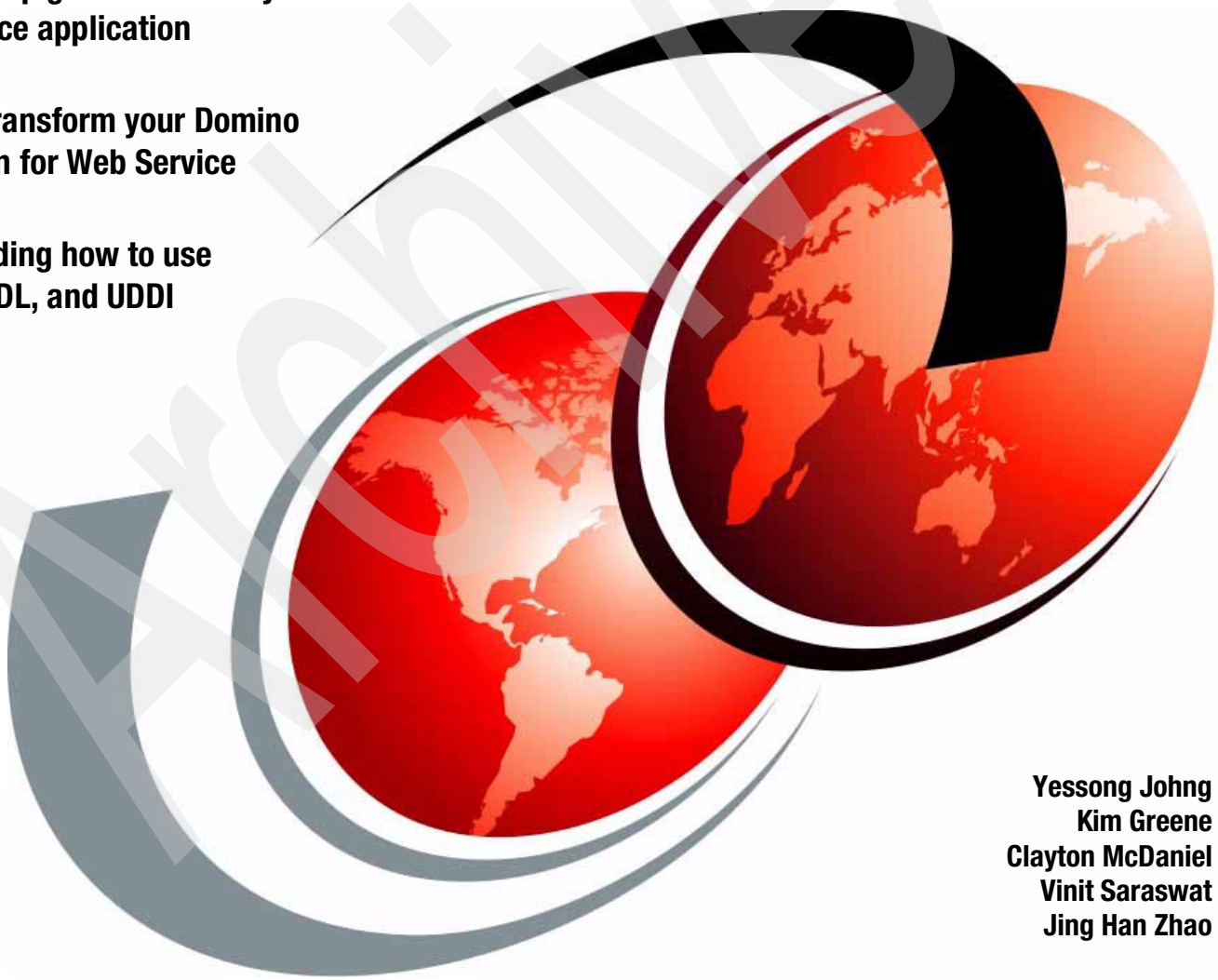
IBM

Domino Web Service Application Development

for the IBM eserver iSeries Server

- Step-by-step guide to create your first Web Service application
- Guide to transform your Domino application for Web Service
- Tips including how to use SOAP, WSDL, and UDDI



Yessong Johng
Kim Greene
Clayton McDaniel
Vinit Saraswat
Jing Han Zhao

Archive

Redbooks

ibm.com/redbooks

Archived

International Technical Support Organization

Domino Web Service Application Development for the IBM @server iSeries Server

November 2002

Archived

SG24-6862-00

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

Archived

First Edition (November 2002)

This edition applies to OS/400 V5R1 (product number 5722-SS1), IBM HTTP Server for iSeries V5R1 (5722-DG1), IBM Developer Kit for Java V5R1 (5722-JV1), WebSphere Application Server 4.02 (5733-WA4), Domino 5.0.10, and WebSphere Development Studio Client for iSeries 4.0.

© Copyright International Business Machines Corporation 2002. 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

Contents iii

Notices vii
Trademarks viii

Preface ix
The team that wrote this redbook. ix
Notice x
Become a published author x
Comments welcome. xi

Chapter 1. Introduction 1
1.1 Introduction to Web Services 2
1.1.1 What a Web Service is 2
1.1.2 Lotus' strategic plans for Domino and Web Services 5
1.2 Web Service development and deployment environment 6
1.2.1 HTTP Server 7
1.2.2 WebSphere Application Server 7
1.2.3 WDSc 7
1.2.4 Domino Server 8

Chapter 2. Writing my first Web Service application 9
2.1 Scenario description 10
2.1.1 Business view 10
2.1.2 Technology view 11
2.1.3 StockQuoteApp creation flow 12
2.2 Developing and deploying the applications 14
2.2.1 Create a new project in WDSc 14
2.2.2 Import StockQuoteService class file in Web Project 20
2.2.3 Create a new Web Service 24
2.2.4 Test the Web Service under a WDSc test environment 38
2.2.5 Setting up iSeries environment for deployment 40
2.2.6 Deployment of Web Service on WebSphere Application Server 43
2.2.7 Test the WebSphere Application Server deployment of the Web Service 49
2.3 Analysis of StockQuoteApp Web Service application 51
2.3.1 Provider Java program: StockQuoteService.java 51
2.3.2 Deployment Descriptor XML: DDS.XML 52
2.3.3 SOAP Server components: soapcfg.jar, xsd.bean.runtime.jar 53
2.3.4 WSDL documents 53
2.3.5 StockQuoteServiceProxy.java: Web requester Java client 55
2.3.6 Requester Programs: TestClient.jsp, Result.jsp 57
2.4 Tips and techniques 60

Chapter 3. Transforming a Domino application to a Web Service application 61
3.1 Existing Domino applications — before transformation. 62
3.2 Porting Domino application to Java application 63
3.2.1 Total transformation flow. 63
3.2.2 Porting tips 64
3.3 Deploying ported Java application using WDSc 68

© Copyright IBM Corp. 2002. All rights reserved.

3.3.1 ExpertiseLocatorApp creation — step-by-step 68
3.3.2 Tips on troubleshooting 92
3.4 Analysis of LWSEK 93
3.4.1 What you get from it 93
3.4.2 What we did with LWSEK 94
3.4.3 Applications we tested 95

Chapter 4. Consuming Web Services from a Domino application 97
4.1 Web Service consumption tutorial 98
4.1.1 Web Service consumer introduction 98
4.1.2 Basic requirements to be a Web Service consumer 99
4.1.3 What is SOAPConnect? 100
4.2 SOAPConnect TempfinderSample 101
4.2.1 Using TempfinderSample 101
4.2.2 Understanding TempfinderSample 102
4.2.3 TempfinderSample runtime flow summary 106
4.3 Consuming StockQuoteService using SOAPConnect 107
4.3.1 Customizing StockPortfolioSample 107
4.3.2 Testing your new consumer 112
4.3.3 Debugging tips 114
4.4 Consuming Web Services with a supplied proxy client 116
4.4.1 What is a proxy client? 117
4.4.2 StockQuoteServiceProxy class 117
4.4.3 Web Service consumer Java agent 119
4.5 Creating a proxy client from a WSDL file 124
4.5.1 What is a WSDL file? 124
4.5.2 Finding a WSDL file 126
4.5.3 Generating a proxy client class with WDSc 127
4.5.4 Using your new proxy client 142

Appendix A. Creating your own Web Service application 147
Scenario description 148
Business view 148
Technology view 148
InventoryAvailabilityService creation: step-by-step guide 152
Create a new project 152
Create InventoryAvailabilityService class in iWebLabSvc Project 156
Create a new Web Service 157
Test the Web Service under WDSc's WebSphere test environment 171
References 173
Tips and techniques 175

Appendix B. Using UDDI for both ends — publishing and requesting 177
What is UDDI's role in Web Service? 178
Business scenario 178
Technical scenario 178
Publication of Web Service using UDDI 179
Creation of Web Services client 192
Create Web Services client Web Project 192
Import the WSDL implementation files from UDDI Explorer 194
Create and test Web Services client 196
Tips and techniques 204

Archived

Related publications 205
IBM Redbooks 205
Other resources 205
Referenced Web sites 205
How to get IBM Redbooks 206
IBM Redbooks collections. 206

Index 207

Archived

Archived

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

© Copyright IBM Corp. 2002. All rights reserved.

Trademarks

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

alphaWorks®
AS/400®
DB2®
IBM®
IBM eServer™
iSeries™
OS/400®
PartnerWorld®
Redbooks(logo)™
SP™
WebSphere®

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

Domino Designer®
Domino™
K-station™
LearningSpace®
Lotus Discovery Server™
Lotus Knowledge Discovery System™
Lotus®
Notes®
QuickPlace™
Sametime®

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

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

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

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

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

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

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

Archived

Preface

“Contextual collaboration” is a new term whirling in the industry today. This phenomenal new type of business asset can be realized by two major mechanisms: the *components managing mechanism* and the *services delivery mechanism*. The component managing mechanism deals with calendar, workflow, messaging, e-mail, and so on, while the services delivery mechanism deals with receiving and responding to a request. In terms of IBM product offerings, we have Domino for the component managing mechanism and WebSphere for the services delivery mechanism.

We need standards to connect the applications from different business entities: your customers, your suppliers, your business partners, and you. These business entities may run their business applications on different platforms, different application languages, different networks, and so on. To connect these un-identical entities together, we need standards. For that, we have Web Service standards such as SOAP, WSDL, XML, and UDDI.

This IBM Redbook provides real application scenarios which illustrate how to transform existing Domino applications into either a Web Service provider or a Web Service requester. This is achieved and served through the integration of Domino and WebSphere Application Server.

The team that wrote this redbook

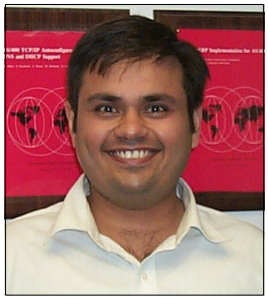
This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Yessong Johng is a Senior IT Specialist at the IBM International Technical Support Organization, Rochester Center. He started his IT career at IBM as a S/38 Systems Engineer in 1982 and has been with S/38, AS/400, and now iSeries for 20 years. He writes extensively and develops and teaches IBM classes worldwide in the areas of e-business on iSeries. His major responsibilities are Domino, WebSphere, and Java implementation on iSeries, focusing on Domino and WebSphere integration. His other areas of interest include VPN, TCP/IP, and SQL performance.

Kim Greene, president of Kim Greene Consulting, Inc., specializes in Domino for iSeries consulting, customized education, and performance. Kim has over four years of experience with Domino and 12 years of experience with the AS/400 and iSeries platform. Kim specializes in iSeries Domino performance analysis, system and application tuning, enterprise integration with back-end applications and data, and Java and Domino integration. Kim is also a frequent writer for technology magazines and speaks at many conferences worldwide. Previously she was employed at IBM where she worked in the PartnerWorld for Developers (PWD) organization helping IBM business partners incorporate Domino into their existing applications. Prior to working in PWD she worked on several areas of OS/400 performance in the AS/400 development laboratory.

© Copyright IBM Corp. 2002

Clayton McDaniel is an Advanced Technical Support IT Specialist at IBM in Rochester. He provides technical support to the internal IBM deployment of Domino on the iSeries. He has concentrated on developing various applications on Domino for two years. He holds a degree in Computer Science from Iowa State University of Science and Technology in Ames, Iowa. His main areas of expertise are Domino administration, application development, performance and architecture.



Vinit Saraswat is Consultant for Bridan Technologies. He holds a master's degree in Electronics from School of Electronics, DAVV, Indore, India and is a Sun Certified Java Programmer and IBM Certified WebSphere Specialist. He has six years of experience in the IT field, specializing in WebSphere Application Server and J2EE application development for the last two and a half years. His area of expertise also include design and development of e-business solutions for iSeries customers using Domino and WebSphere products.

Jing Han Zhao is a Technical Support Specialist for iSeries at IBM in China. She has three years experience on iSeries operating systems and architecture. She is responsible for technical support on iSeries, including system management, performance tuning, ERP and e-business solutions, especially Domino for iSeries. She also provides technical marketing support to the iSeries sales team, and holds technical training for customers and business partners.

Notice

This publication is intended to help LotusScript and Java applications developers and administrators to transform existing Domino applications into a Web Service provider or requester applications on the iSeries Server.

Become a published author

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

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  `ibm.com/redbooks`

- Send your comments in an Internet note to:

  `redbook@us.ibm.com`

- Mail your comments to:

- IBM Corporation, International Technical Support Organization
  Dept. JLU Building 107-2
  3605 Highway 52N
  Rochester, Minnesota 55901-7829

Archived

Archived

1

Introduction

This book is about Web Services and Domino Web Service applications on iSeries. We discuss these main subjects:

- What a Web Service is and how it is implemented on iSeries
- Real examples of Web Service applications on iSeries
- Transforming existing Domino applications into Web Service applications on iSeries

In this first chapter of the book, we also cover:

- An introduction to Web Services
- The development and deployment environment for the scenarios covered in the book

Archived

© Copyright IBM Corp. 2002. All rights reserved.

1.1 Introduction to Web Services

This section provides a brief introduction of Web Services. It also discusses Lotus' strategy for Domino related to Web Services.

1.1.1 What a Web Service is

A Web Service is a new technological framework that describes a standards-based architecture that provides the long awaited capability for systems to integrate with one another automatically. This common framework enables dynamic, Web-based system-to-system interaction. The incredibly powerful thing about Web Services is their accretive basis. Web Services add capabilities to applications and systems already in use. Incorporating Web Services into your existing applications does not mean re-architecting the existing software. Web Services is about changing the way your current applications are exposed.

Web Services are self describing, modular programs. They provide process to process interaction, there is no user interface involved in Web Services. The functionality is provided by back-end processing. Web Services can be thought of as encapsulating business operations. They provide a simple request-reply mechanism to full business process interactions. Web Services use methods and properties known as industry standards to externalize themselves and be consumed by others. These industry standards include XML, SOAP, WSDL, and UDDI.

There have been many attempts in the past to provide standards for application integration. In the 1990s EDI emerged and was used by organizations to force value chain integration. EDI imposed many constraints on the applications involved. They had to be adapted to the technology, and connect to and use the data in the manner prescribed by the owner of the data. Next came Web standards such as HTTP, HTML, XML, OBI, Rosetta Net, and cXML. The problem with these attempts at application integration was a lack of program-to-program integration. There was no application description, dynamic discovery and binding, transaction management, and workflow.

The Web Services model solves this problem by shifting the focus to integrating applications independent of platform, language, or data structure. This model eliminates the need for much of the system-to-system integration, easing the biggest trouble spot of integration, differing APIs.

As stated above, there are four main technologies that enable Web Services:

- **XML** (eXtensible Markup Language) - The universal language of Web Services
- **SOAP** (Simplified Object Access Protocol) - The remote procedure call (RPC) facility for Web Services
- **WSDL** (Web Services Description Language) - Allows Web Services-enabled systems to tell each other what capabilities they have, and how to programmatically interact with them
- **UDDI** (Universal Description, Discovery, and Integration) - An XML-based directory, or registry, for Web Services

These technologies are the core pieces of the Web Services development model. Developers use these pieces to create distributed, modular functions that connect applications together into useful and strategic processes.

The Web Services model shows how these pieces fit together.

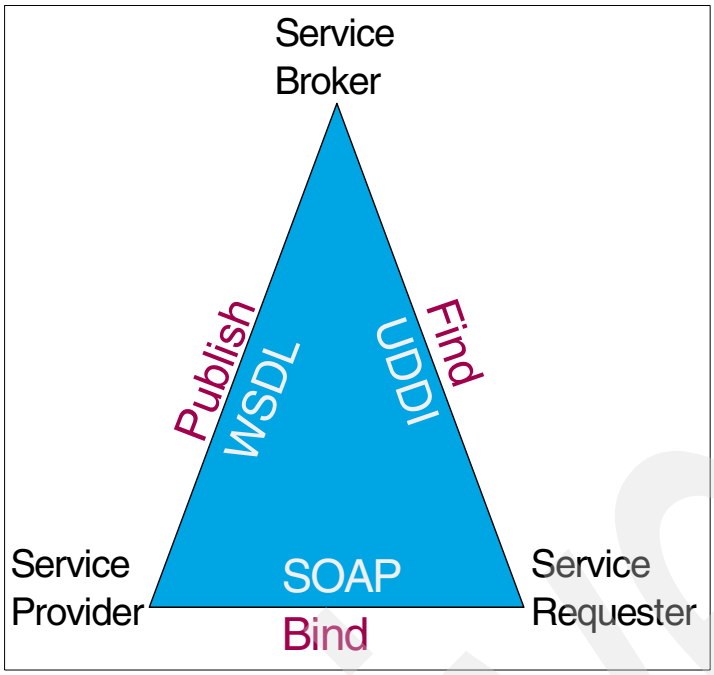


Figure 1-1 Web Services model

The Web Services model (Figure 1-1) shows three different corners, the service provider, the service broker, and the service requester. The service provider publishes their Web Service using a combination of WSDL and UDDI. The WSDL file is in XML format and contains information that describes the contents of a Web Service and what methods and properties are associated with the service. The Web Service is published in an external or internal registry using UDDI. Once the Web Service is publicly or privately available in the appropriate UDDI registry, the service requester uses UDDI to find the Web Service and consume it. SOAP is used to invoke a Web Service, therefore binding the service requester to the service provider. Remember, Web Services are process-to-process connections, they do not directly involve end-user interaction.

Understanding the components of Web Services

In this section we will delve into the four primary components of Web Services.

XML

XML is a markup language that provides the mechanism to define data. A markup language is used in many different scenarios. It is used by word processors to specify formatting and layout, by communications programs to express the meaning of data sent over the wire, by database applications that must associate meaning and relationships with the data they serve, and by multimedia processing programs which must express metadata about images or sound.

Because XML is a tagged language, the information necessary to communicate what the data means and/or what the receiver should do with the data is sent along with the data. Let's look at an XML example to provide you with a point of reference.

