER-NAME
* -----
*                               Channel Passed To The Web Service Call
* -----
  MOVE 'SERVICE-CHANNEL'
  TO WS-CHANNEL-NAME
* -----
*                               WEBSERVICE resource installed in this CICS region
* -----
  MOVE 'choicetest'
  TO WS-WEBSERVICE-NAME
* -----

```

```

*           Operation To Invoke On The Remote Web Service
* -----
*   MOVE OPERATION-NAME-1
*     TO WS-OPERATION-NAME
* -----
*           Populate Request Language Structure
* -----
*   INITIALIZE LANG-CHOICI01
*
*   DISPLAY 'data is being sent in the firstchoice field'
*
* The WSDL specifies that only one of the two fields can
* be sent to the service
*   EITHER firstchoice or secondchoice
*
*   move 'first choice data' to firstchoice
*     of choicI01-firstchoice
*   move 18 to firstchoice-length
*     of choicI01-firstchoice
*
*   DISPLAY 'data to be sent is ==>' firstchoice
*     of choicI01-firstchoice
*
*   set firstchoice of wsXreqarea to true
*
*   move 'CHOICE-CONT' to choiceData-cont of wsXreqarea
*
*   EXEC CICS PUT CONTAINER(choiceData-cont of wsXreqarea)
*     CHANNEL(WS-CHANNEL-NAME)
*     FROM(choicI01-firstchoice)
*   END-EXEC
*   PERFORM CHECK-CONTAINER-COMMAND
*
* -----
*           Put Request Language Structure Into SOAP Container
* -----
*   EXEC CICS PUT CONTAINER(WS-CONTAINER-NAME)
*     CHANNEL(WS-CHANNEL-NAME)
*     FROM(LANG-CHOICI01)
*   END-EXEC
*   PERFORM CHECK-CONTAINER-COMMAND
*
* -----
*           Invoke The Web Service
* -----
*   Move 'http://9.146.153.15:9080/RedbookWS5/INLINE01Service'

```



```

        to WS-URI-OVERRIDE

EXEC CICS INVOKE WEBSERVICE(WS-WEBSERVICE-NAME)
      CHANNEL(WS-CHANNEL-NAME)
      URI(WS-URI-OVERRIDE)
      OPERATION(WS-OPERATION-NAME)
      RESP(COMMAND-RESP) RESP2(COMMAND-RESP2)
END-EXEC
PERFORM CHECK-WEBSERVICE-COMMAND
* -----
*           Receive Response Language Structure
* -----
EXEC CICS GET CONTAINER(WS-CONTAINER-NAME)
      CHANNEL(WS-CHANNEL-NAME)
      INTO(LANG-CHOIC001)
END-EXEC
PERFORM CHECK-CONTAINER-COMMAND
* -----
*           Process Response Language Structure
* -----
*
* Check which of the "choice" fields have been returned
* and process the result
*
EVALUATE TRUE
  when empty of wsXretarea
    display 'nothing returned'

  when firstchoice of wsXretarea
    display 'data was returned in the firstchoice field'

    EXEC CICS GET CONTAINER(choiceData-cont of wsXretarea)
      CHANNEL(WS-CHANNEL-NAME)
      INTO(choic001-firstchoice)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

    display 'data returned is ==>'
      firstchoice of choic001-firstchoice

  when secondchoice of wsXretarea
    display 'data was returned in the secondchoice field'

    EXEC CICS GET CONTAINER(choiceData-cont of wsXretarea)
      CHANNEL(WS-CHANNEL-NAME)

```

```

        INTO(choic001-secondchoice)
        END-EXEC
        PERFORM CHECK-CONTAINER-COMMAND

        display 'data returned is ==>'
            secondchoice of choic001-secondchoice

    END-EVALUATE

* -----
*                               Finished
* -----

    EXEC CICS RETURN
    END-EXEC
.
CHECK-CONTAINER-COMMAND.
    EVALUATE COMMAND-RESP
        WHEN DFHRESP(CCSIDERR)
            EXEC CICS ABEND ABCODE('C001') END-EXEC
            CONTINUE
        WHEN DFHRESP(CONTAINERERR)
            EXEC CICS ABEND ABCODE('C002') END-EXEC
            CONTINUE
        WHEN DFHRESP(INVREQ)
            EXEC CICS ABEND ABCODE('C002') END-EXEC
            CONTINUE
        WHEN DFHRESP(LENGERR)
            EXEC CICS ABEND ABCODE('C002') END-EXEC
            CONTINUE
    END-EVALUATE
.
CHECK-WEBSERVICE-COMMAND.
    EVALUATE COMMAND-RESP
        WHEN DFHRESP(INVREQ)
            PERFORM INVREQ-PROCESSING
            EXEC CICS ABEND ABCODE('W001') END-EXEC
            CONTINUE
        WHEN DFHRESP(NOTFND)
            EXEC CICS ABEND ABCODE('W002') END-EXEC
            CONTINUE
    END-EVALUATE
.
INVREQ-PROCESSING.
    IF EIBRESP2 = 6 THEN

```

```
*   ** An EIBRESP2 of 6 indicates a SOAP fault **
*   **   has been returned in DFHWS-BODY       **
      EXEC CICS
          GET CONTAINER('DFHWS-BODY')
          CHANNEL(WS-CHANNEL-NAME)
          INTO(WS-DFHWS-BODY)
      END-EXEC
      DISPLAY WS-DFHWS-BODY
  END-IF
  .

*End Procedure Division
END PROGRAM 'CHOICETE'.
```

WSDL <xsd:choice>

```
<?xml version="1.0"?>
<!--This document was generated using 'DFHLS2WS' at mapping level '2.2'. -->
<definitions targetNamespace="http://www.INLINE01.REQY.com"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:reqns="http://www.INLINE01.REQY.Request.com"
xmlns:resns="http://www.INLINE01.RESPY.Response.com"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.INLINE01.REQY.com">
  <types>
    <xsd:schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://www.INLINE01.REQY.Request.com"
xmlns:tns="http://www.INLINE01.REQY.Request.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:annotation>
        <xsd:documentation source="http://www.ibm.com/software/htp/cics/annotations">
          This schema was generated by the CICS Web services
          assistant.</xsd:documentation>
        </xsd:annotation>
        <xsd:annotation>
          <xsd:appinfo source="http://www.ibm.com/software/htp/cics/annotations">
            com.ibm.cics.wsdl.properties.mappingLevel=2.2</xsd:appinfo>
          </xsd:annotation>
          <xsd:complexType abstract="false" block="#all" final="#all"
mixed="false" name="ProgramInterface">
            <xsd:sequence>
              <xsd:element name="ws_reqarea" nillable="false">
                <xsd:complexType mixed="false">
                  <xsd:sequence>
                    <xsd:element name="choiceData">
                      <xsd:complexType>
                        <xsd:choice>
                          <xsd:element name="firstchoice"
type="xsd:string" />
                          <xsd:element name="secondchoice"
type="xsd:string" />
                        </xsd:choice>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:schema>
    </types>
  </definitions>
```

```

        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="INLINE01operation" nillable="false"
        type="tns:ProgramInterface" />
</xsd:schema>
<xsd:schema attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="http://www.INLINE01.RESPY.Response.com"
    xmlns:tns="http://www.INLINE01.RESPY.Response.com"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation source="http://www.ibm.com/software/htp/cics/annotations">
            This schema was generated by the CICS Web services
            assistant.</xsd:documentation>
        </xsd:annotation>
    <xsd:annotation>
        <xsd:appinfo source="http://www.ibm.com/software/htp/cics/annotations">
            com.ibm.cics.wsd1.properties.mappingLevel=2.2</xsd:appinfo>
        </xsd:annotation>
    <xsd:complexType abstract="false" block="#all" final="#all"
        mixed="false" name="ProgramInterface">
        <xsd:sequence>
            <xsd:element name="ws_retarea" nillable="false">
                <xsd:complexType mixed="false">
                    <xsd:sequence>
                        <xsd:element name="choiceData">
                            <xsd:complexType>
                                <xsd:choice>
                                    <xsd:element name="firstchoice"
                                        type="xsd:string" />
                                    <xsd:element name="secondchoice"
                                        type="xsd:string" />
                                </xsd:choice>
                            </xsd:complexType>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="INLINE01operationResponse"
        nillable="false" type="tns:ProgramInterface" />
</xsd:schema>
</types>
<message name="INLINE01operationResponse">

```

```

    <part element="resns:INLINE01OperationResponse"
        name="ResponsePart" />
</message>
<message name="INLINE01OperationRequest">
    <part element="reqns:INLINE01Operation" name="RequestPart" />
</message>
<portType name="INLINE01Port">
    <operation name="INLINE01Operation">
        <input message="tns:INLINE01OperationRequest"
            name="INLINE01OperationRequest" />
        <output message="tns:INLINE01OperationResponse"
            name="INLINE01OperationResponse" />
    </operation>
</portType>
<binding name="INLINE01HTTPSoapBinding" type="tns:INLINE01Port">
    <!-- This soap:binding indicates the use of SOAP 1.1 -->
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="INLINE01Operation">
        <soap:operation soapAction="" style="document" />
        <input name="INLINE01OperationRequest">
            <soap:body parts="RequestPart" use="literal" />
        </input>
        <output name="INLINE01OperationResponse">
            <soap:body parts="ResponsePart" use="literal" />
        </output>
    </operation>
</binding>
<service name="INLINE01Service">
    <port binding="tns:INLINE01HTTPSoapBinding"
        name="INLINE01Port">
        <!-- This soap:address indicates the location of the Web service over HTTP.
            Please replace "my-server" with the TCPIP host name of your CICS
region.
            Please replace "my-port" with the port number of your CICS
TCPIPService. -->
        <soap:address location="http://my-server:my-port/inline/test" />
        <!-- This soap:address indicates the location of the Web service over HTTPS.
-->
        <!-- <soap:address location="https://my-server:my-port/inline/test"/> -->
        <!-- This soap:address indicates the location of the Web service over WebSphere
MQSeries.
            Please replace "my-queue" with the appropriate queue name. -->

```

```
        <!-- <soap:address  
location="jms:/queue?destination=my-queue&connectionFactory=()&targetService=  
/inline/test&initialContextFactory=com.ibm.mq.jms.Nojndi" /> -->  
        </port>  
    </service>  
</definitions>
```

Request Language Structure - choicI01

```
* ++++++
* This file contains the generated request language structure(s)
* for WSDL operation 'INLINE01operation'.
* The response message for this WSDL Operation may be replaced
* with a SOAP Fault message.
* This structure was generated using 'DFHWS2LS' at mapping level
* '2.2'.
*
*
*   03 INLINE01operation.
*     06 wsXreqarea.
*       09 choiceData.
*
*
* The 'choiceData-enum' field indicates which option from a set
* of possible values is being used. The associated value is
* stored in the container referenced in 'choiceData-cont'.
* A value of X'00' indicates no content.
* A value of X'01' indicates an instance of structure
* 'choicI01-firstchoice'.
* A value of X'02' indicates an instance of structure
* 'choicI01-secondchoice'.
*
*       12 choiceData-enum          PIC X DISPLAY.
*       88 empty                    VALUE X'00'.
*       88 firstchoice              VALUE X'01'.
*       88 secondchoice            VALUE X'02'.
*       12 choiceData-cont         PIC X(16).
*
*
* This structure describes data associated with enumeration
* 'choiceData-enum' with a value X'01'.
* 01 choicI01-firstchoice.
*
* Comments for field 'firstchoice':
* This field represents the value of XML element
* '/INLINE01operation/ws_reqarea/choiceData/firstchoice'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
*   03 firstchoice-length          PIC S9999 COMP-5 SYNC.
*   03 firstchoice                 PIC X(255).
```