Example 1-1 A look at XML

```
<CONTACT>
<NAME>Kim Greene</NAME>
<ID>001</ID>
<COMPANY>Kim Greene Consulting, Inc.</COMPANY>
```

```
<EMAIL>kim@kimgreene.com</EMAIL>
<PHONE>507-367-2888</PHONE>
<STREET>307 12th Lane, SW</STREET>
<CITY>Oronoco</CITY>
<STATE>Minnesota</STATE>
<ZIP>55960</ZIP>
</CONTACT>

<CONTACT>
<NAME>Vinit Saraswat</NAME>
<ID>002</ID>
<COMPANY>Bridan Tech. Inc.</COMPANY>
<EMAIL>vinit_sarawat@hotmail.com</EMAIL>
<PHONE>201-556-6345</PHONE>
<STREET>149 Finnington Ave</STREET>
<CITY>Saddle Brook</CITY>
<STATE>New Jersey</STATE>
<ZIP>07663</ZIP>
</CONTACT>
```

In Example 1-1, we get a first look at XML. There are tags that tell us what components make up a Contact. The Name, ID, Company, Email, and so on are all data elements within a contact. The XML tags tell us information about the data, the syntax for marking up the data and the meaning behind the markup. The definition of the markup tags used in XML are handled by a Document Type Definition (DTD). The DTD communicates the structure of the markup language and defines what the valid tags are for the syntax markup. In time, a new type of definition mechanism of XML Schema is replacing DTD.

SOAP

SOAP can be thought of as an Internet-friendly alternative to DCOM, RMI, and CORBA/IIOP. It is simply a transport protocol. SOAP allows methods to be invoked against endpoints (URLs) over HTTP. SOAP method requests are transported in HTTP POST requests. A SOAP method is a request and an optional response. Both the request and the response are encoded as a serialized instance of a type. The type of the request is simply a <type> whose fields correspond to the in and in-out parameters of the method. Once the server-side operation has executed, an HTTP response message is returned to the client containing the results of the operation. There are no SOAP-specific HTTP response headers. The HTTP that is returned contains an XML document that contains the results of the operation.

WSDL

WSDL is a file containing XML syntax used to describe Web Services. It provides an interface to Web Services and makes their contents known. WSDL describes how a Web Service can be used and what functions it provides. The WSDL file lists or exposes the properties and methods for the programs that make up the Web Service.

UDDI

There are two components to UDDI, registration and discovery. UDDI is used to make Web Services available. UDDI is the mechanism for registering a Web Service in the directory, and publishing that it is available. UDDI is also used to find the right Web Service when it is needed.

Providing and using a Web Service

A Web Service can be either consumed and/or hosted. There are two main components to consuming a Web Service. These involve software to send requests and receive responses, and a SOAP engine to receive and manage the requests. Consuming a Web Service involves

calling or invoking the service and getting a response back. This response will contain the information requested from the program invoking the Web Service via parameters. The invoker can be either a program or another Web Service, because this is process-to-process communication. No end user interaction is required to consume a Web Service.

To provide a Web Service, it must be hosted. Hosting a Web Service has heavier requirements than consuming one. The host needs a SOAP engine to receive and manage requests. It also needs to be able to wrap responses in XML format and send the responses back. The hosting program needs to be long running, such as a servlet or Enterprise Java Bean (EJB). Typical hosts of Web Services today are WebSphere and Tomcat.

1.1.2 Lotus' strategic plans for Domino and Web Services

IBM's strategic plan is to produce software that is Web Service-enabled, ensuring customers will be able to extend the value of their IBM-based and Lotus-based systems and applications. Many of Lotus' products work with Web Services today via the APIs that each of these products provides.

A number of Lotus products can externalize Domino applications, such as Web Services with the proper enabling. The Domino Application Server hosts many multi-platform Web-based collaboration applications. These applications can be modified to provide SOAP interfaces and WSDL descriptions using XML to provide these applications as Web Services. Domino workflow applications can also be enhanced with SOAP interfaces and WSDL descriptions to create Web Services to allow external applications to use Domino Workflow-based applications over the Web.

The Lotus Knowledge Discovery System can also partake in Web Services today. The Lotus Knowledge Discovery System is comprised of two primary components: Lotus K-station and Lotus Discovery Server. Lotus K-station is a knowledge portal with out-of-the-box collaborative capabilities that provide a browser-like interface to many back-end data sources. The Lotus Discovery Server is a knowledge server that provides search and expertise location within an organization's relevant and collective experiences. Developers can add SOAP/WSDL interfaces over the expertise database that the Knowledge Discovery System builds, enabling that information to be served as a Web Service via any Web Services-enabled system.

The Lotus Sametime and QuickPlace products can add online awareness and teaming, as Web Services. By using Java APIs and Beans, developers can build Web Services that incorporate Sametime's online presence awareness and instant messaging capabilities as part of any Web Services-enabled system. The rich functionality of QuickPlace such as secure meeting places, threaded discussions, calendar and task management, content libraries, and customized modules can be exposed as Web Services as well through APIs.

The functionality of Lotus LearningSpace can also be externalized via Web Services by using the Java APIs and Beans to incorporate capabilities such as course lists and schedules and live or asynchronous courses.

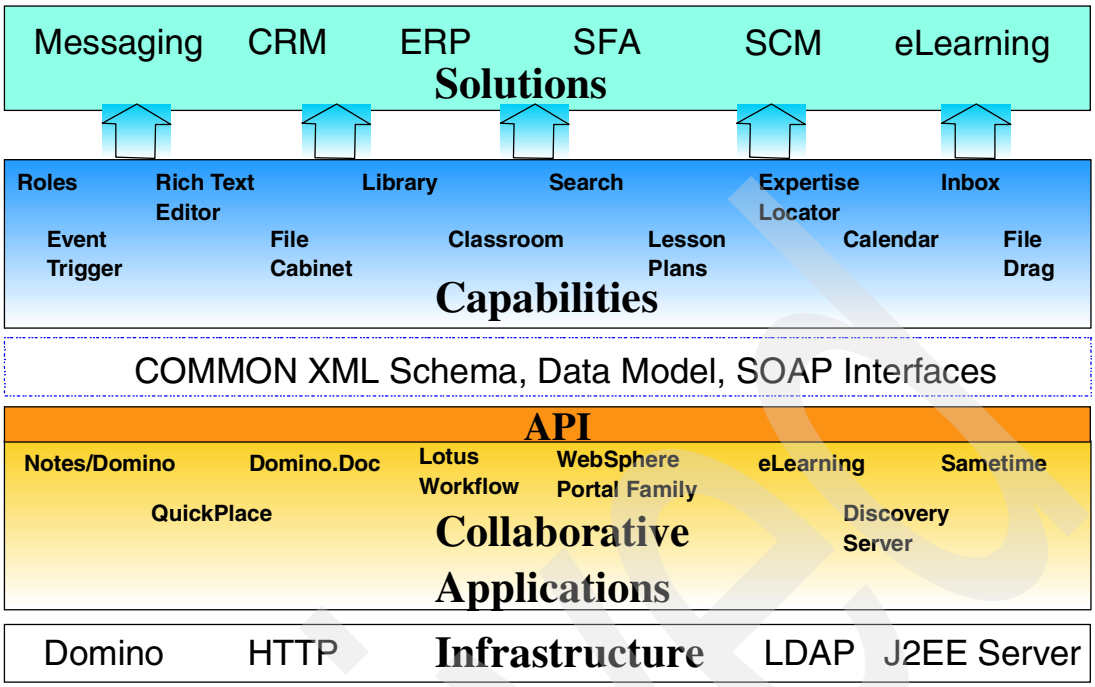


Figure 1-2 Lotus Web Services vision

Figure 1-2 provides a pictorial view of Lotus' strategy regarding Web Services. At the Infrastructure layer, the bottom layer of the figure, are many core, standards-based technologies including Domino, HTTP, LDAP, and J2EE. Above this infrastructure layer is the existing Domino product portfolio of collaboration applications. Many of these applications contain APIs that can be used today to work with and externalize these collaboration products into your application suites.

Lotus' strategic vision is to provide a common XML schema, data model, and SOAP interface to allow each of these collaborative applications to be used as Web Services. This new Web Services based interface will make it much easier to integrate the collaboration products with each other and with third party applications.

This in turn will allow the many collaborative capabilities of the Lotus Domino product family to be embedded in various third party and Domino solutions. As an example, the calendar function will be able to be easily integrated into a CRM application. Or an expertise locator function can be included in eLearning or ERM application. While these solutions can be provided today using the existing set of APIs for the underlying collaboration products, Lotus' strategic vision will ease this integration by providing a standards based interface into the Lotus product suite.

1.2 Web Service development and deployment environment

This section discusses the development and deployment environment we used while producing this redbook. Again, our focus is using WebSphere Application Server as a Web Service engine and Domino as a back-end application engine. Therefore, Java plays a middle layer language. Products we use, therefore, include:

- WebSphere Application Server (WAS) 4.02
  - Java-based Web applications server which includes Web Service, in other words, SOAP server engine

- WebSphere Development Studio Client for iSeries (WDSc) 4.0
  - Java-based Web applications development and deployment tools which include Web Service applications
- Domino 5.0.10
  - Back-end applications for both ends of a Web Service provider side and requester side

1.2.1 HTTP Server

There are two versions of HTTP Server that we used. The primary purpose of this redbook is demonstrating how to transform the existing Domino applications into either a Web Service provider application or Web Service requester application. We thought it may help the readers if we start the technical discussion by showing how a generic Web Service (without Domino integration) applications work.

We used two versions of HTTP Servers available on the iSeries respectively:

- For a generic Web Service applications scenario, we used IBM HTTP Server powered by Apache
- For the Domino integrated Web Service applications scenario, we used Domino HTTP Server

1.2.2 WebSphere Application Server

WebSphere is a set of software products that helps customers develop and manage high-performance Web sites and integrate those Web sites with new or existing non-Web business information systems. The WebSphere family consists of WebSphere Application Server and other WebSphere family software that is tightly integrated with WebSphere Application Server and enhances its performance.

WebSphere Application Server is fully compliant with Sun Microsystem's Java 2, Enterprise Edition (J2EE TM) standard and supports other common industry standards such as CORBA, XA, Secure Sockets Layer (SSL) and Lighweight Directory access protocol (LDAP).

In this redbook we have used WebSphere Application Server Advance Edition 4.0.2. The WebSphere Application Server Advanced Edition combines the portability of server-side business applications with performance and manageability of Java technologies to offer a comprehensive platform for designing Java-based Web applications. It provides support for distributed server environments and includes the underlying services for managing these environments. It also offers close integration with non-Web business systems. You can obtain more information about various flavors of WebSphere Application Server by visiting the Web site:

http://www.ibm.com/websphere

1.2.3 WDSc

You can quickly develop and deploy traditional and e-business applications on your iSeries system with IBM WebSphere Development Studio Client for iSeries (WDSc). This powerful suite of tools represents the next generation of WebSphere development tools for iSeries and is the client component of the IBM WebSphere Development Studio product, which contains all of the host ILE compilers for RPG, C, C++, COBOL, and the Application Development ToolSet. WDSc for iSeries is comparable to WebSphere Studio Application Developer (WSAD), but it differs from it in terms of added features of application development for iSeries. It has iSeries specific Java classes and connection interfaces. Most of the steps executed in this redbook with WDSc can also be done on WSAD.

The version of WDSc we used in this redbook was 4.0. We used this tool to create, test and deploy a Web Service in a WebSphere environment. WDSc served its purpose as a powerful tool in the case of Web Service creation because of its following salient features:

- WDSc has user friendly features which help you create a Web Service out of a simple Java Bean class. In this tool you can easily define your parameters for your Web Service such as:
  - URI of Web Service - Uniform Resource Identifier
  - Java to XML and vice versa mappings - For creation of Apache SOAP envelope
  - Settings for sample test client and proxy - The built-in tool support creates proxy client and sample code to test the Web Service.
  - WebSphere Test Environment - This environment helps to test the Web Service without the need to deploy it to the WebSphere Application Server.
- WDSc packages the Web Service for its easy deployment on WebSphere Application Server. By using WDSc for iSeries you can eliminate the dependency to AAT (Application Assembly Tool) from WebSphere Application Server.
- WDSc automates the process of Web Service publication. UDDI (Universal Description Discovery and Integration) explorer in WDSc helps you to discover a Web Service and create a Web Service client. Apart from that it helps you publish your Web Service so that it can be located by other businesses.

1.2.4 Domino Server

The Domino integrated Web Service application examples in this redbook use Domino as the HTTP broker for the Web Service transactions. For this project we chose to use **Lotus Domino 5.0.10**. We assume that you already have the Domino Server code installed on your iSeries and you have already configured a Domino server to work with. If you need help in this area, you can refer to the redbook *Lotus Domino for AS/400 R5: Implementation*, SG24-5592.

Archived

2

Writing my first Web Service application

This chapter describes how to write a Web Service application. Again, what we show here is a generic Web Service application scenario on the iSeries server. If you are already familiar with this subject and want to know how to transform a Domino application into a Web Service application, please skip to the next chapter.

Here we chose to describe the StockQuoteService Web Service application. You will develop a running model of StockQuote Web Service in this chapter.

This chapter describes the following:

- Business benefits
  - How to provide commonly required portal type services without maintaining your own applications. For example:

    You are a local newspaper Web site builder and you want to provide current stock quote service to the subscribers of your portal site. And you want to do this without maintaining your own application to provide the service. In other words, you want to use a Web Service provided by someone else out there.
- Technology and product components
  - Web Service standards
    - SOAP
    - WSDL
    - XML
  - Tools for creation of a Web Service
    - WDSc
    - WAS

© Copyright IBM Corp. 2002. All rights reserved.

2.1 Scenario description

VCK Web Service Inc. is a fictitious company we created for this redbook project. VCK Web Service Inc. provides Web Services over the Internet and charges 50 cents per transaction served. In this scenario, we will demonstrate how to transform an existing Java application of StockQuote into a Web service application: StockQuoteApp.

Note: Our example application uses yet someone else's, XMLToday's, service to get the stock quote. If you want to see the example of a stand-alone Java application being a Web Service provider application, please refer to Appendix A, "Creating your own Web Service application" on page 147.

2.1.1 Business view

The ever present business requirement in today's fast changing world is needed to address the portability and availability of time tested applications. These business applications reside in enterprise iSeries servers on legacy code. They should be available as service on the Internet. The business application should provide services with minimum portability or coding efforts. "Web Service" is the answer to this business need.

Business problems include:

- Satisfying dynamically increasing diversified needs of portal users
- Ever increasing cost of developing and maintaining the applications to meet those needs
- Ever increasing cost of dynamistic character and diversity of Web applications

Let's say there's a local newspaper company called Rochester Times (yet another fictitious company) that needs to provide stock quotes to its customers on demand. Rochester Times does not want to build and maintain their own application for providing this service. So they contacted a Web Service brokerage company, VCK Inc., which agreed to create a Web Service for Rochester Times. In turn, VCK Inc. uses XMLToday's stockquote URL. This way VCK Inc. takes responsibility to customize their Web service according to XMLToday's underlying application and provides a worry free uninterrupted service to Rochester Times. Rochester Times' application only needs to communicate with VCK Inc.'s Web Service application. What VCK Inc. does with XMLToday does not concern Rochester Times at all.

Archive

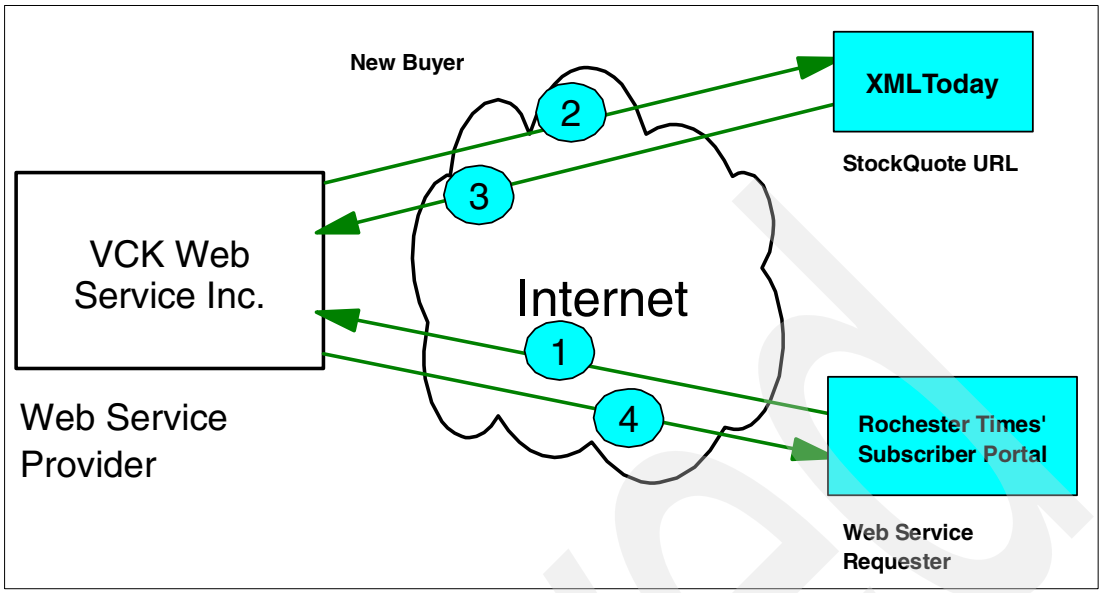


Figure 2-1 Business view of StockQuoteApp Web Service

Figure 2-1 illustrates the business view of the StockQuoteApp Web Service application.

1. Subscriber of Rochester Times clicks the 'StockQuote' portlet to check on any particular stock's current price. This will be formatted as a Web Service request and sent to VCK Inc.'s Web Service server.
2. VCK Inc.'s Web Service provider program will send the request (this is *not* a Web Service request: just a URL request) to XMLToday.
3. XMLToday responds with the quote or requested stock.
4. VCK Inc.'s Web Service provider program will format the Web Service response message embedding quoted stock price, a response from XMLToday, and send it to Rochester Times. At this point, the subscriber will see quoted stock price on his portal.

2.1.2 Technology view

Let's map this business view with technology view.

Archive

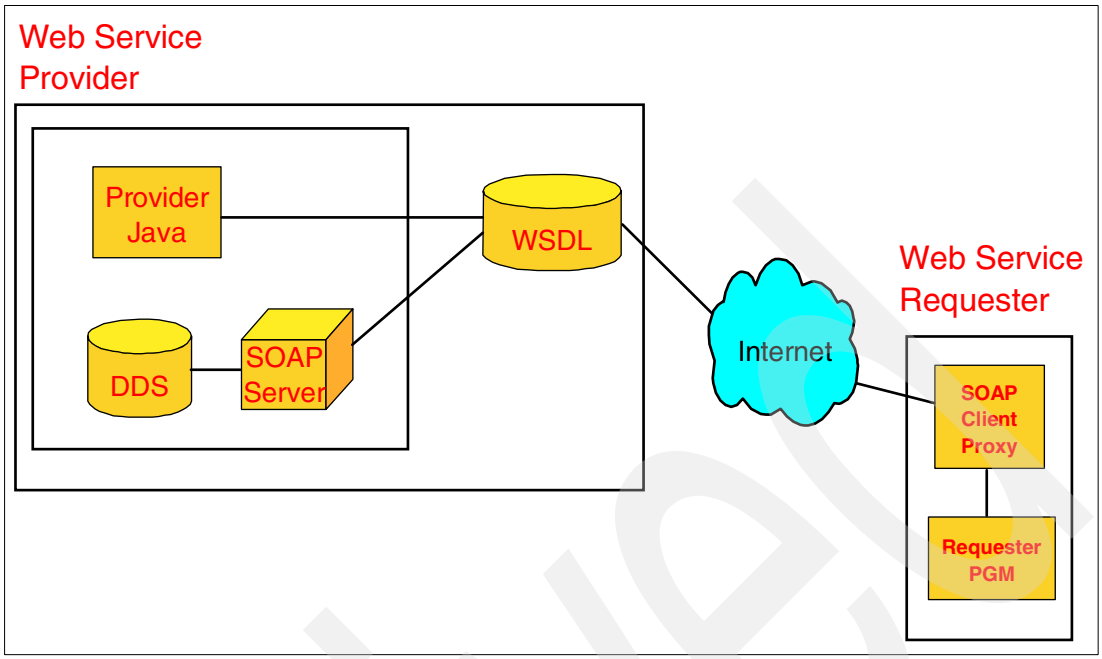


Figure 2-2 Technology view of StockQuoteApp Web Service

Figure 2-2 illustrates the technology view of StockQuoteApp Web Service application. If you map this with the previous figure, Figure 2-1, you will find one difference. Where is XMLToday? We removed that portion from technology view because it has nothing to do with Web Service architecture. Our provider Java program just happens to retrieve quoted stock price from that URL site instead of maintaining the data ourselves. For instance, that portion can be replaced our own provider Java program accessing the local database.