```

*
*
* This structure describes data associated with enumeration
* 'choiceData-enum' with a value X'02'.
* 01 choicI01-secondchoice.
*
* Comments for field 'secondchoice':
* This field represents the value of XML element
* '/INLINE01operation/ws_reqarea/choiceData/secondchoice'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
* 03 secondchoice-length          PIC S9999 COMP-5 SYNC.
* 03 secondchoice                 PIC X(255).
*
*
* ++++++

```

```

03 INLINE01operation.
06 wsXreqarea.
09 choiceData.

```

```

12 choiceData-enum          PIC X DISPLAY.
88 empty                    VALUE X'00'.
88 firstchoice              VALUE X'01'.
88 secondchoice             VALUE X'02'.
12 choiceData-cont         PIC X(16).

```

```

01 choicI01-firstchoice.
03 firstchoice-length      PIC S9999 COMP-5 SYNC.
03 firstchoice             PIC X(255).

```

```

01 choicI01-secondchoice.
03 secondchoice-length    PIC S9999 COMP-5 SYNC.
03 secondchoice           PIC X(255).

```

Response Language Structure - choic001

```
* ++++++
* This file contains the generated response language
* structure(s) for WSDL operation 'INLINE010operation'.
* The response message for this WSDL Operation may be replaced
* with a SOAP Fault message.
* This structure was generated using 'DFHWS2LS' at mapping level
* '2.2'.
*
*
*      03 INLINE010operationResponse.
*          06 wsXretarea.
*              09 choiceData.
*
*
* The 'choiceData-enum' field indicates which option from a set
* of possible values is being used. The associated value is
* stored in the container referenced in 'choiceData-cont'.
* A value of X'00' indicates no content.
* A value of X'01' indicates an instance of structure
* 'choic001-firstchoice'.
* A value of X'02' indicates an instance of structure
* 'choic001-secondchoice'.
*
*      12 choiceData-enum          PIC X DISPLAY.
*      88 empty                   VALUE X'00'.
*      88 firstchoice             VALUE X'01'.
*      88 secondchoice            VALUE X'02'.
*      12 choiceData-cont        PIC X(16).
*
*
* This structure describes data associated with enumeration
* 'choiceData-enum' with a value X'01'.
*      01 choic001-firstchoice.
*
* Comments for field 'firstchoice':
* This field represents the value of XML element
* '/INLINE010operationResponse/ws_retarea/choiceData/firstchoice'
* .
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
*      03 firstchoice-length      PIC S9999 COMP-5 SYNC.
```

```

*   03 firstchoice                PIC X(255).
*
*
* This structure describes data associated with enumeration
* 'choiceData-enum' with a value X'02'.
* 01 choic001-secondchoice.
*
* Comments for field 'secondchoice':
* This field represents the value of XML element
* '/INLINE01operationResponse/ws_retarea/choiceData/secondchoice
* '.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'preserve'.
* This field contains a varying length array of characters or
* binary data.
*   03 secondchoice-length        PIC S9999 COMP-5 SYNC.
*   03 secondchoice                PIC X(255).
*
*
*
* ++++++

```

```

03 INLINE01operationResponse.
  06 wsXretarea.
    09 choiceData.

```

```

      12 choiceData-enum          PIC X DISPLAY.
      88 empty                    VALUE X'00'.
      88 firstchoice              VALUE X'01'.
      88 secondchoice             VALUE X'02'.
      12 choiceData-cont         PIC X(16).

```

```

01 choic001-firstchoice.
  03 firstchoice-length          PIC S9999 COMP-5 SYNC.
  03 firstchoice                 PIC X(255).

```

```

01 choic001-secondchoice.
  03 secondchoice-length        PIC S9999 COMP-5 SYNC.
  03 secondchoice                PIC X(255).

```

Program to call minOccurs/maxOccurs example service

```
PROCESS CICS,NODYNAM,NSYMBOL(NATIONAL),TRUNC(STD)
* *****
* *****RDz**7.5*****
* *****
*           New CICS TS 3.x Web Service Requester
* *****RDz**7.5*****
* *****

IDENTIFICATION DIVISION.
*Begin Identification Divsion
  PROGRAM-ID. 'REDBOOKW'.
  AUTHOR. WD4Z.
  INSTALLATION. 9.1.200.V200903111338.
  DATE-WRITTEN. 21/09/09 18:38.
*End Identification Divsion
DATA DIVISION.
*Begin Data Divsion
WORKING-STORAGE SECTION.
*Begin Working-Storage Section
* *****
*           Operations Available On The Remote Web Service
* *****
  1 OPERATION-NAME-1.
  2 PIC X(17) USAGE DISPLAY
    VALUE 'INLINE01operation'.
*End Working-Storage Section
LOCAL-STORAGE SECTION.
*Begin Local-Storage Section
* *****
*           Program Work Variables
* *****
  1 SOAP-PIPELINE-WORK-VARIABLES.
  2 WS-WEBSERVICE-NAME PIC X(32).
  2 WS-OPERATION-NAME PIC X(255).
  2 WS-CONTAINER-NAME PIC X(16).
  2 WS-CHANNEL-NAME PIC X(16).
  2 COMMAND-RESP PIC S9(8) COMP.
  2 COMMAND-RESP2 PIC S9(8) COMP.
*Specify A URI To Override The Web Service Description
  1 URI-RECORD-STRUCTURE.
  2 FILLER PIC X(10).
  2 WS-URI-OVERRIDE PIC X(255).
```

```

1 WS-DFHWS-BODY      PIC x(400) value spaces.
1 WS-XML-ERRORMSG   PIC x(400) value spaces.

1 WS-RECORDS-ARRAY.
  2 WS-RECORD        PIC X(80) occurs 20 times.

* used to display the records returned by the service
1 ws-record-returned.
  2 Filler           PIC X(26)
    value ' returned record number '.
  2 ws-returned-rec-num pic 99.
  2 Filler           PIC X(05)
    value '===> '.
  2 ws-returned-rec-data pic X(80).