Now we will analyze this technology view deeper.

2.1.3 StockQuoteApp creation flow

Section 2.2, “Developing and deploying the applications” on page 14 which is immediately following this section will walk you through the detailed step-by-step guideline. Before we do that, let us give you a big picture of the application creation flow. Figure 2-3 illustrates the first task of the Web Service creation: creating a provider Java program.

Archive

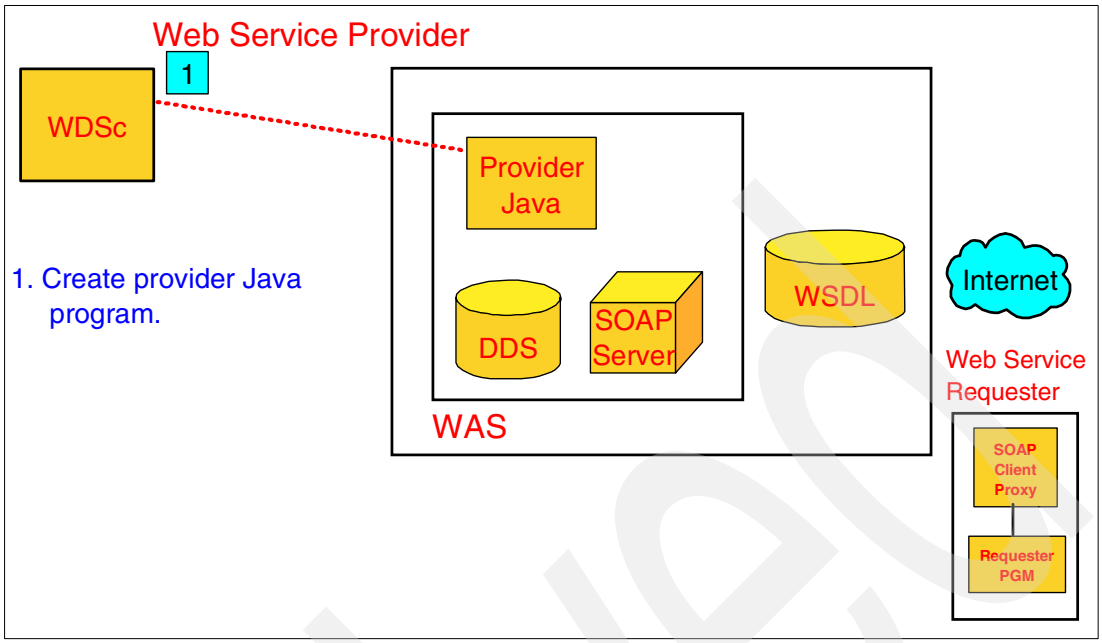


Figure 2-3 StockQuoteApp creation flow 1 of 3

Once the provider Java program is created, the next task is creating related resources to make this Java program a true Web Service provider application. This task is illustrated by Figure 2-4. Using the wizards provided by WDSc greatly simplifies this task.

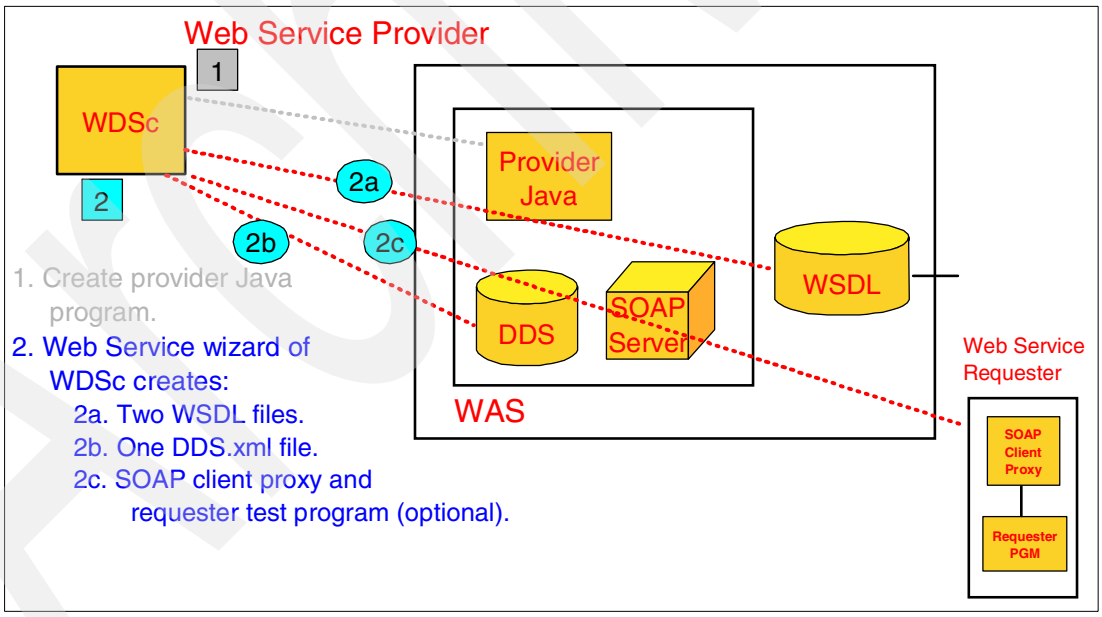


Figure 2-4 StockQuoteApp creation flow 2 of 3

This task consists of three sub tasks:

- First, we create two WSDL files.
- Then, we create a DDS.xml file.
- Optionally, we can create a SOAP client proxy and requester test program. The Web Service requester can download these two to their system, later at run time. This can greatly simply their tasks of creating a Web Service requester program.

Figure 2-5 illustrates the final task: deploying the Web Service provider application to the Web application server, which is WebSphere Application Server in our case.

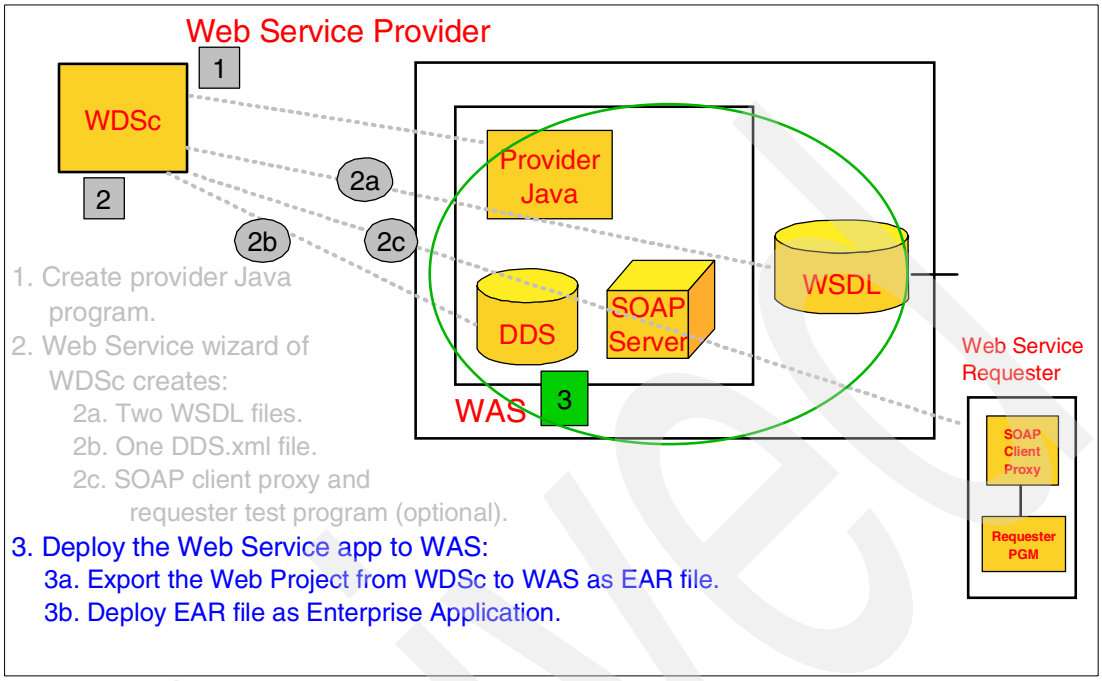


Figure 2-5 StockQuoteApp creation flow 3 of 3

We will now walk you through the whole cycle of the creation on the following section in greater detail.

2.2 Developing and deploying the applications

In this section, we take you through, step-by-step, the actual development and deploying process of our Web Service application, Stock Quote Application. Again, we are using WDSc as a development tool and WebSphere Application Server as our server system.

2.2.1 Create a new project in WDSc

The first thing to do to create new Web Service project is to create a Web Project in WDSc.

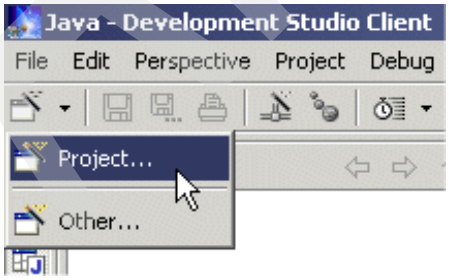


Figure 2-6 Create new project in WDSc

1. In WDSc, as illustrated in Figure 2-6, click **File -> New -> Project -> Web Project**.

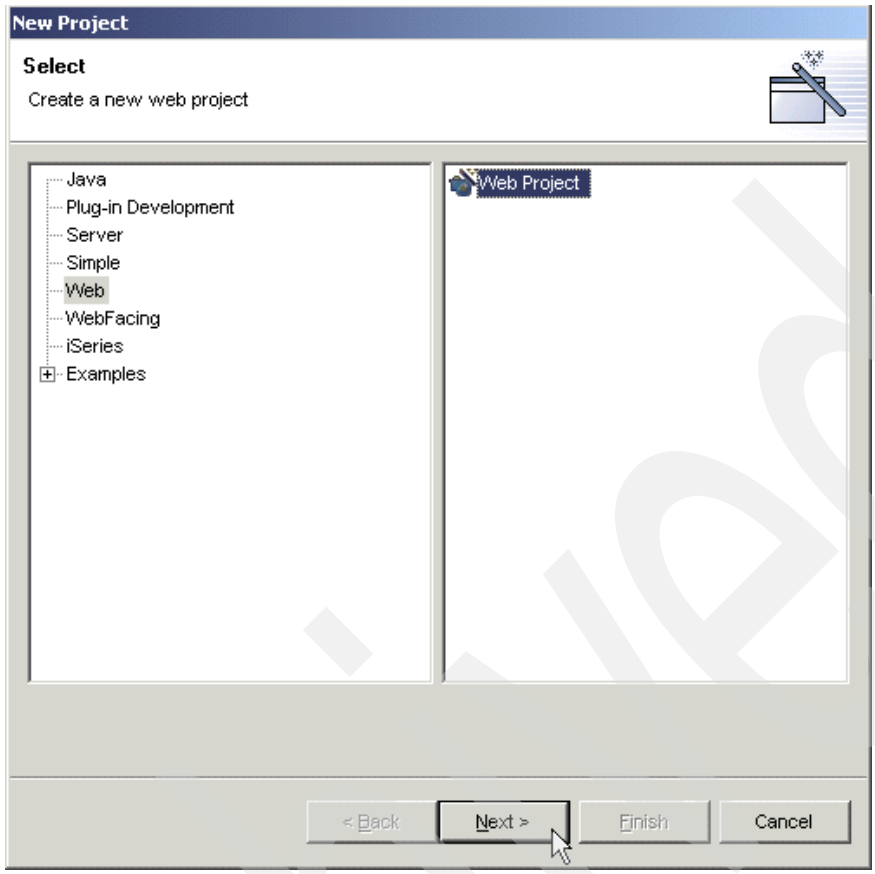


Figure 2-7 Select New Project type "Web Project" in WDSc

2. From the selection menu for new project, select **Web** from the left pane, then select **Web Project** from the right pane as shown in Figure 2-7. Click **Next**.

Archived

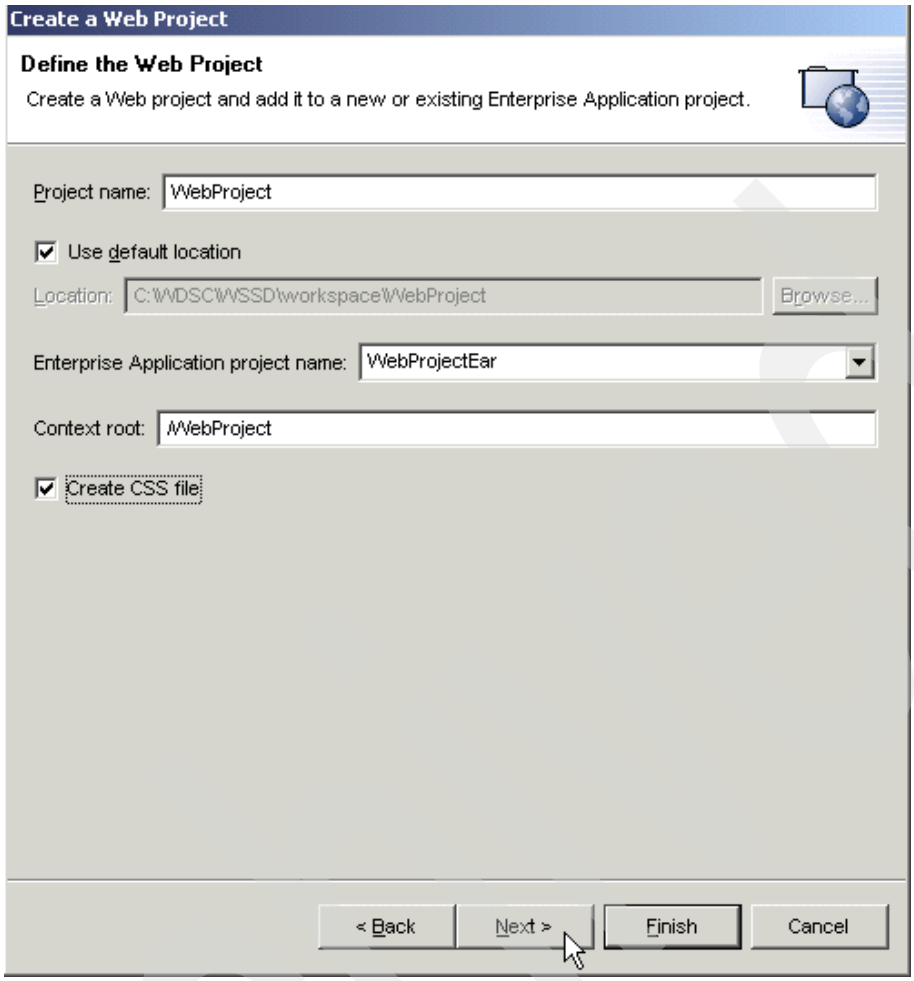


Figure 2-8 Specifying project name, context root and EAR project file in WDSc

3. Fill in the next window as shown in Figure 2-8:
   - For **Project Name**, enter **WebProject**.
   - For **Enterprise Application Project Name**, enter **WebProjectEar**.
   - For **Context Root**, enter **/WebProject**.

   Note: Context root has to be entered with **"/"** because this will map to the directory path in your Web service when you deploy it on the WebSphere Application Server. Context root will make up the URL of the Web Service.

   - Check the boxes for **use default location** and **create CSS file**.

   Click **Next**.

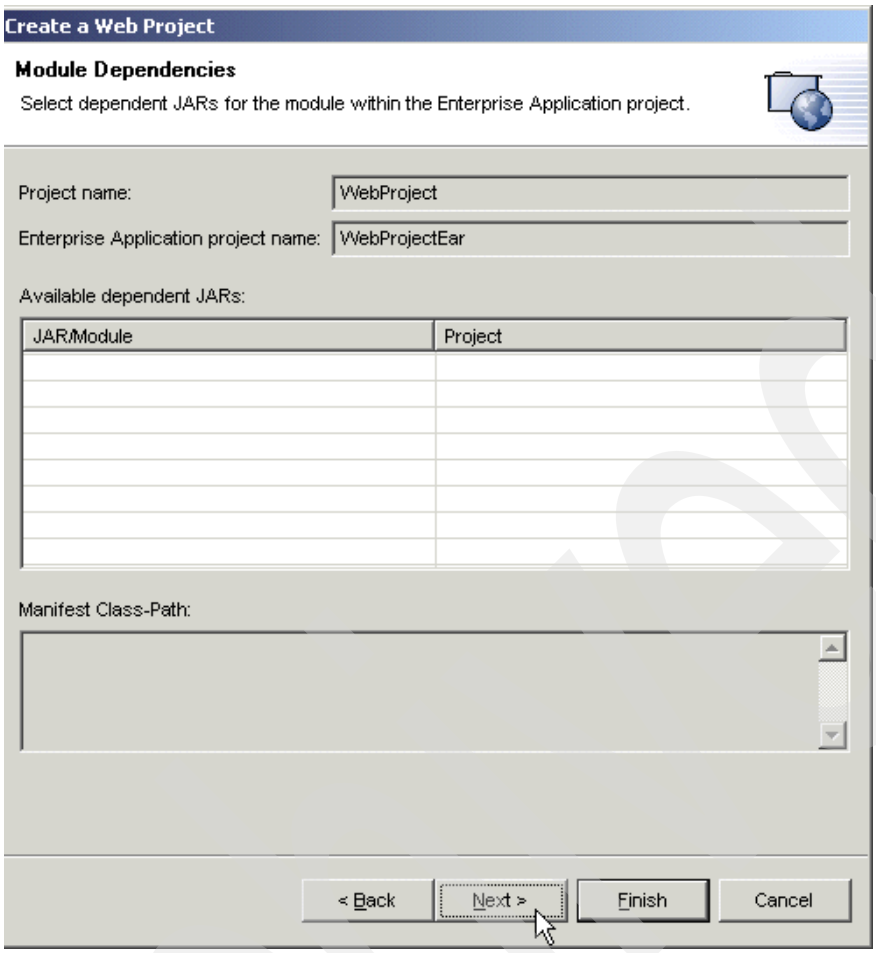


Figure 2-9 Specifying Module Dependencies on WDSc

4. In the Module Dependencies selection window make sure that the Project name and Enterprise Application project names are correct as shown in Figure 2-9.

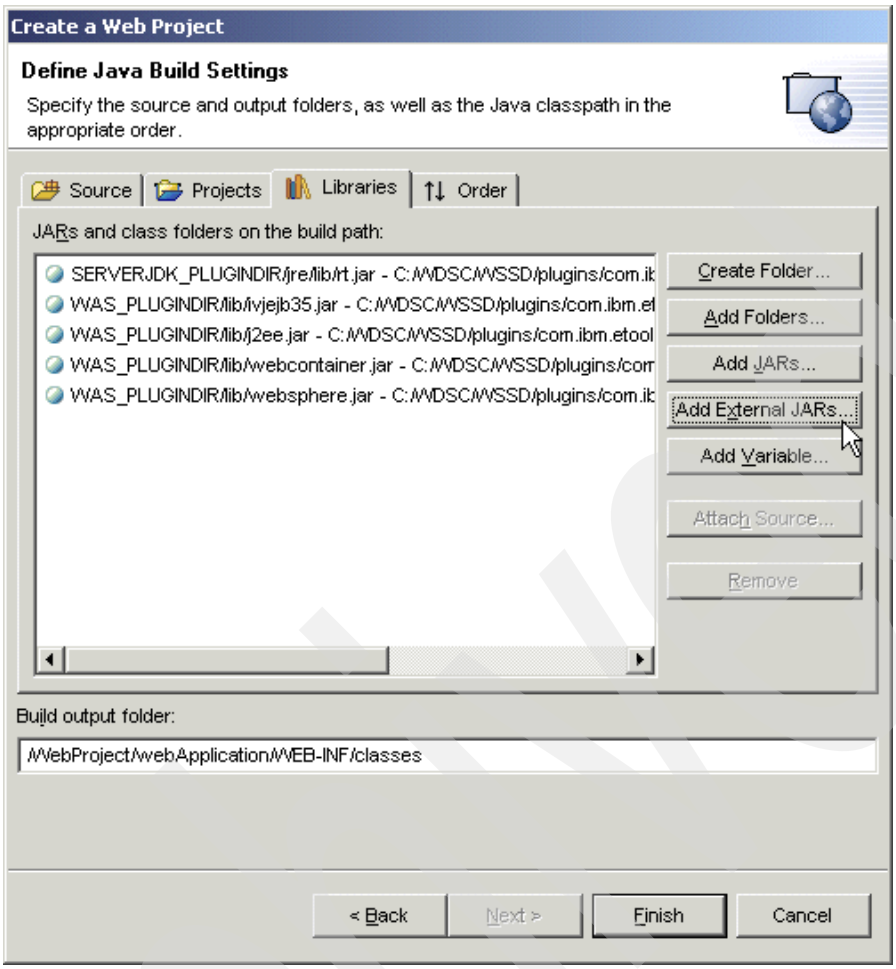


Figure 2-10 Adding External Jars to Web Project classpath in WDSc

5. To add dependant jar files for classes, click **Libraries -> Add External JARs** as shown in Figure 2-10.

Archive

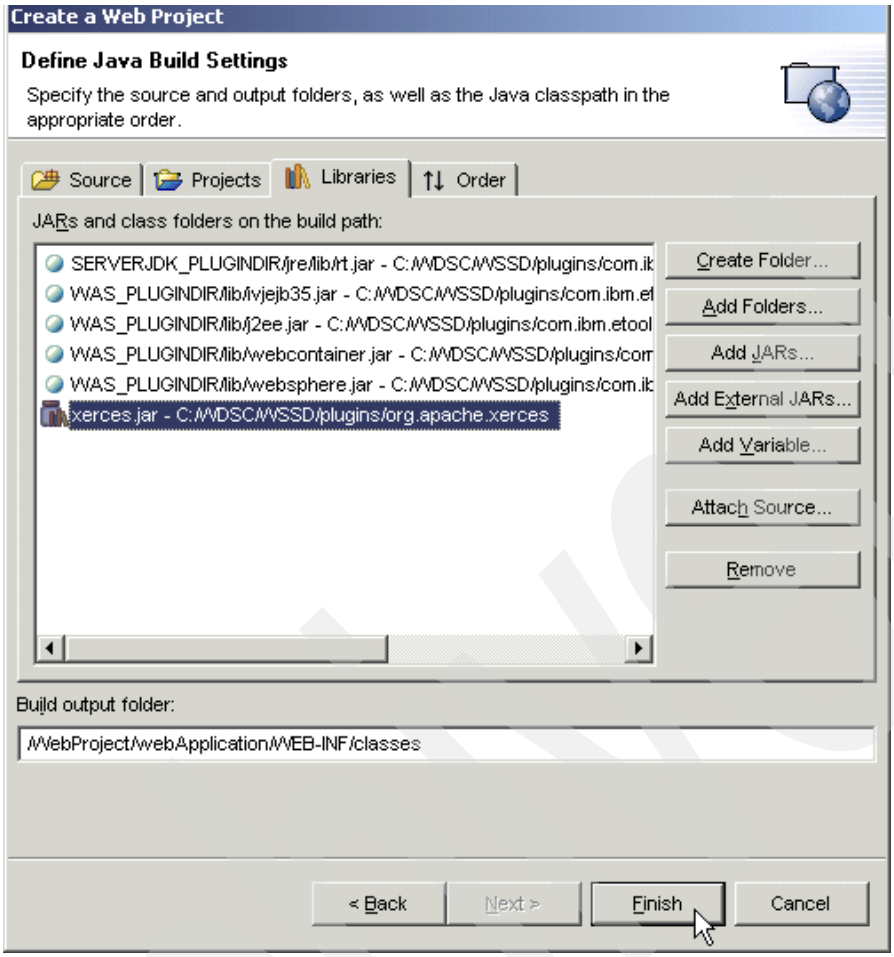


Figure 2-11 Add Xerces.jar file from WDSc plug-ins directory

6. Refer to Figure 2-11 and select **C:\WDSc\WSSD\Plugins\org.apache.xerces\xerces.jar** to be added as an external jar in the Web Project. You should likewise import other jar files such as **xml4j.jar** and **soap.jar** files.These jar files are found on your **WDSc\WSSD\Plugins** directory. Click **Finish** to create the Web project.

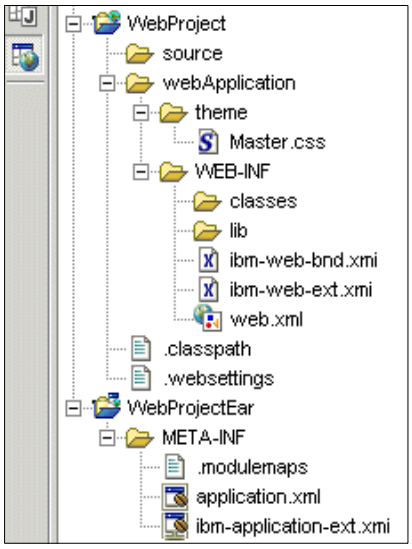


Figure 2-12 Directory Structure of WebProject in WDSc

7. After creation of the Web Project, click **Perspective -> Open -> Web** and expand the Web Project, which we just created, to look at the directory structure as depicted in Figure 2-12.

2.2.2 Import StockQuoteService class file in Web Project

The next task is writing actual Java programs to provide the application service for Web Service applications. In our case, the Java program is already provided with a name like StockQuoteService.

Figure 2-13 Source folder in WebProject directory in WDSc

1. In the source folder of Web Project we will import our source file for creating a Web Service which is highlighted in Figure 2-13.

Archived

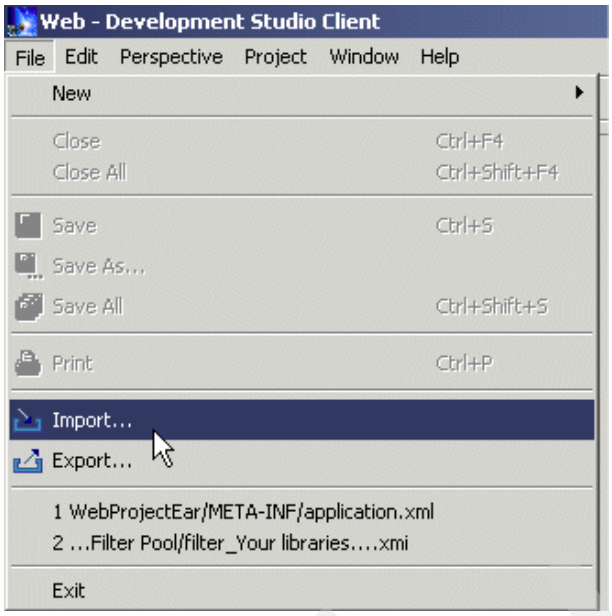


Figure 2-14 Import source files in WebProject source folder in WDSc

2. Highlight source folder in WebProject as shown in Figure 2-13, then click **File > Import** as shown in Figure 2-14.

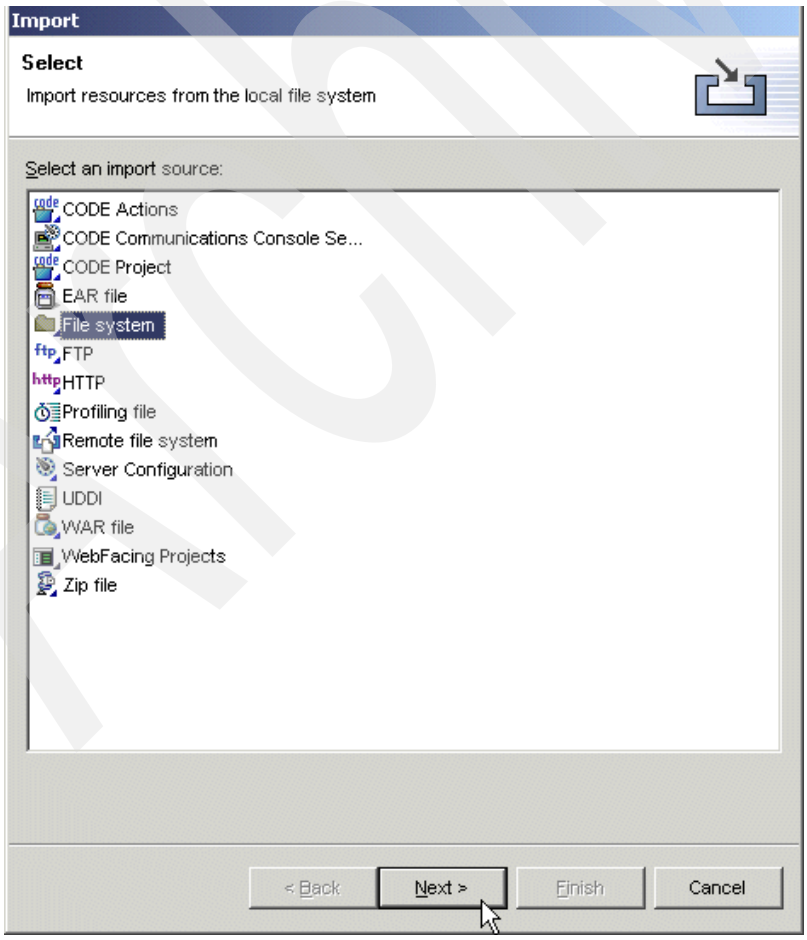


Figure 2-15 Import from File System in WDSc

3. As shown in Figure 2-15, select **File System** as the import source to import class files in WebProject source folder.

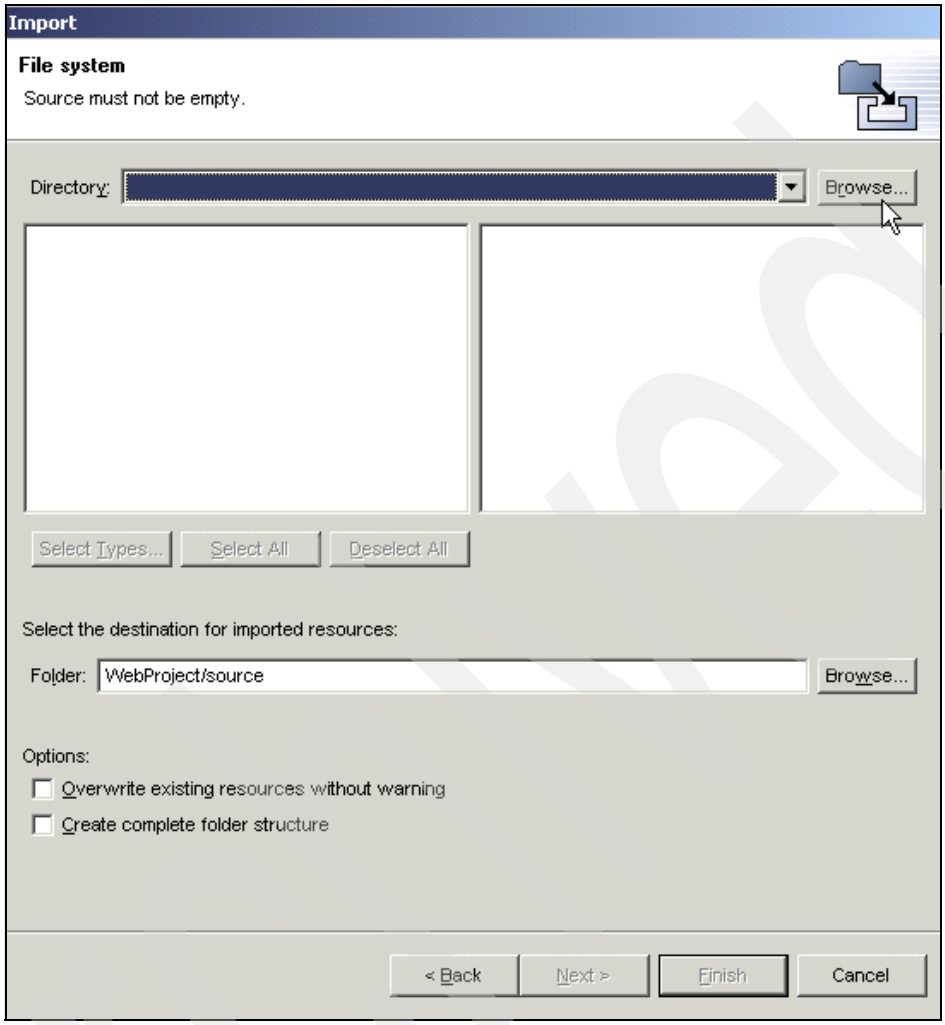


Figure 2-16 Browse to folder to select required source files to import in WDSc

4. As shown in Figure 2-16, click **Browse** and select the appropriate folder to bring in the Java bean source file which will create a Web Service.

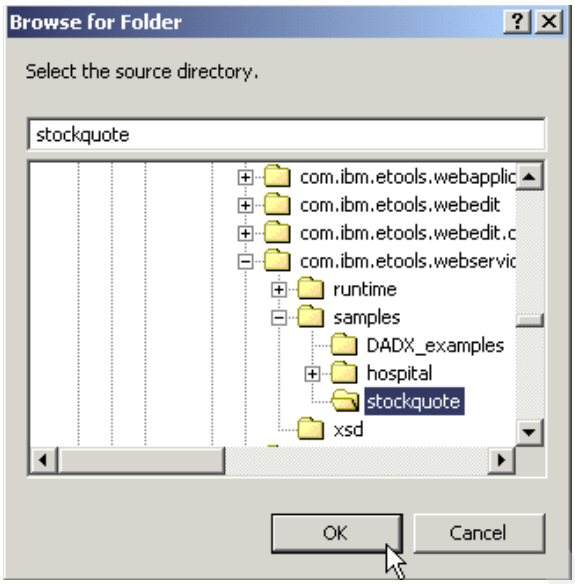


Figure 2-17 StockQuote sample source from WDSc plug-ins folder

5. As shown in Figure 2-17, select:
   C:\WDSC\WSSD\plugins\com.ibm.etools.webservice\samples\stockquote
   to import source file **StockQuoteService.java**.

Archived

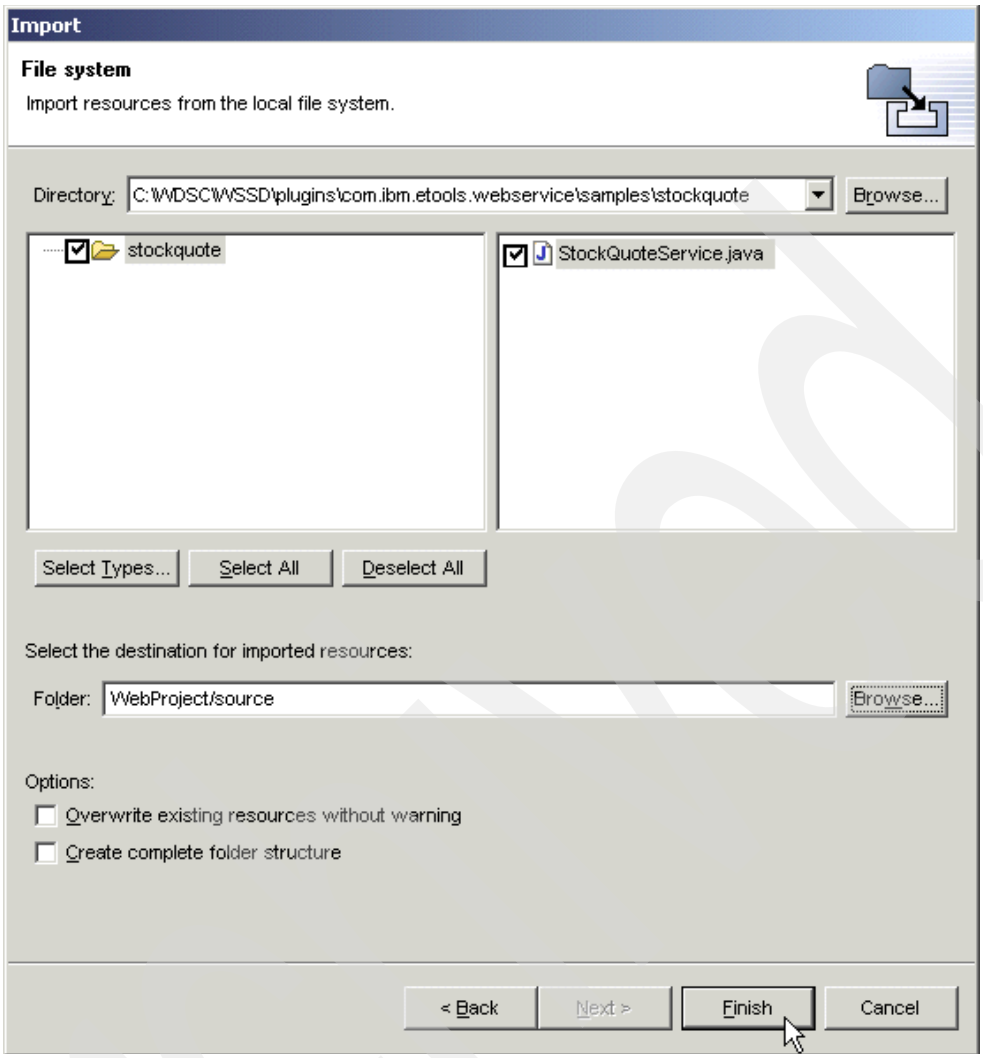


Figure 2-18 Check on StockQuoteService.java file to be imported in WDSc

6. As shown in Figure 2-18, click the **stockquote** folder to select the **StockQuoteService.java** source file for import in the WDSc folder **WebProject/source**. Click **Finish** to import the source file.