1 records-length    PIC s9(8) comp.
1 ws-record-data.
  2 FILLER           pic x(07) value 'Record '.
  2 ws-count         pic 99 value zero.
* *****
*                               Language Structures
* *****

1 LANG-REDBOI01.
  COPY redboI01.
1 LANG-REDBO001.
  COPY redbo001.
*End Local-Storage Section
LINKAGE SECTION.
*Begin Linkage Section
*End Linkage Section
*End Data Division
PROCEDURE DIVISION
.
*Begin Procedure Division
MAINLINE SECTION.
* -----
*                               Initialize Work Variables
* -----
  INITIALIZE SOAP-PIPELINE-WORK-VARIABLES.
  INITIALIZE URI-RECORD-STRUCTURE.
* -----
* Container DFHWS-DATA must be present when a service requeste
* r program issues an EXEC CICS INVOKE WEBSERVICE command. Whe
* n the command is issued, CICS converts the language structur

```

```

* e that is in the container into a SOAP request. When the soa
* p response is received, CICS converts it into another langua
* ge structure that is returned to the application in the same
* container.

```

```

* -----
MOVE 'DFHWS-DATA'
TO WS-CONTAINER-NAME

```

```

* -----
* Channel Passed To The Web Service Call
* -----

```

```

MOVE 'SERVICE-CHANNEL'
TO WS-CHANNEL-NAME

```

```

* -----
* WEBSERVICE resource installed in this CICS region
* -----

```

```

MOVE 'redbookWS6'
TO WS-WEBSERVICE-NAME

```

```

* -----
* Operation To Invoke On The Remote Web Service
* -----

```

```

MOVE OPERATION-NAME-1
TO WS-OPERATION-NAME

```

```

* -----
* Populate Request Language Structure
* -----

```

```

INITIALIZE LANG-REDBO101

```

```

*--- we are going to send 4 records
move 4 to recs-num of wsXreqarea
DISPLAY " "
DISPLAY "====="
DISPLAY "Sending " recs-num of wsXreqarea " records"

```

```

*--- populate our array with our data
Perform recs-num of wsXreqarea times
add 1 to ws-count
Move ws-record-data to recs2 of redboI01-recs
move redboI01-recs to WS-RECORD(ws-count)
END-Perform

```

```

*--- calculate how long the data is
compute records-length =
length of redboI01-recs * recs-num of wsXreqarea

```

```

*--- store the name of our data container in the

```

```

*--- request language structure
    move "RECS-CONTAINER" to recs-cont of wsXreqarea

*--- put the array into the container
    EXEC CICS PUT CONTAINER(recs-cont of wsXreqarea)
        CHANNEL(WS-CHANNEL-NAME)
        FROM(WS-RECORDS-ARRAY)
        FLENGTH(records-length)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

*
* .....
* .....
* .....
* -----
*           Put Request Language Structure Into SOAP Container
* -----
    EXEC CICS PUT CONTAINER(WS-CONTAINER-NAME)
        CHANNEL(WS-CHANNEL-NAME)
        FROM(LANG-REDBOIO1)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

* -----
*                               Invoke The Web Service
* -----

*--- override the URI - remove if the WSDL has the correct URI
    Move 'http://9.173.199.45:9080/RedbookWS6/INLINE01Service'
        to WS-URI-OVERRIDE

    EXEC CICS INVOKE WEBSERVICE(WS-WEBSERVICE-NAME)
        CHANNEL(WS-CHANNEL-NAME)
        URI(WS-URI-OVERRIDE)
        OPERATION(WS-OPERATION-NAME)
        RESP(COMMAND-RESP) RESP2(COMMAND-RESP2)
    END-EXEC
    PERFORM CHECK-WEBSERVICE-COMMAND

* -----
*                               Receive Response Language Structure
* -----
    EXEC CICS GET CONTAINER(WS-CONTAINER-NAME)
        CHANNEL(WS-CHANNEL-NAME)
        INTO(LANG-REDBO001)
    END-EXEC
    PERFORM CHECK-CONTAINER-COMMAND

```

```

* -----
*           Process Response Language Structure
* -----

*--- get the returned data which is in the container
*--- named in the response language structure
EXEC CICS GET CONTAINER(recs-cont of wsXretarea)
      CHANNEL(WS-CHANNEL-NAME)
      INTO(WS-RECORDS-ARRAY)
END-EXEC
PERFORM CHECK-CONTAINER-COMMAND

      DISPLAY '===== '
      DISPLAY recs-num of wsXretarea ' records returned'

*--- Display each of the returned records.
*--- The number of records returned is in recs-num
*--- which has been extracted from the response container
*--- into the response language structure
      move 1 to ws-count
      PERFORM recs-num of wsXretarea times
      move ws-count to ws-returned-rec-num
      MOVE ws-record(ws-count) to ws-returned-rec-data
      DISPLAY ws-record-returned
      add 1 to ws-count

      END-PERFORM

* -----
*           Finished
* -----

      EXEC CICS RETURN
      END-EXEC
.
CHECK-CONTAINER-COMMAND.
      EVALUATE COMMAND-RESP
      WHEN DFHRESP(CCSIDERR)
        EXEC CICS ABEND ABCODE('C001') END-EXEC
        CONTINUE
      WHEN DFHRESP(CONTAINERERR)
        EXEC CICS ABEND ABCODE('C002') END-EXEC
        CONTINUE
      WHEN DFHRESP(INVREQ)

```



```

        EXEC CICS ABEND ABCODE('C003') END-EXEC
        CONTINUE
    WHEN DFHRESP(LENGERR)
        EXEC CICS ABEND ABCODE('C004') END-EXEC
        CONTINUE
    END-EVALUATE
.
CHECK-WEBSERVICE-COMMAND.
    EVALUATE COMMAND-RESP
    WHEN DFHRESP(INVREQ)
        PERFORM INVREQ-PROCESSING
        EXEC CICS ABEND ABCODE('W001') END-EXEC
        CONTINUE
    WHEN DFHRESP(NOTFND)
        EXEC CICS ABEND ABCODE('W002') END-EXEC
        CONTINUE
    END-EVALUATE
.
INVREQ-PROCESSING.
    IF EIBRESP2 = 6 THEN
*   ** An EIBRESP2 of 6 indicates a SOAP fault **
*   ** has been returned in DFHWS-BODY      **
        EXEC CICS
            GET CONTAINER('DFHWS-BODY')
            CHANNEL(WS-CHANNEL-NAME)
            INTO(WS-DFHWS-BODY)
        END-EXEC
        DISPLAY WS-DFHWS-BODY
    END-IF
*   ** An EIBRESP2 of 13 indicates an input error **
*   ** has been detected a message is returned **
*   ** in DFH-XML-ERRORMSG                    **
    IF EIBRESP2 = 13 THEN
        EXEC CICS
            GET CONTAINER('DFH-XML-ERRORMSG')
            CHANNEL(WS-CHANNEL-NAME)
            INTO(WS-XML-ERRORMSG)
        END-EXEC
        DISPLAY WS-XML-ERRORMSG
    END-IF
.
*End Procedure Division

```

WSDL - minOccurs/maxOccurs

```
<?xml version="1.0"?>
<!--This document was generated using 'DFHLS2WS' at mapping level '2.2'. -->
<definitions targetNamespace="http://www.INLINE01.REQY.com"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:reqns="http://www.INLINE01.REQY.Request.com"
xmlns:resns="http://www.INLINE01.RESPY.Response.com"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.INLINE01.REQY.com">
  <types>
    <xsd:schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://www.INLINE01.REQY.Request.com"
xmlns:tns="http://www.INLINE01.REQY.Request.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:annotation>
        <xsd:documentation source="http://www.ibm.com/software/htp/cics/annotations">
          This schema was generated by the CICS Web services
          assistant.</xsd:documentation>
        </xsd:annotation>
        <xsd:annotation>
          <xsd:appinfo source="http://www.ibm.com/software/htp/cics/annotations">
            com.ibm.cics.wsdl.properties.mappingLevel=2.2</xsd:appinfo>
          </xsd:annotation>
          <xsd:complexType abstract="false" block="#all" final="#all"
mixed="false" name="ProgramInterface">
            <xsd:sequence>
              <xsd:element name="ws_reqarea" nillable="false">
                <xsd:complexType mixed="false">
                  <xsd:sequence>
                    <xsd:element maxOccurs="10" minOccurs="1"
name="recs" nillable="false">
                      <xsd:complexType mixed="false">
                        <xsd:sequence>
                          <xsd:element name="recs" nillable="false">
                            <xsd:simpleType>
                              <xsd:annotation>
                                <xsd:appinfo
source="http://www.ibm.com/software/htp/cics/annotations">
                                  com.ibm.cics.wsdl.properties.charlength=fixed
                                </xsd:appinfo>
                              </xsd:annotation>
                            </xsd:element>
                        </xsd:sequence>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:annotation>
      </xsd:schema>
    </types>
  </definitions>
```

```

        <xsd:restriction base="xsd:string">
            <xsd:maxLength value="80" />
            <xsd:whiteSpace value="collapse" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="INLINE01operation" nillable="false"
type="tns:ProgramInterface" />
</xsd:schema>
<xsd:schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://www.INLINE01.RESPY.Response.com"
xmlns:tns="http://www.INLINE01.RESPY.Response.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation source="http://www.ibm.com/software/htp/cics/annotations">
            This schema was generated by the CICS Web services
            assistant.</xsd:documentation>
        </xsd:annotation>
    <xsd:annotation>
        <xsd:appinfo source="http://www.ibm.com/software/htp/cics/annotations">
            com.ibm.cics.wsd1.properties.mappingLevel=2.2</xsd:appinfo>
        </xsd:annotation>
    <xsd:complexType abstract="false" block="#all" final="#all" mixed="false"
name="ProgramInterface">
        <xsd:sequence>
            <xsd:element name="ws_retarea" nillable="false">
                <xsd:complexType mixed="false">
                    <xsd:sequence>
                        <xsd:element maxOccurs="10" minOccurs="1" name="recs"
nillable="false">
                            <xsd:complexType mixed="false">
                                <xsd:sequence>
                                    <xsd:element name="recs" nillable="false">
                                        <xsd:simpleType>
                                            <xsd:annotation>

```

```

        <xsd:appinfo
source="http://www.ibm.com/software/htp/cics/annotations">
        com.ibm.cics.wsdl.properties.charlength=fixed

com.ibm.cics.wsdl.properties.synchronized=false</xsd:appinfo>
        </xsd:annotation>
        <xsd:restriction base="xsd:string">
        <xsd:maxLength value="80" />
        <xsd:whiteSpace value="collapse" />
        </xsd:restriction>
        </xsd:simpleType>
        </xsd:element>
        </xsd:sequence>
        </xsd:complexType>
        </xsd:element>
        </xsd:sequence>
        </xsd:complexType>
        </xsd:element>
        </xsd:sequence>
        </xsd:complexType>
        <xsd:element name="INLINE01OperationResponse"
nillable="false" type="tns:ProgramInterface" />
</xsd:schema>
</types>
<message name="INLINE01OperationResponse">
    <part element="resns:INLINE01OperationResponse"
name="ResponsePart" />
</message>
<message name="INLINE01OperationRequest">
    <part element="reqns:INLINE01Operation" name="RequestPart" />
</message>
<portType name="INLINE01Port">
    <operation name="INLINE01Operation">
        <input message="tns:INLINE01OperationRequest"
name="INLINE01OperationRequest" />
        <output message="tns:INLINE01OperationResponse"
name="INLINE01OperationResponse" />
    </operation>
</portType>
<binding name="INLINE01HTTPSoapBinding" type="tns:INLINE01Port">
    <!-- This soap:binding indicates the use of SOAP 1.1 -->
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="INLINE01Operation">
        <soap:operation soapAction="" style="document" />

```