2.2.3 Create a new Web Service

Now we have created a new Web Project in WDSc, have imported actual Java programs within it, and are ready to create a new Web Service.

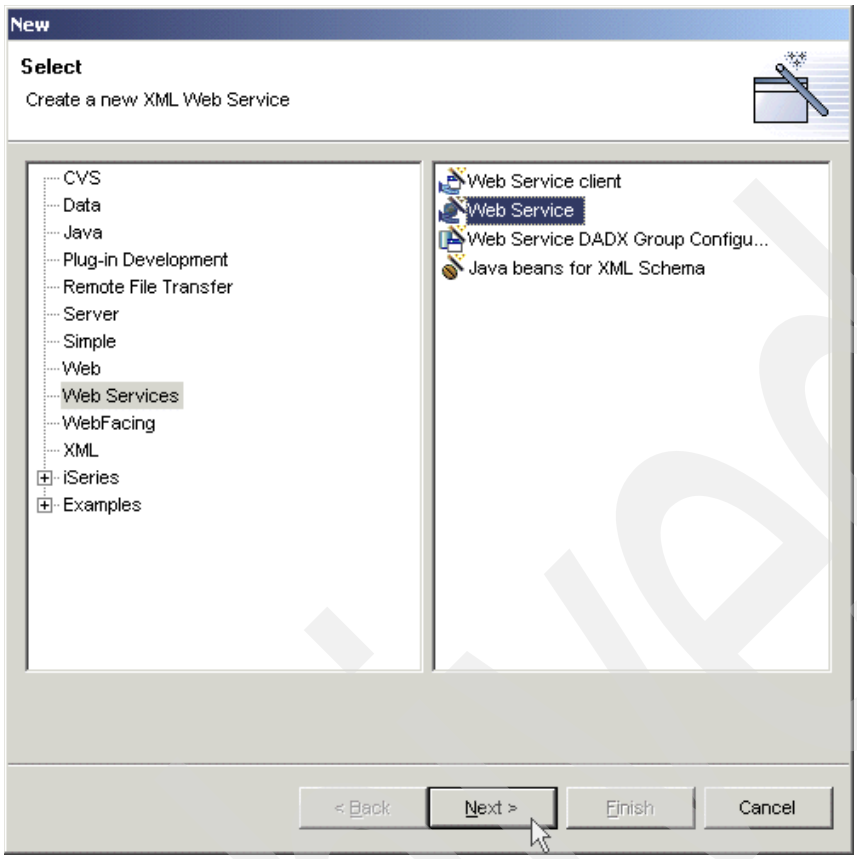


Figure 2-19 Create Web Service from StockQuoteService.java in WDSc

1. As shown in Figure 2-19, click **File -> New -> Other -> Web Services -> Web Service** to create a Web service. Click **Next** to go to the Web Service options page.

Archived

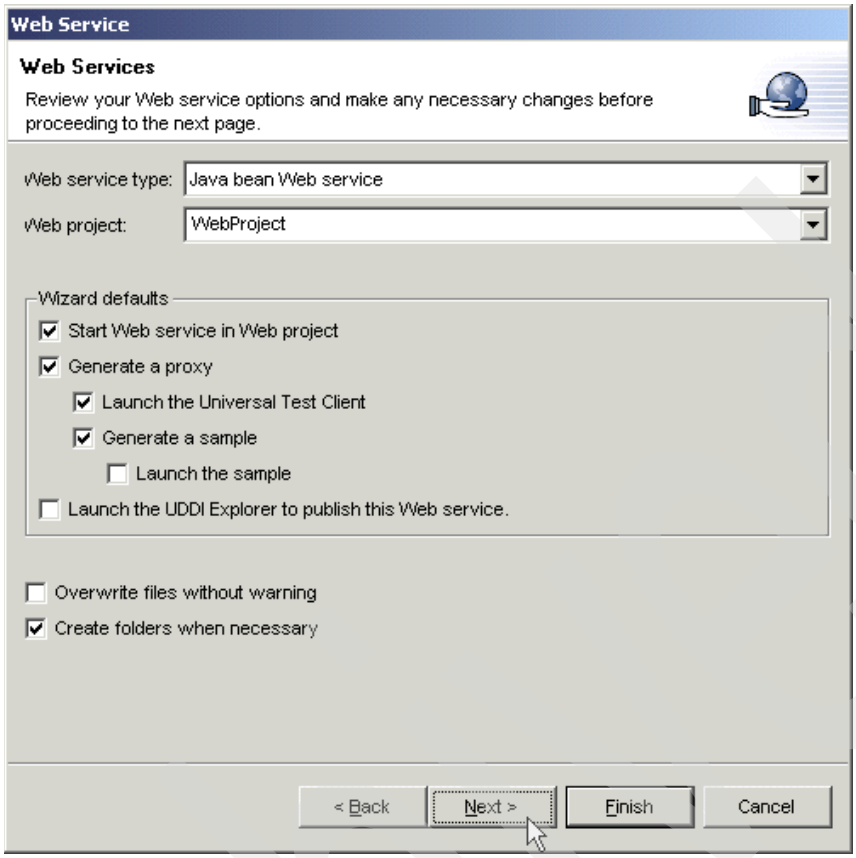


Figure 2-20 Web Service Create options in WDSc

2. As given in Figure 2-20, the Web Service create options page, make sure **Web service type** is **Java Bean Web Service**. Also the **Web project** name should be correct, and in our case, it is **WebProject**. Select options:
   - **Wizard defaults**
     - **Start Web service in Web project**
     - **Generate Proxy - Launch Universal Test Client**, **Generate the sample**.
   - Also check on **Create folders when necessary**.

   Leave the other options unchecked and click **Next**.

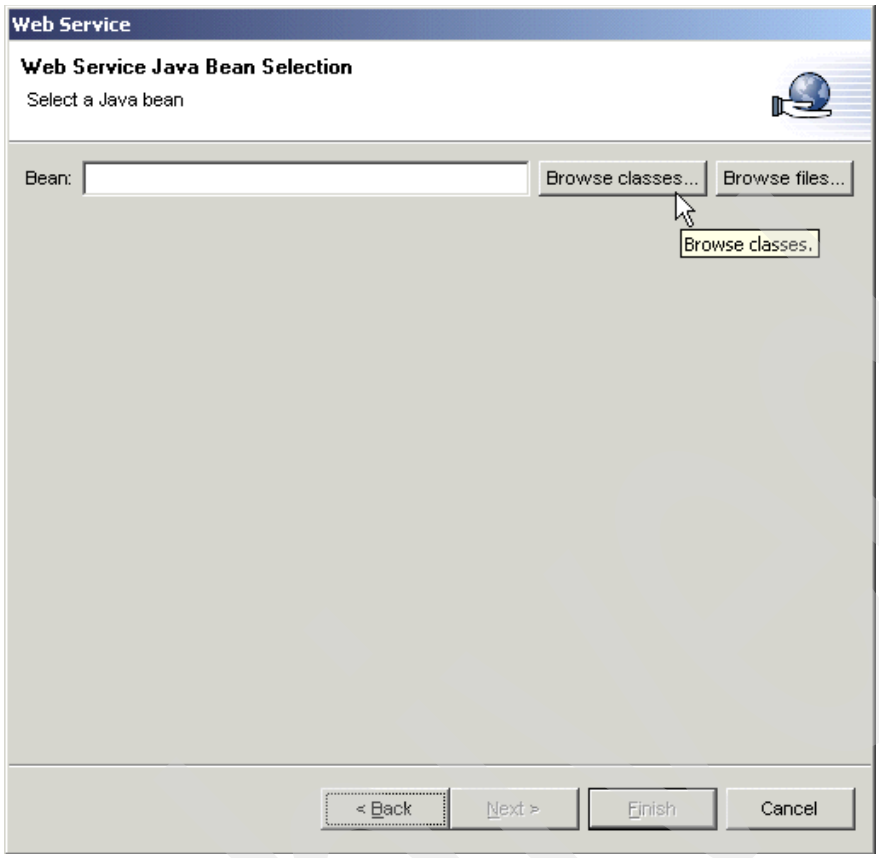


Figure 2-21 Select Java bean on which Web Service will be built in WDSc

3. Every Web Service must have a service component which provides the core functionality of the Web Service. In our StockQuoteApp example, the Java bean StockQuoteService will encapsulate the core functionality of the Web Service. Here on the Java Bean selection option we browse to select StockQuoteService Java bean. As shown in Figure 2-21, click **Browser Classes** to make a selection.

Figure 2-22 Select StockQuoteService Java bean in browser classes on WDSc

4. As shown in Figure 2-22, enter **StockQuoteService** Java bean name as browser classes.

Archived

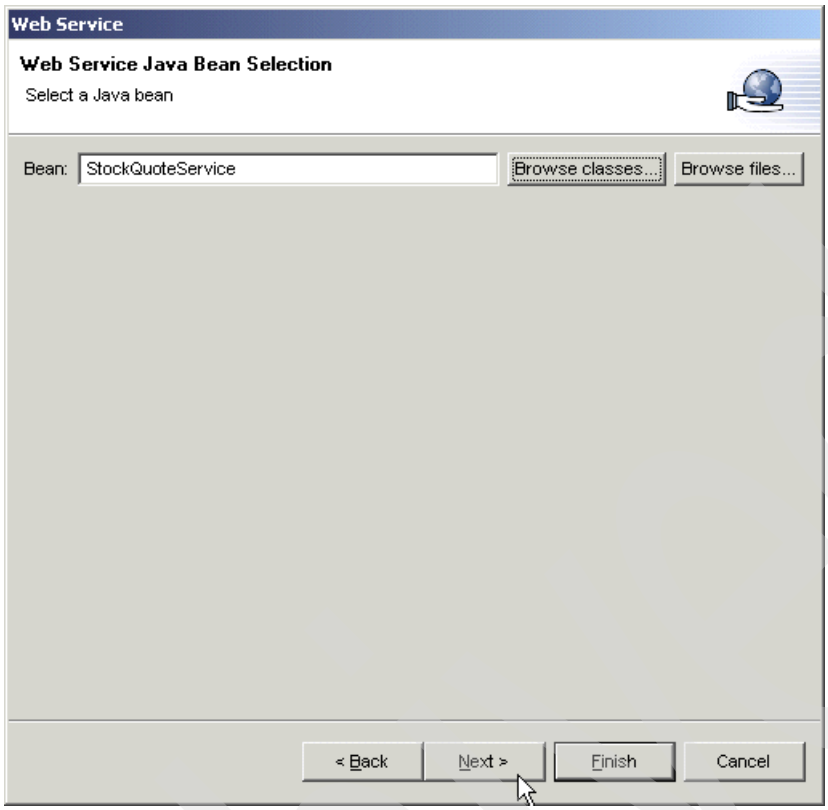


Figure 2-23 Select StockQuoteService after selection from Java Bean browser in WDSc

5. As shown in Figure 2-23, you will see it appear as the Bean name on the option after the selection from class browser. Click **Next**.

Archive

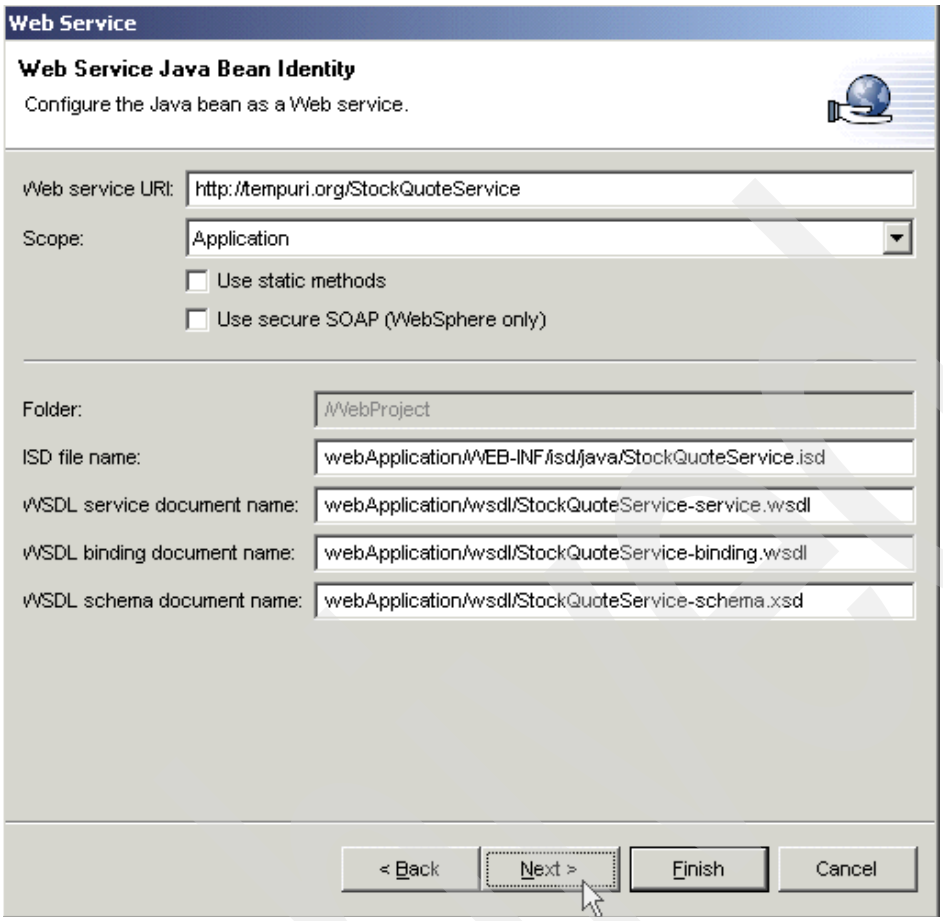


Figure 2-24 Specify Web Service Java Bean identity on WDSc

6. As depicted here in Figure 2-24, on the option window of Web Service Java Bean Identity, select a name for **Web Service URI**. This URI can be in the form of URL **http://tempuri.org/StockQuoteService** or it can be **urn:StockQuoteService**. They are both common notation and later identified to Web Service components like a proxy client as XML namespace representation.You will also notice a few WSDL files here. These define the framework of StockQuote Web Service in XML files such as StockQuoteService-service.wsdl. Leave these options as defaults and click **Next**.

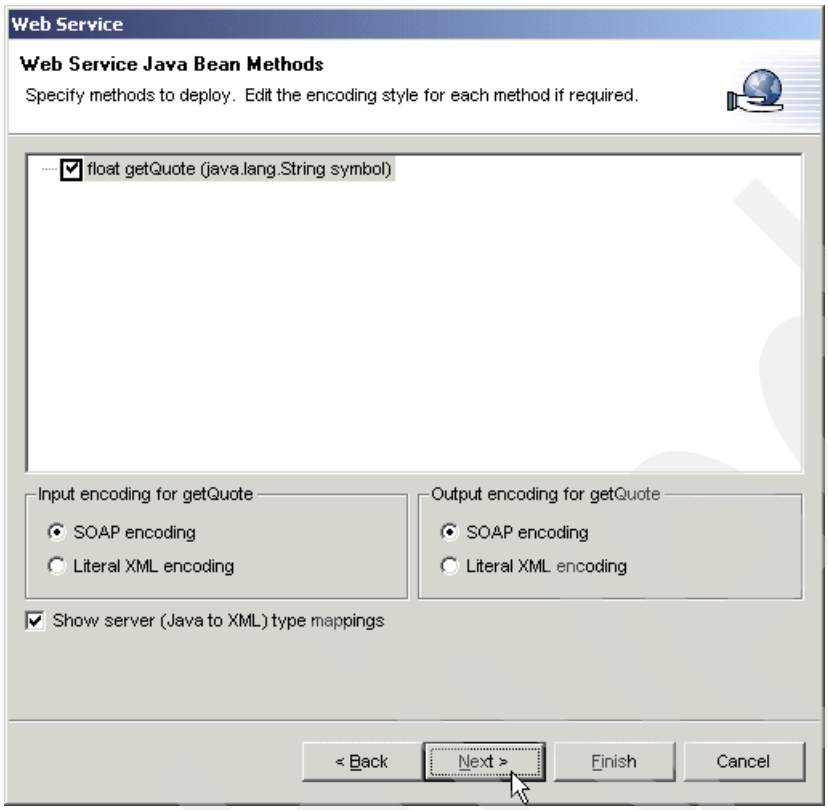


Figure 2-25 Specify encoding for Web Service Java Bean Methods in WDSc

7. Next, we specify the methods to deploy. As shown in Figure 2-25, methods of your Web Service Java Bean appear in selection. Make sure that **Input encoding for getQuote** method and **Output encoding for getQuote** method are both checked for **SOAP encoding**. This is because we are building a Web Service on the SOAP encoding envelope. Check **Show Server (Java to XML) type mappings** and click **Next**.

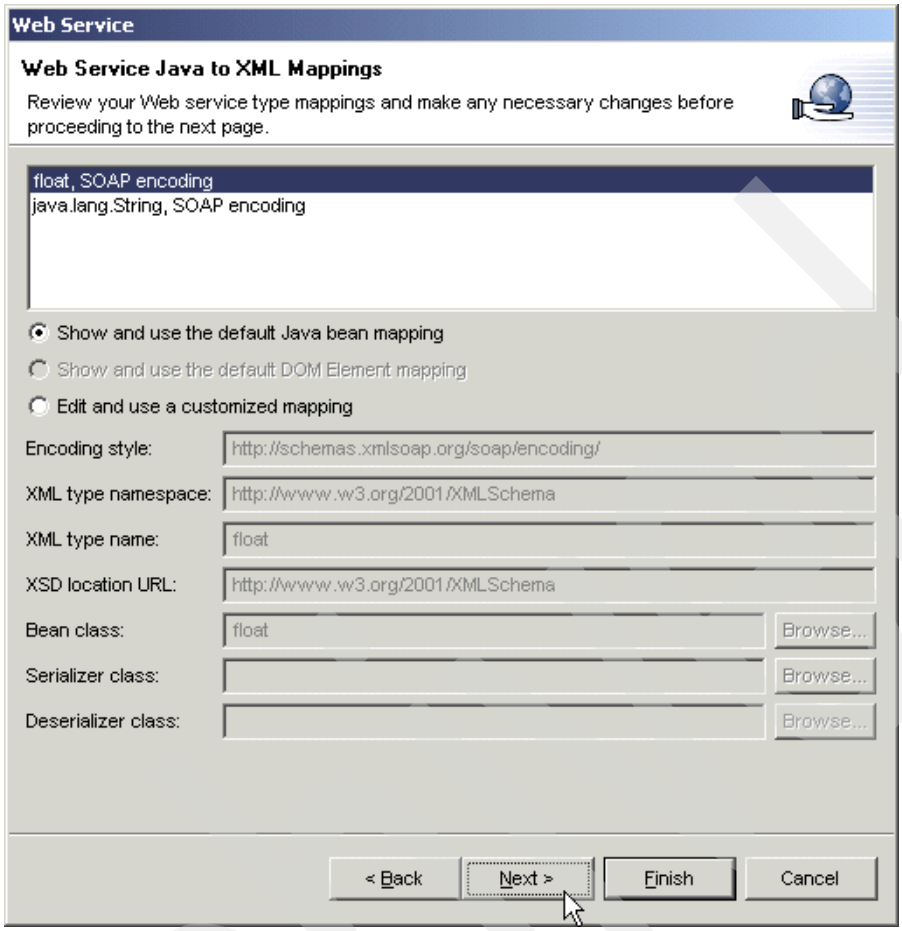


Figure 2-26 Verify Java to XML Mappings for Web Service on WDSc

8. Figure 2-26 shows the details of StockQuoteService Java bean method argument and return type with its Web Service encoding style. In this case **float, SOAP encoding** depicts the return type and **java.lang.String, SOAP encoding** depicts the call argument type for StockQuote Web Service. In this example we have the **Show and use the default Java bean mapping** radio button checked. But you can also have customized mappings by clicking on **Edit and use a customized mapping**. Customized mappings are specified when you want to use specialized classes in the call argument or return type method of the Web Service. For all other primitive data types like int, float and predefined Java runtime object class like java.lang.String you can use default mappings. Leave everything on this option as defaults and click **Next**.

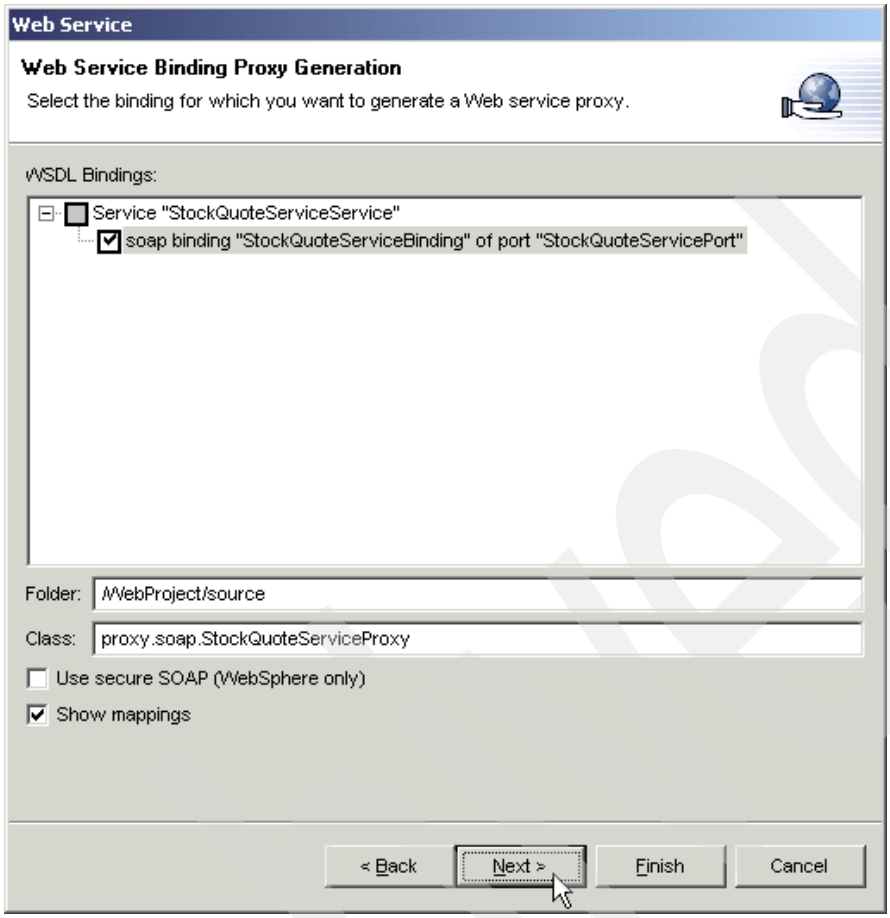


Figure 2-27 Verify Web Service Binding Proxy Generation option in WDSc

9. As shown in Figure 2-27, you will see binding type that will be generated for StockQuote Web Service. This binding will be stored in WSDL files and make the framework of the Web Service. To consume the Web Service, a proxy client is also generated in **WebProject/Source** folder and named as class **proxy.soap.StockQuoteServiceProxy**. Leave these options as defaults. Check the **Show mappings** box and click **Next** to continue.

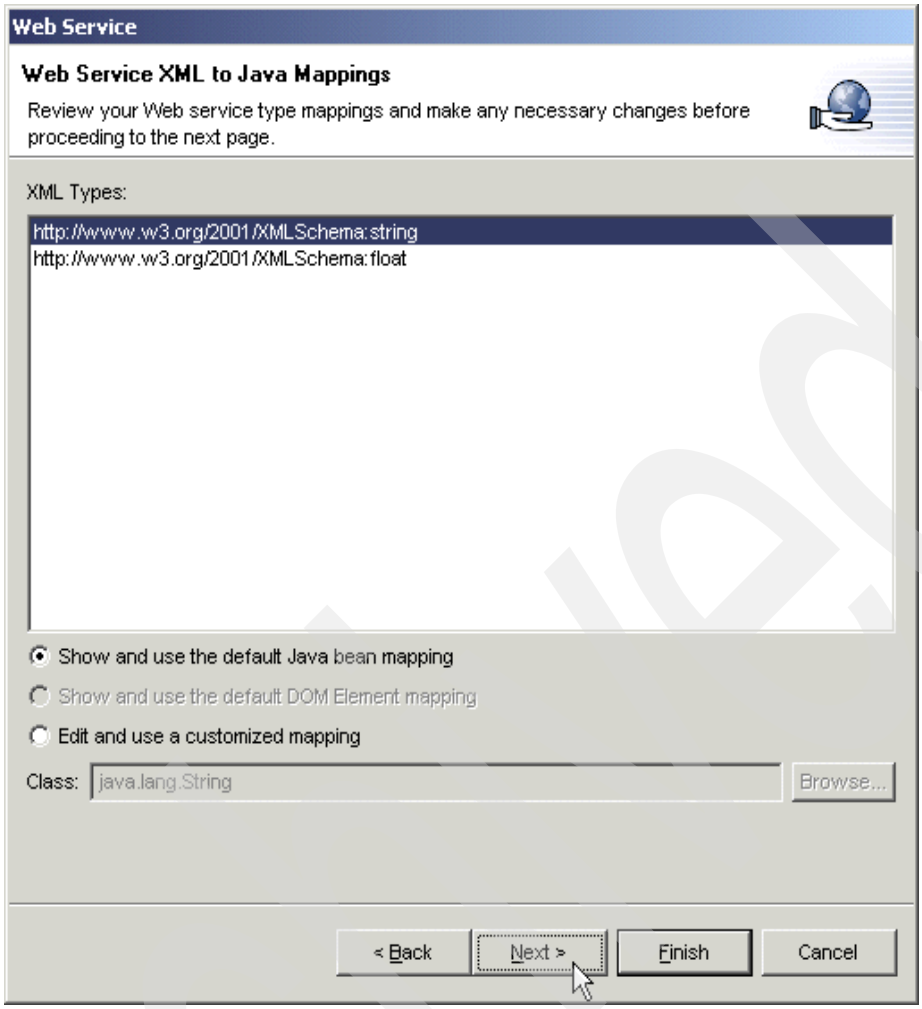


Figure 2-28 Verify the input and output parameter Schema mappings for Web Service in WDSc

10. As shown in Figure 2-28, WDSc then lets you verify whether the XMLSchema mapping and Java Class mapping for a particular input or output parameter is appropriate. Also here you get a chance to verify the mappings. In our case, we will leave default options and click **Next**.

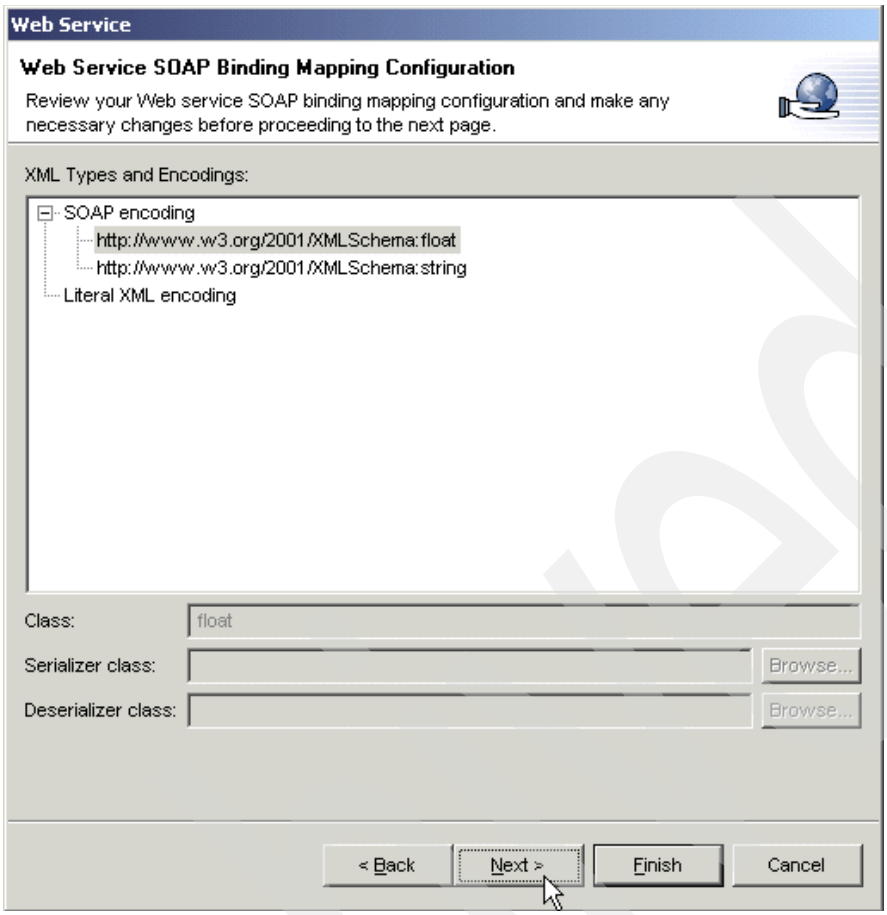


Figure 2-29 Web Service SOAP Binding configuration in WDSc

11. Figure 2-29 shows another figure to cross verify the configuration with SOAP encoding types for parameters and return types in the Web Service method. In some cases when the request parameter or return parameter are specialized classes we populate Serializer and Deserializer class fields. Leave everything as it is and click **Next**.

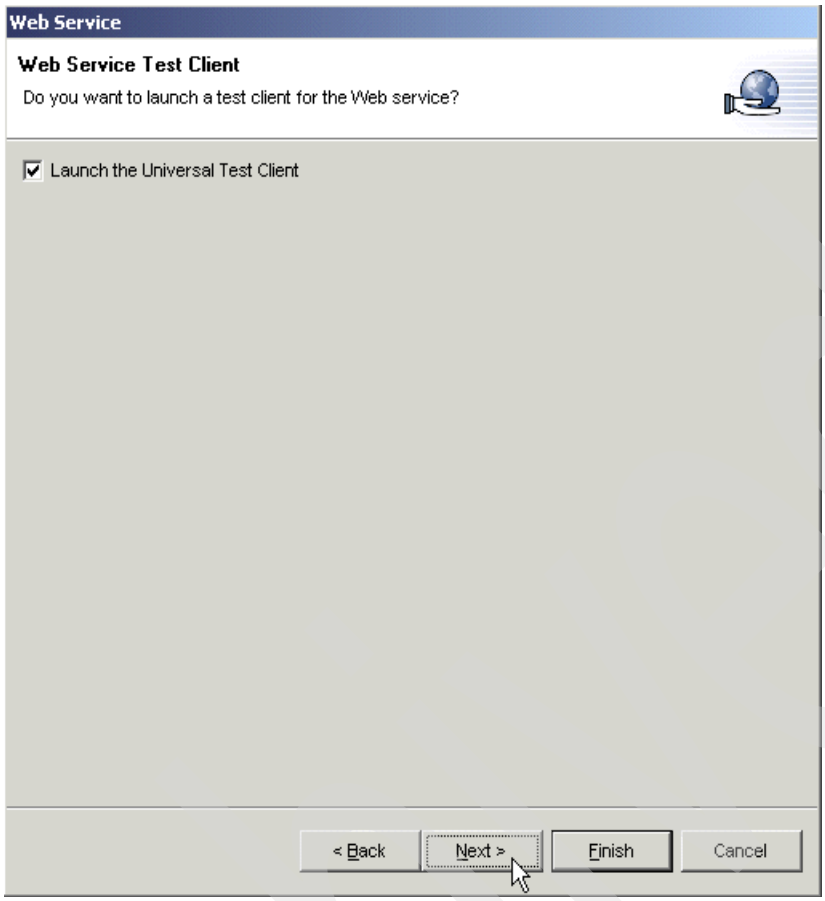


Figure 2-30 Test Client create selection window for Web Service on WDSc

12. In the Figure 2-30, you will have the option to create a Web Service test client. This test client is created as TestClient.jsp in WDSc and packaged under **sample** folder. Check on **Launch the Universal Test Client** and click **Next**.

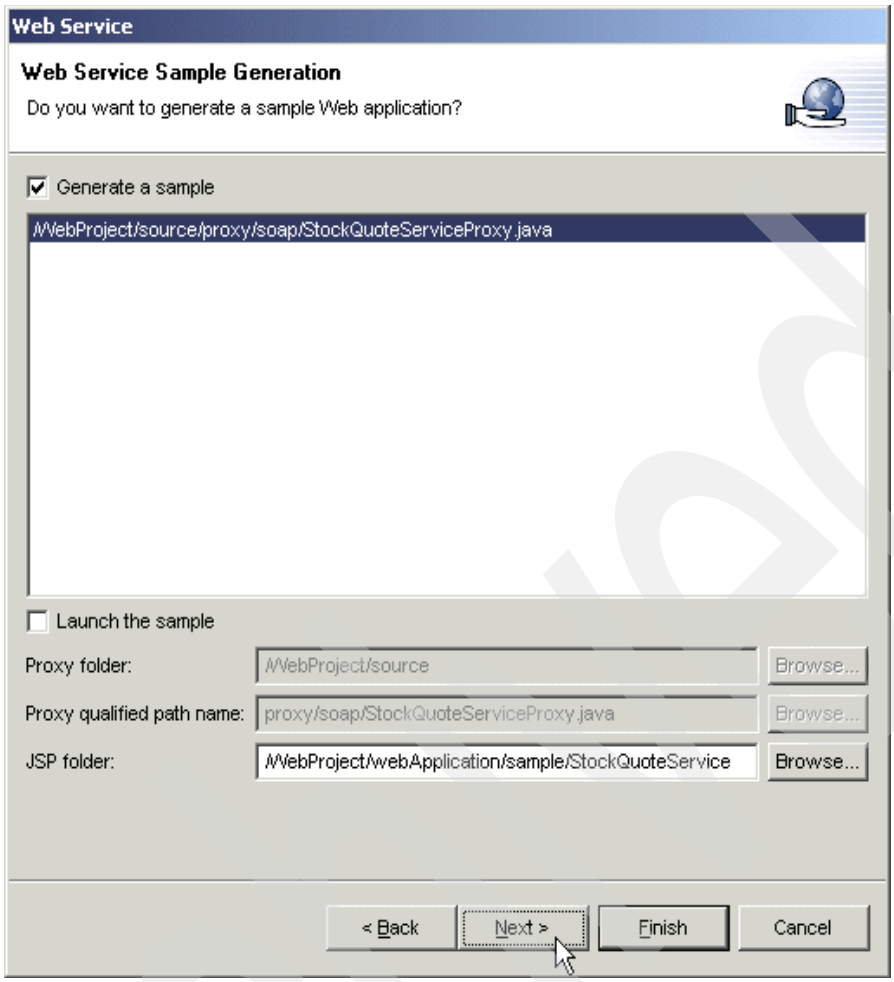


Figure 2-31 Web Service Sample Generation option for Web Service in WDSc

13. As shown in Figure 2-31, you then specify the sample generation option by checking on **Generate sample**, the sample TestClient.jsp will be generated in **JSP folder** as specified in the figure. Click **Next**.

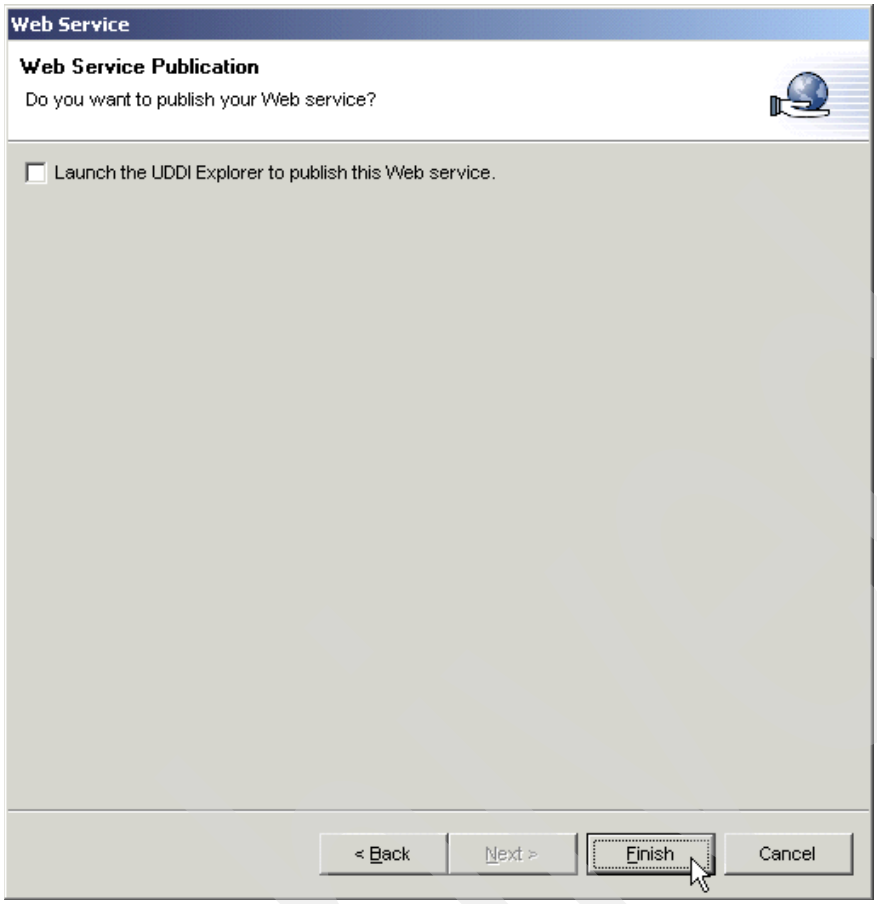


Figure 2-32 Web Service Publication option in WDSc

14. The next window, as shown in Figure 2-32, is for UDDI. In this chapter of our first Web Service application, we do not discuss Web Service publishing for UDDI. Leave the UDDI publication option unchecked and click **Finish**. This will complete the creation of a Web Service in WDSc.

> **Note:** For further discussion of UDDI, refer to Appendix B, “Using UDDI for both ends — publishing and requesting” on page 177.