```

    <input name="INLINE01operationRequest">
      <soap:body parts="RequestPart" use="literal" />
    </input>
    <output name="INLINE01operationResponse">
      <soap:body parts="ResponsePart" use="literal" />
    </output>
  </operation>
</binding>
<service name="INLINE01Service">
  <port binding="tns:INLINE01HTTPSoapBinding"
    name="INLINE01Port">
    <!-- This soap:address indicates the location of the Web service over HTTP.
      Please replace "my-server" with the TCPIP host name of your CICS
region.
      Please replace "my-port" with the port number of your CICS
TCPIP SERVICE. -->
    <soap:address location="http://localhost:9080/RedbookWS6/INLINE01Service" />
    <!-- This soap:address indicates the location of the Web service over HTTPS.
-->
    <!-- <soap:address location="https://my-server:my-port/inline/test"/> -->
    <!-- This soap:address indicates the location of the Web service over WebSphere
MQSeries.
      Please replace "my-queue" with the appropriate queue name. -->
    <!-- <soap:address
location="jms:/queue?destination=my-queue&connectionFactory=()&targetService=
/inline/test&initialContextFactory=com.ibm.mq.jms.Nojndi" /> -->
    </port>
  </service>
</definitions>

```

Request Language Structure - redboI01

```
* ++++++
* This file contains the generated request language structure(s)
* for WSDL operation 'INLINE01operation'.
* The response message for this WSDL Operation may be replaced
* with a SOAP Fault message.
* This structure was generated using 'DFHWS2LS' at mapping level
* '2.2'.
*
*
*      03 INLINE01operation.
*      06 wsXreqarea.
*
*
* CONTAINER 'recs-cont' contains 'recs-num' instances of
* structure 'redboI01-recs', each of which represents an
* instance of XML element '/INLINE01operation/ws_reqarea/recs'.
* The CONTAINER must be read from and written to in BIT mode.
* There should be at least '1' instance(s).
* There should be at most '10' instance(s).
*      09 recs-num                PIC S9(9) COMP-5 SYNC.
*      09 recs-cont              PIC X(16).
*
*
* This structure describes one instance of the data in CONTAINER
* 'recs-cont'.
*      01 redboI01-recs.
*      03 recs.
*
* Comments for field 'recs2':
* This field represents the value of XML element
* '/INLINE01operation/ws_reqarea/recs/recs'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'collapse'.
* XML 'maxLength' facet value: '80'.
*      06 recs2                  PIC X(80).
*
*
* ++++++
```

03 INLINE01operation.
06 wsXreqarea.

```
09 recs-num          PIC S9(9) COMP-5 SYNC.  
09 recs-cont        PIC X(16).  
  
01 redboI01-recs.  
  03 recs.  
    06 recs2        PIC X(80).
```

Response Language Structure - redbo001

```
* ++++++
* This file contains the generated response language
* structure(s) for WSDL operation 'INLINE010operation'.
* The response message for this WSDL Operation may be replaced
* with a SOAP Fault message.
* This structure was generated using 'DFHWS2LS' at mapping level
* '2.2'.
*
*
*      03 INLINE010operationResponse.
*          06 wsXretarea.
*
*
* CONTAINER 'recs-cont' contains 'recs-num' instances of
* structure 'redbo001-recs', each of which represents an
* instance of XML element
* '/INLINE010operationResponse/ws_retarea/recs'. The CONTAINER
* must be read from and written to in BIT mode.
* There should be at least '1' instance(s).
* There should be at most '10' instance(s).
*      09 recs-num                PIC S9(9) COMP-5 SYNC.
*      09 recs-cont                PIC X(16).
*
*
* This structure describes one instance of the data in CONTAINER
* 'recs-cont'.
*      01 redbo001-recs.
*          03 recs.
*
* Comments for field 'recs2':
* This field represents the value of XML element
* '/INLINE010operationResponse/ws_retarea/recs/recs'.
* XML data type: 'string'.
* XML 'whiteSpace' facet value: 'collapse'.
* XML 'maxLength' facet value: '80'.
*      06 recs2                    PIC X(80).
*
*
* ++++++
```

03 INLINE010operationResponse.


```
06 wsXretarea.  
    09 recs-num          PIC S9(9) COMP-5 SYNC.  
    09 recs-cont        PIC X(16).  
  
01 redbo001-recs.  
    03 recs.  
    06 recs2            PIC X(80).
```


Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 354. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *Securing CICS Web Services*, SG24-7658
- ▶ *CICS Web Services Workload Management and Availability*, SG24-7144
- ▶ *Considerations for CICS Web Services Performance*, SG24-7687
- ▶ *Implementing CICS Web Services*, SG24-7206
- ▶ *Architecting Access to CICS within an SOA*, SG24-5466

Other publications

These publications are also relevant as further information sources:

- ▶ *CICS Transaction Server for z/OS Version 4 Release 1*, SC34-7020-00

Online resources

These Web sites are also relevant as further information sources:

- ▶ The CICS Web services knowledge collection
<http://www-01.ibm.com/support/docview.wss?uid=swg27010507>
- ▶ The Information Center for CICS TS V3.1
<http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp>
- ▶ The Information Center for CICS TS V3.2
<http://publib.boulder.ibm.com/infocenter/cicsts/v3r2/index.jsp>
- ▶ The Information Center for CICS TS V4.1
<http://publib.boulder.ibm.com/infocenter/cicsts/v4r1/index.jsp>

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this Web site:

ibm.com/redbooks

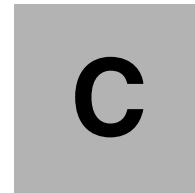
Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services



Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG247126>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247126.

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.



Application Development for CIGS Web Services

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages

Index

tag 278
tag 258

Numerics

3270 interface 177

A

assembler 104

B

base64 binary data 10
batch 127
BINDING 175
BMS map 140
bottom-up approach 63, 142
bundle directory 221

C

C 34
C/C++ 104
C++ 34
catalog manager 74, 79
CCSID
 description 80
CEDA 88
 DEF WEBSERVICE(EXINQCWS) G(EXAM-
 PLE) 92
 INSTALL PIPELINE(EXPIPE01) G(SOAS-
 DEVWS) 90
 INSTALL PIPELINE(EXPIPE02) G(SOAS-
 EVWS) 90
CEDA transaction 44
CEMT
 I TCPIPS 100
 INQUIRE WEBSERVICE 90
CEMT INQUIRE 178
CEOT 77
channels and containers 59
CICS 127
 applications 73
 as a service provider 36

BMS map 140
catalog manager 137, 143
CEDA DEF TCPIPSERVICE(EXMPPPORT) 88
CEDA I G(SOADEV) 77
COBOL 141
commands
 CEDA DEF URIMAP(INQCURI) G(EXAM-
 PLE) 93
 CEMT INQUIRE URIMAP(\$509340) 141
 CEMT INQUIRE WEBSERVICE 141
 CEMT PERFORM PIPELINE(EXPIPE03)
 SCAN 140
Group
 DFH\$EXBS 77
infrastructure 178
transaction
 CPIH 39
 EGUI 182
transactions
 ECFG 76, 78
 EGUI 76
 URIMAP 43
 Web service provider 138
 Web Services Assistant 34
CICS Explorer 44
CICS options 116
CICS SFR
 debugger 249
CICS transaction
 CWYN 39
CICS Transaction Gateway 70
CICS TRANSFORM 221
CICS TS
 Web service resource definitions 43
CICS Web service APIs 60
CICS Web Services Addressing 58
CICSplex® SM Business Application Services 44
CMNISO01 140
CMNISO1 140
CMNISO01 140
COBOL 34–35, 73
COBOL REDEFINES 142
code page support 80
COMMAREA 37–38, 40, 89, 143

- compiler options 115
 - ADATA 115
 - EXIT 115
 - SYSLIB 115
 - TEST 115
- components of RDz 104
- CONFIGFILE 37
- container 38
- CONTID 146
- Copybook
 - DFH0XCP4 142
- copybook 142
- CORBA 4
- CPIH 39
 - transaction 39
- CREATE PIPELINE 60
- CREATE URIMAP 60
- CREATE WEBSERVICE 60
- custom handlers 214
- CWXN 39
 - transaction 39

D

- Data Store Stub 79
- Data Store VSAM 79
- datastore type 78
- DB2 stored procedures 126
- Debugging with RDz 126
- DFH\$ECAT 76
- DFH\$ECNF 76
- DFH\$EXBS 77
- DFH\$EXWS 88
- DFH\$WBSR 219
- DFH\$WBST 219
- DFH0XCMN 143
- DFH0XCP1 148
- DFH0XCP2 148, 177
- DFH0XCP4 142
- DFH0XCP7 176–177
- DFH0XCP8 176
- DFH0XSOD 78, 181
- DFH0XWC3 140, 147
- DFH0XWC4 140, 147
- DFH0XWOD 79, 176, 181
- DFHCSDUP 44
- DFHERROR 60
- DFHFUNCTION 60, 215
 - values

- HANDLER-ERROR 215
- NO-RESPONSE 215
- PROCESS-REQUEST 215
- RECEIVE-REQUEST 215
- RECEIVE-RESPONSE 215
- SEND-REQUEST 215
- SEND-RESPONSE 215
- DFHHANDLERPLIST 60
- DFHHEADER 60
- DFHLS2WS 32, 34–35, 37, 142
 - JCL 142, 165
- DFHLS2WS with WSRR 165
- DFHNORESPONSE 60
- DFHPITP 40, 214
- DFHREQUEST 60, 217
- DFHRESPONSE 60, 217
 - deletion 217
- DFHRPL 155
- DFHSC2LS 221
- DFH-SERVICEPLIST 60
- DFHWS2LS 32, 34, 41, 174
- DFHWS2LS with WSRR 167
- DFHWS-APPHANDLER 60
- DFHWS-BODY 60, 170, 221
- DFHWS-DATA 42–43, 60
- DFHWS-OPERATION 60, 214
- DFHWS-PIPELINE 60
- DFHWS-SOAPACTION 60, 214
- DFHWS-SOAPLEVEL 60
- DFHWS-TRANID 60
- DFHWS-URI 60
- DFHWS-USERID 60
- DFHWS-WEBSERVICE 60
 - container 40
- DFHWS-XMLNS 60, 170
- Domains 210

E

- EAR file 189
- ECFG 76, 78, 181
- editor 109
- EGUI 75–76
- electronic data interchange (EDI) 2
- Enterprise Generation Language (EGL) 35, 104
- Enterprise Services Toolkit (EST) 35
- error
 - handling 15
- EXEC CICS API commands

- INVOKE WEBSERVICE 33
- SOAPFAULT ADD | CREATE | DELETE 33
- EXEC CICS INVOKE WEBSERVICE 198
- EXEC CICS LINK 196
- EXEC CICS TRANSFORM XMLTODATA CHANNEL(channel-name) 225
- EXEC CICS WEB READ 216
- EXINQSWs 92, 94
- EXMPCONF 75
- EXODEPWS 92, 94
- EXODRQWS 92
- EXORDRWS 92, 94
- Extensible Markup Language (XML) 6, 9

F

- FTP 7, 95

H

- HFS
 - directory 37, 85–86, 175
- HFS directory 36
- HTTP
 - server support 70
 - traffic 88
 - transport 36, 88
- HTTP client 5
- HTTP header 214

I

- INLINE-MAXOCCURS-LIMIT 227
- INQSURI 93–94
- INQUIRE WEBSERVICE 60
- interactive development environments (IDEs) 35
- interfaces 204
- INVOKE WEBSERVICE 60