2.2.4 Test the Web Service under a WDSc test environment

Before we actually export what we have created to WebSphere Application Server, in other words, before we leave WDSc, we can test the application under test environment of WDSc. This section describes its procedures.

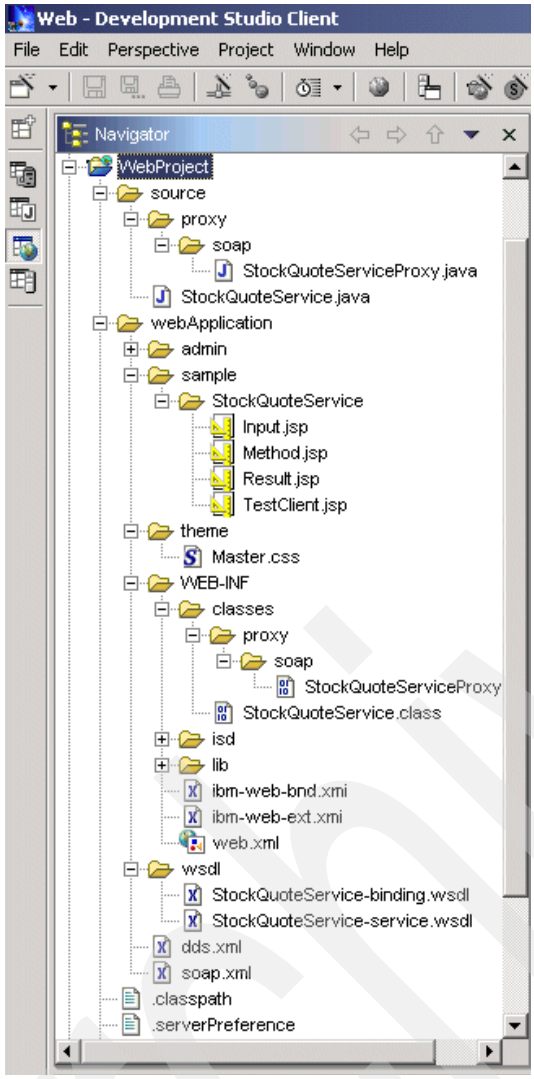


Figure 2-33 Directory structure after creation of StockQuote Web Service

1. Figure 2-33 shows you the directory structure after the Web Service StockQuote is created. This is the view from **Web perspective** of WDSc. You will notice that it has a Web Application module structure.
   - The **source** folder has all the Java files. One which is newly created in the Web Service creation is StockQuoteServiceProxy.java.
   - The next folder **webApplication** contains all the executable class files and other resource files.
   - The **admin** folder within **webApplication** contains **index.html**, which gives details on all the running Web Services.
   - The **sample** folder contains the TestClient.jsp file and other jsp resources.
   - **WEB-INF -> classes** folder contains all the classes required by the Web Service.
   - The **WSDL** folder contains the WSDL and XML files which define the framework of the Web Service in terms of Web serving methods, the parameters in these methods, their return types, XML encoding and so on.

Arrived

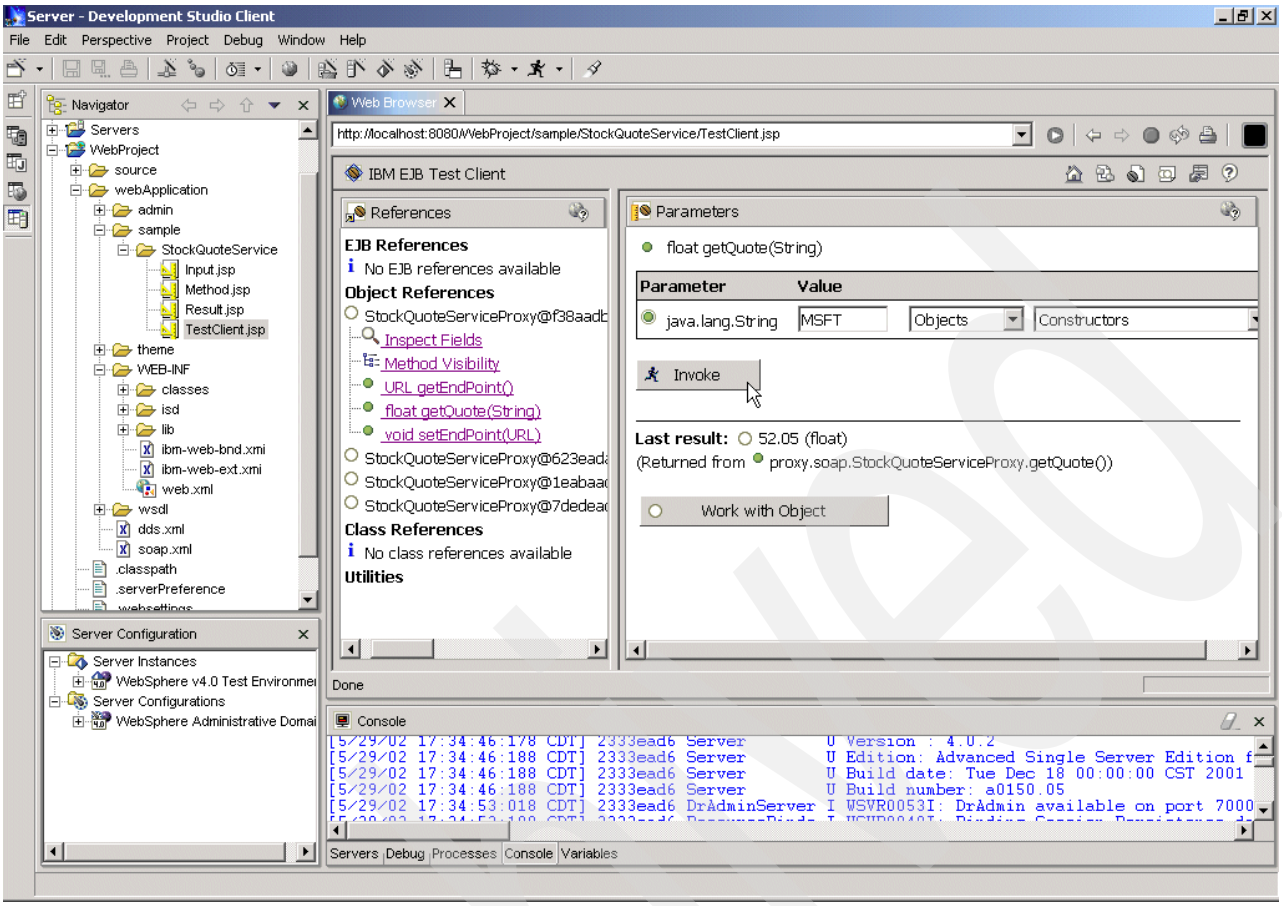


Figure 2-34 Launch the TestClient.jsp WebSphere Test Environment on WDSc

2. Figure 2-34 shows you how to make use of WebSphere Test Environment in WDSc to test your first Web Service application. After creation of a Web Service, WDSc will automatically start the WebSphere Test Environment for you and you will see a window similar to Figure 2-34. Since this Web Service by default is deployed to **localhost**, you can type in **http://localhost/WebProject/sample/StockQuoteService/TestClient.jsp** in the Web Browser pane of WDSc. On accessing the URL you will be provided with all the object reference to proxy client in the Web Service. In our case of StockQuote Web Service we want to test the getQuote method using StockQuoteServiceProxy. Click **float getQuote(String)** in the references pane and enter symbol **MSFT** in the value field of Parameters pane as java.lang.String input Parameter. Click **Invoke** test the Web Service. On a successful run of the Web Service you will see the result in the **Last Result:52.02** section of the Parameters pane. This is the return parameter of Stock's Current Value.

Congratulations! you have successfully created and tested your first Web Service on WDSc and WebSphere Test Environment under WDSc. If you had any trouble on the creation of the Web Service, refer to 2.4, "Tips and techniques" on page 60 for troubleshooting hints.

2.2.5 Setting up iSeries environment for deployment

There is a couple of admin items to prepare the iSeries for deployment, test, and actual running of a Web Service application. If you already have Web applications runtime environment under WebSphere Application Server, you may skip this part.

Steps to create a new instance on WAS on iSeries

Now you want to deploy your newly created Web Service in real world WebSphere Application Server environment. To deploy your Web Service on WebSphere Application Server, we will create a new instance called **iwebsvc** on WebSphere Application Server. You don't have to create a new instance, for example you can use default instance of WebSphere Application Server, but this way will give you more flexibility because you have your own runtime environment not to disturb (or being disturbed by) the others.

1. To create new instance on iSeries WebSphere Application Server, sign on to the iSeries server with QSECOFR authority.
2. On the command line type **strqsh** to start a QShell utility. You will get the window of QSHELL interpreter. This interpreter facilitates run of Java program on iSeries.
3. On QShell interpreter command line type **cd /QIBM/ProdData/WebAsAdv4/bin**. This will change the current directory path of QShell to bin directory where all the executable jar/class/bat files are located.
4. Make sure after run of each command that you get a "**$**" prompt. Type in command **crtnewinst -instance iwebsvc -bootstrap 8000 -lsd 12000 -exthttp 8800 -inthttp 5500**

   Note: WebSphere Application Server comes with its own internal HTTP stack. By specifying -inthttp 5500, we can run the test over the internal HTTP Server of WebSphere Application Server. In this scenario, we actually run the test over this port. This will simplify the test because we don't need to deal with the external HTTP Server instance. In your own runtime environment, ensure you can use external HTTP Server and the result should remain the same. Of course, this will be closer to real life production environment.

5. This will create an instance on WebSphere Application Server named **iwebsvc**. If there is any error on this command make sure that subsystem job QEJBADV4 in library QEJBADV4 is running.
6. Start the newly created instance by typing in command - **strwasinst -instance iwebsvc**.
7. Press **F3** and come out of QSHELL interpreter on AS/400 command line. Type in command **WRKACTJOB.**

Example 2-1 Verify new instance monitor job in WRKACTJOB on iSeries

```
Work with Active Jobs                   AS06
                                                          05/30/02  18:02:26
 CPU %:    7.8     Elapsed time:   08:25:18     Active jobs:   340

 Type options, press Enter.
   2=Change   3=Hold   4=End   5=Work with   6=Release   7=Display message
   8=Work with spooled files   13=Disconnect ...

 Opt  Subsystem/Job  User        Type  CPU %  Function        Status
      QEJBADV4       QSYS        SBS      .0                  DEQW
        IWEBSVMNTR   QEJB        BCH      .0  PGM-QEJBMNTR    EVTW
        QEJBADMIN    QEJB        BCI      .0  PGM-QEJBADMIN   JVAW
        QEJBADMIN    QEJB        BCI      .2  PGM-QEJBADMIN   JVAW
        QEJBADMIN    QEJB        BCI      .0  PGM-QEJBADMIN   JVAW
        QEJBMNTR     QEJB        ASJ      .0  PGM-QEJBMNTR    EVTW
        QEJBSVR      QEJB        BCI      .1  PGM-QEJBSVR     JVAW
        QEJBSVR      QEJB        BCI      .0  PGM-QEJBSVR     JVAW
        QEJBSVR      QEJB        BCI      .0  PGM-QEJBSVR     JVAW
                                                                More...
 Parameters or command
```

```
===>
F3=Exit    F5=Refresh       F7=Find      F10=Restart statistics
F11=Display elapsed data    F12=Cancel   F23=More options   F24=More keys
```

You can see in this Example 2-1, the monitor job **IWEBSVMNTR** is monitor job for the **iwebsvc** instance.

Change the Web Service configuration files for deployment on WAS

Now you need to customize your Web Service configuration files to fit in your WebSphere Application Server instance.

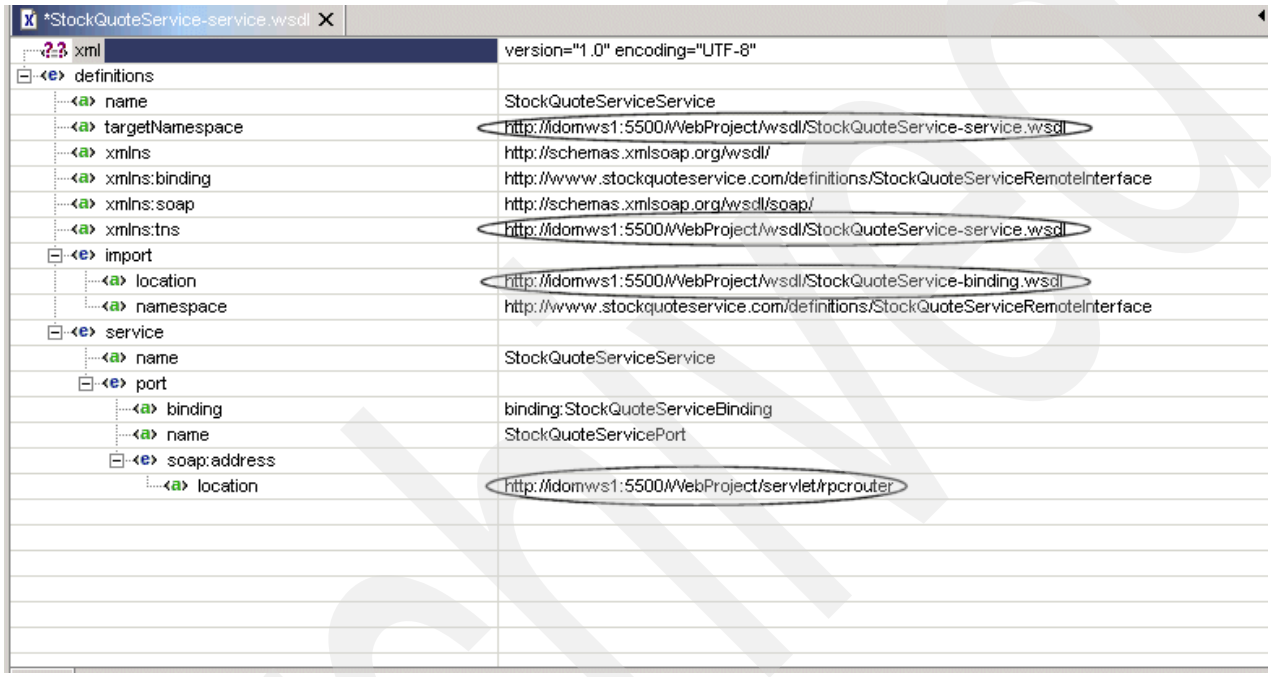


Figure 2-35 Make Changes in WSDL files on WebProject to point to WebSphere Application Server

1. Figure 2-35 illustrates how to customize the newly created StockQuote Web Service for deployment on WebSphere Application Server.
   a. Click the **WSDL** folder in the **Web perspective** view of WebProject.
   b. Double-click the **StockQuoteService-service.wsdl** file from the **WSDL** folder.
   c. Expand the **definitions**, **import** and **service** portion of WSDL file.
   d. Change the host name from **localhost** to **idomws1:5500** in:
      - **targetNameSpace**
      - **xmlns:tns**
      - **import -> location**
      - **service -> port -> soap:address -> location**
      - Here **idomws1** is the Domino HTTP Server on iSeries and **5500** is the port number on which our instance of application server is running on WebSphere Application Server.

Archived

Note: We use the HTTP instance under Domino server, because we already set up the runtime environment like this for our next scenario, “Transforming a Domino application to a Web Service application” on page 61. In your case, you are free to use another HTTP Server, such as IBM HTTP Server powered by Apache.

e. After completing changes in WSDL files, change or save it by clicking **File -> Save**.

```
public class StockQuoteServiceProxy
{
  private Call call = new Call();
  private URL url = null;
  private String stringURL = "http://idomws1:9080/WebProject/servlet/rpcrouter";
  private SOAPMappingRegistry smr = call.getSOAPMappingRegistry();

  public StockQuoteServiceProxy()
  {
  }
```

Figure 2-36 Change the StockQuoteServiceProxy to point to your WAS instance in WDSc

2. Then, change the StockQuoteServiceProxy.java in proxy/soap folder under source, as shown in Figure 2-36. In the StockQuoteServiceProxy code make sure the http://**localhost**/WebProject/servlet/rpcrouter is changed to http://**idomws1:5500**/WebProject/servlet/rpcrouter.

2.2.6 Deployment of Web Service on WebSphere Application Server

We have setup the environment on iSeries. Now it’s time to actually export the Web Service to WebSphere Application Server for the deployment.

Export the Web Service for deployment on WAS

To deploy StockQuote Web Service on the instance **iwebsvc** we first need to export it to WebSphere Application Server. This section describes the export procedures of the Web Service to WebSphere Application Server.