- INVREQ 221

J

- J2EE 188
- Java 104
- Java bean Web Service 184
- Java EE 290
- Java program with RDz 123
- JavaBean 183
- JCL 32
 - CMNISW 146
 - DFHLS2WS 142, 165

- SVLCOB 155
- WS2LS 175
- JEE 104
- JMS 7

L

- LANG 175
- Link3270 Bridge 70
- LOGFILE 143

M

- MAPPING-LEVEL 176
- maxOccurs 227
- meet-in-the-middle approach 66
- message exchange pattern (MEP) 7
- META-INF 189
- MIME 10
- minOccurs and maxOccurs 283
- MTOM 10
- mustUnderstand 14

N

- Namespaces 12
- namespaces 12

O

- ODEPURI 93–94
- OPERATIONS 176
- ORDRURI 93–94
- Outbound WebService 79, 181
- Outbound WebService URI 80

P

- PDSE 155
- PDSLIB 175
- PERFORM PIPELINE SCAN 60
- perspective 106
- PIPELINE 45, 178
 - definition 86
- pipeline
 - configuration file 47
- PL/I 34, 104
- port definition 28
- POST 8
- PROTOCOL(HTTP) 37

R

- Rational Developer for System z 104
- Rational Developer for System Z (RDz) 32, 265
 - and CICS application development 104
- RDz
 - components 104
- RDz Debugging 126
- RDz Java program 123
- RDz SCA Tooling 210
- RECEIVE-REQUEST 215
- Redbooks Web site 354
 - Contact us xiv
- remote procedure call (RPC) 16
- REQMEM 146
- REQMEM and RESPMEM 175
- Resource Definition Online (RDO) 92
- Resource Definition Online (RDO). 93
- RESP2 221
- RESPMEM 146
- RPC 16
 - style 30

S

- SCA
 - components 205
 - components, see service
 - component architecture
 - components
 - composites 206
 - operations 206
 - project 211
 - RDz tooling 210
 - runtime 205
 - services 206, 208
- SCA, see service
 - component architecture
- SCD, see service
 - component description language
- service
 - component architecture 204, 206
 - components 204
 - component description language 205
- service broker 3
- Service Component Architecture (SCA) 204
- service oriented architecture (SOA) 1–2, 205
 - service requestors 74
- service provider 3, 40
- SHELF 37

- SMTP 7
- SOAP 6
 - binding 29
 - body 15
 - inbound data conversion 40
 - outbound data conversion 40
 - communication styles 16
 - document 16
 - RPC 16
 - encodings 16
 - literal 17
 - SOAP encoding 17
 - envelope 11, 13
 - fault 15
 - fault API 213, 220
 - headers 13
 - intermediary 14
 - introduction 11
 - messaging mode 17
 - MustUnderstand 14
 - namespaces 12
 - request 218
 - validation 251
- SOAP Body 214
- SOAP header 214
- SOAPFAULT 52, 220
 - commands
 - ADD 53
 - CREATE 53
 - DELETE 53
- SOAPFAULT ADD 60, 220
- SOAPFAULT CREATE 60, 220
- SOAPFAULT DELETE 60, 220
- SQL 104
- SQL options 116
- SQLJ 126
- SSL/TLS 219
- state information 218
- SVLCOB
 - JCL 155

T

- TCP/IP
 - server name 38
- TCPIP SERVICE 37, 39
 - creation 88
 - definition 88
- top-down approach 65, 148

TSO 127

U

UDDI

Universal Description, Discovery, and Integration 9

UML model, see unified modelling language

unified modelling language 205

URIMAP 39

creation 93

resources 90

V

VSAM 75–76, 78

file 74

VSAM File 79

W

Web service

Interoperability 9

properties 5

skeleton 183

Web services 1

Web services assistant 260

Web Services Coordination 9

Web Services Description Language (WSDL) 5, 18, 164, 175

binding 18, 27

bindings 29

definition 23

document 18

anatomy 19

generated by Rational Developer 213

message 18, 24

namespaces 22

operation 18, 25

port 18, 28

port type 18, 25

service definition 28

SOAP binding 29

type 18

types 23, 245

types not supported by WS2LS 213

Web Services Description Language 7

Web Services Explorer 156

Web Services Trust Language 10

WEBSERVICE 178

WebSphere DataPower 219

WebSphere Developer for zSeries (WebSphere Developer) 35

WebSphere Service Registry and Repository (WS-RR) 11

workbench 105

WS2LS

sample JCL 175

WS-Addressing 214

WS-Atomic Transaction 9

WSBIND

file 50, 138

WSBind 35

WSBIND and LOGFILE 175

WSDIR 37, 178

WS-Security 214

WS-Trust 10

X

XML 5

XML (xsd

any) 233

XML choice web service 297

XML constructs 235

XML elements 227

XML occurs web service 301

XML parsing 60

XML-binary Optimized Packaging (XOP) 10

XSDBind 221

XSL transformations (XSLT) 126

Z

z/OS 265

Communications Server IP CICS Socket Interface 70



Application Development for CICS Web Services



Overview of Web services in CICS updated for CICS TS 4.1

This IBM Redbooks publication focuses on developing Web service applications in CICS. It takes the broad view of developing and modernizing CICS applications for XML, Web services, SOAP, and SOA support, and lays out a reference architecture for developing these kinds of applications.

Experience using RDz for development

We start by discussing Web services in general, then review how CICS implements Web services. We offer an overview of different development approaches: bottom-up, top-down, and meet-in-the-middle. After laying out the foundations, we review the CICS catalog manager sample application, as this is the application we used as a basis.

New SOA patterns for CICS TS V4.1

We then look at how you would go about exposing a CICS application (namely, the catalog manager sample application) as a Web service provider, again looking at the different approaches. The book then steps through the process of creating a CICS Web service requester.

We close out by looking at CICS application aggregation (including 3270 applications) with Rational Application Developer for System z. The final chapter offers hints and tips to help you when implementing this technology.

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