Archived

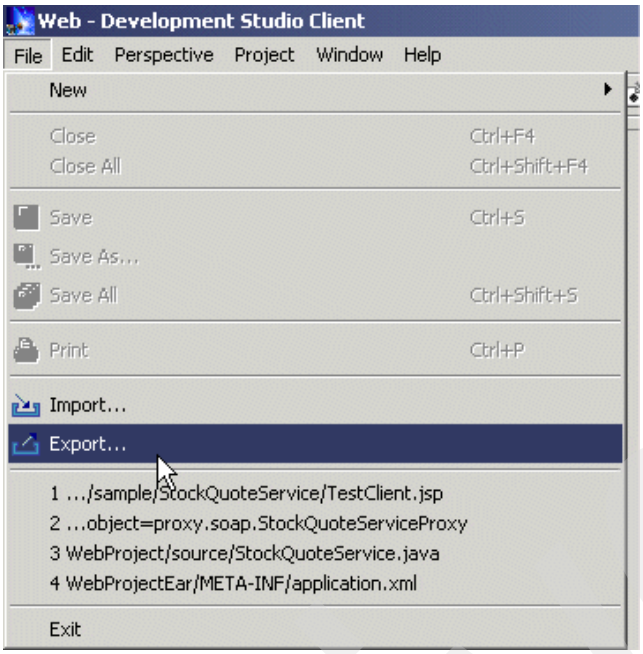


Figure 2-37 Export the StockQuote Service Web project in WDSc

1. Select **WebProject** and click **File** > **Export**, as shown in Figure 2-37.

Archived

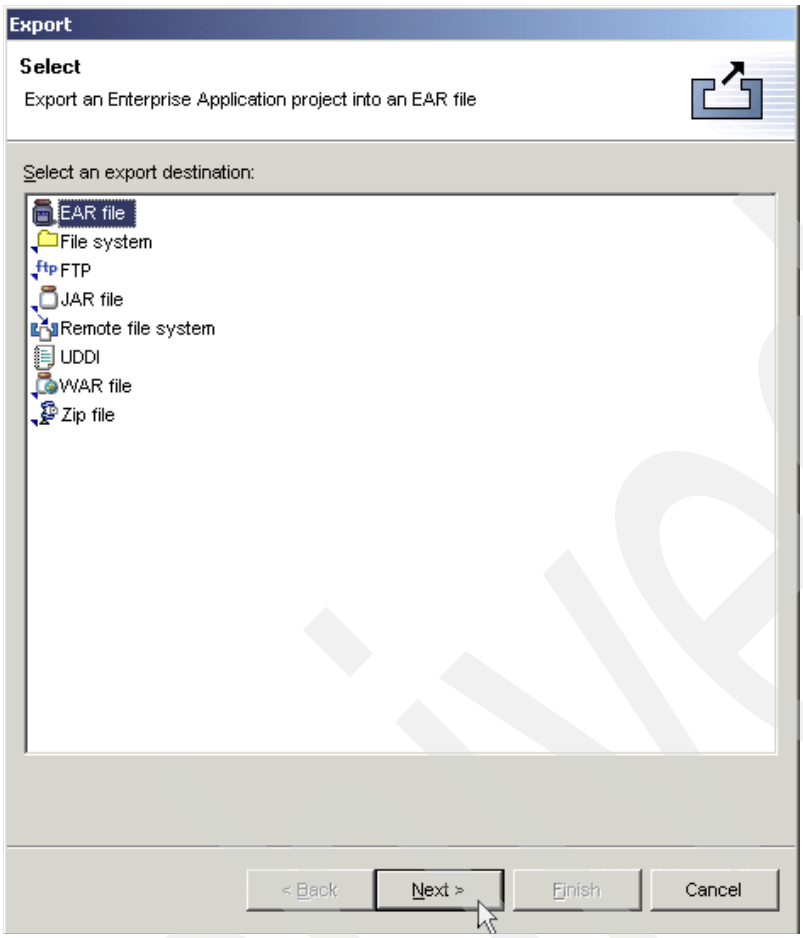


Figure 2-38 Export as EAR file WebProject in WDSc

2. As Figure 2-38 shows, click the **EAR file** as export time for the WebProject Enterprise Application.

Archived

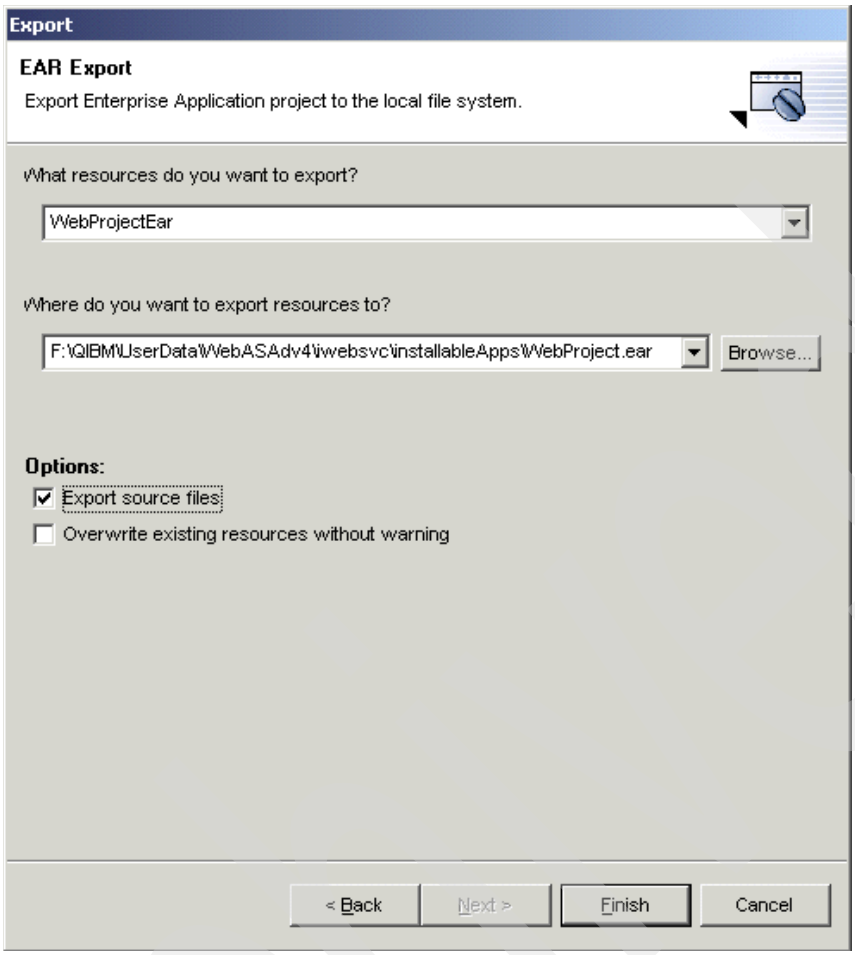


Figure 2-39 Select library path to export the EAR file in WDSc

3. As shown in Figure 2-39, make sure that the option **what resources do you want to export** has **WebProjectEar** selected. On the **Where do you want to export resources to?** option, browse and select the IFS directory to iSeries WebSphere Application Server install, for example: **x:/QIBM/UserData/WebAsAdv4/iwebsvc/installableApps/WebProject.ear**.

   Also check the **Export source files** option. Click **Finish** to export.

Deploy the Web Service on WebSphere Application Server

Now to install this **WebProject.ear** file on WebSphere Application Server, we need to start the admin console for WebSphere Application Server.

1. Go to a DOS command line prompt of your Windows machine and change directory to **WebSphere/AppServer/bin**. On this directory prompt, type in the command **adminclient <node name> <port number>**. This will start the WebSphere Application Server Java based admin console.

   Note: For adminclient port number, use **8000**. This is the port number we assigned for bootstrap when we created our own WebSphere Application Server instance of **iwebsvc**.

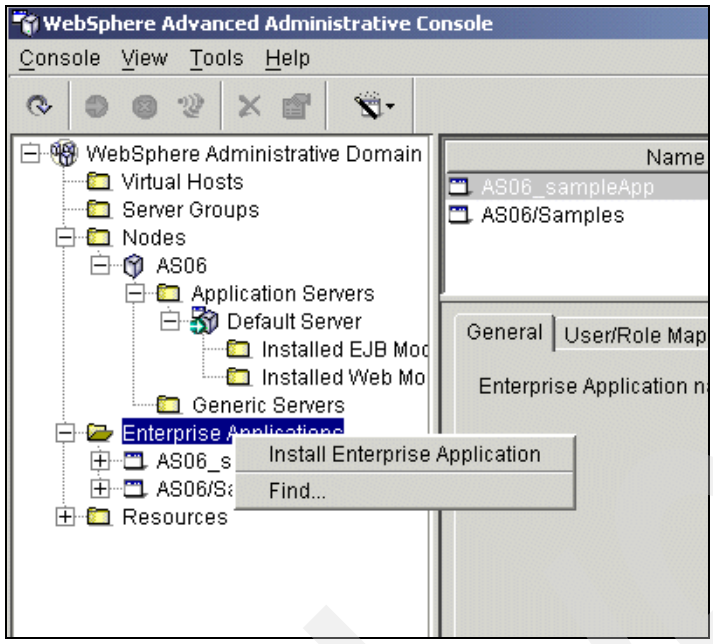


Figure 2-40 Install the Enterprise Application on iwebsvc instance on WebSphere Application Server

2. The initial configuration of the new instance **iwebsvc** has been created with the Default Server Web application server. Right-click **Enterprise Applications** and click **Install Enterprise Application** as shown in Figure 2-40.

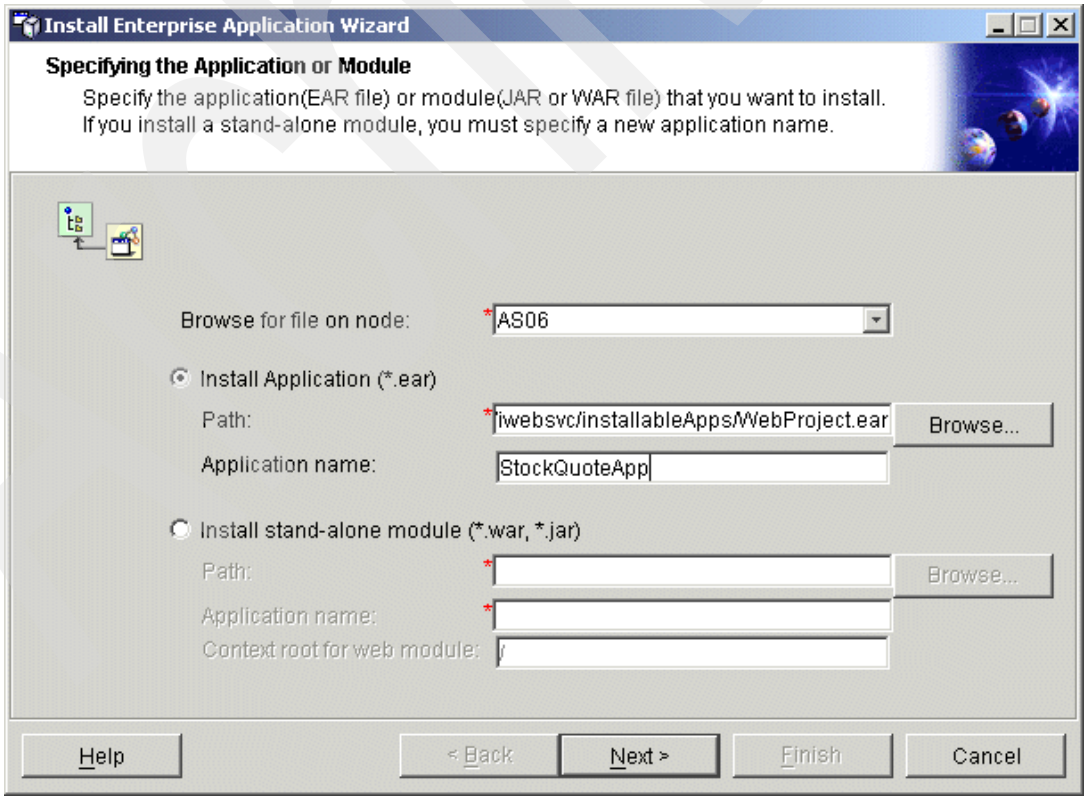


Figure 2-41 Specify path of EAR file on install of Enterprise Application on WAS

3. As shown in Figure 2-41, browse to this directory to select the EAR file for deployment:

   `QIBM/UserData/WebAsAdv4/iwebsvc/installableApps/WebProject.ear`

   Enter the Application name as **StockQuoteApp**.

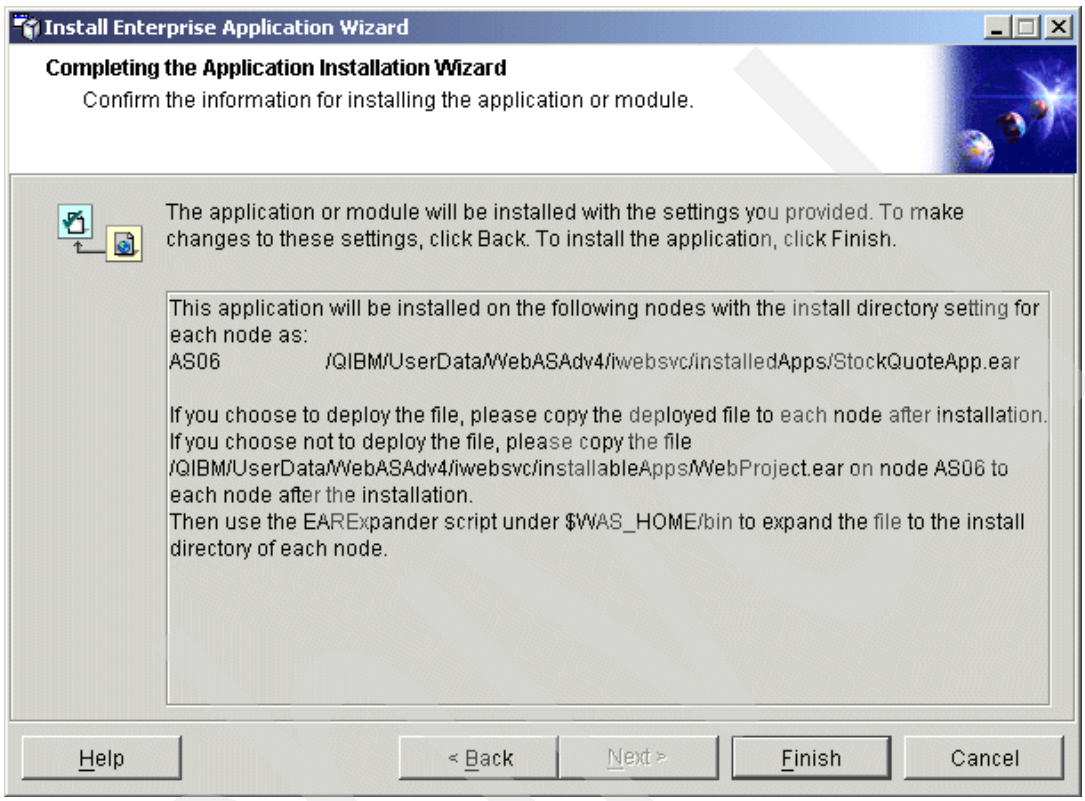


Figure 2-42 Install Enterprise Application Finish window on WebSphere Application Server

4. After taking defaults on other windows, make sure that install directory path is correct on the final window as shown in Figure 2-42 and click **Finish**.

Archive

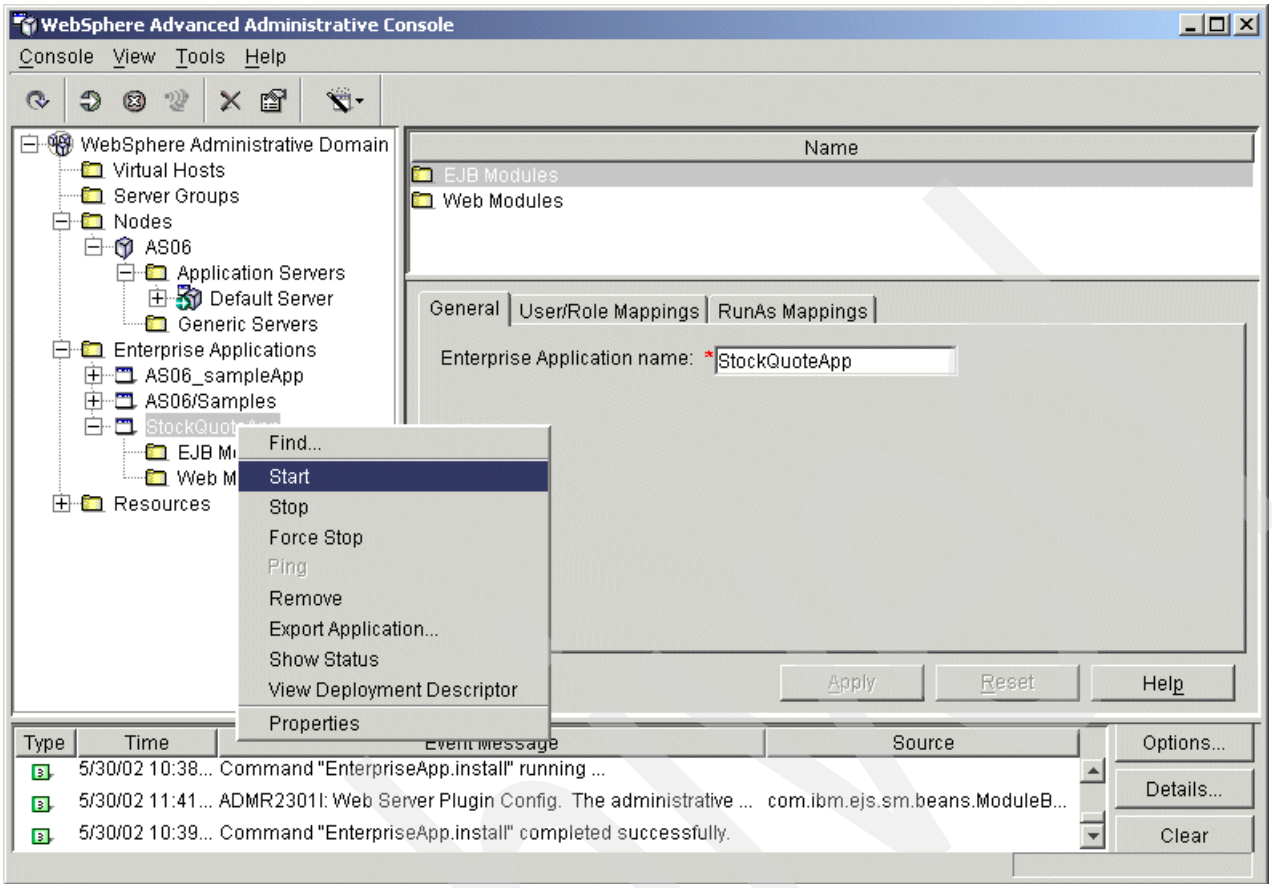


Figure 2-43 Start the StockQuoteApp Enterprise Application on WebSphere Application Server

5. As shown in Figure 2-43, start the StockQuoteApp application by right-clicking on **StockQuoteApp**. Press **Start**. This will complete the deployment and getting ready for the Web Service on WebSphere Application Server.

2.2.7 Test the WebSphere Application Server deployment of the Web Service

We finally came to the end of the whole cycle: Testing our Web Service application from a browser. In this test scenario, the Web browser simulates the consumer of our Web Service application service: in other words, the browser is the Web Service requester.

Figure 2-44 Verify the Web Service is running using XML-SOAP Admin client on Web Browser

1. Point your browser to **http://idomws1:5500/WebProject/admin/index.html**. As shown in Figure 2-44, you will see active services by clicking on **List all services**. In our example the Web Service is identified by URI **http://tempuri.org/StockQuoteService**.

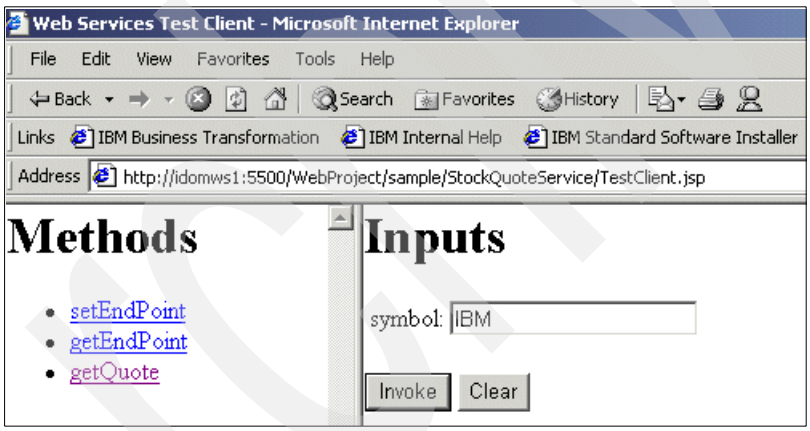


Figure 2-45 Test the Web Service using TestClient.jsp on Web browser

2. Point your browser here to run the sample test client:
   http://idomws1:5500/WebProject/sample/StockQuoteService/TestClient.jsp

   Click **getQuote** method and enter **IBM** as the stock quote symbol.

Figure 2-46 Result of StockQuote Web Service TestClient.jsp on Web browser

3. If you get a response like Figure 2-46, that means your Web Service application is up and running.

Congratulations! you have successfully developed and tested your first Web Service application on WebSphere Application Server.

2.3 Analysis of StockQuoteApp Web Service application

This section provides a detailed discussion on the components created through this scenario. It also discusses actual lines of code of significance from the Web Service application's perspective. This section is primarily for developers. We will show the actual coding and contents of the key components of our scenario here. Figure 2-47 illustrates the components we cover in this section.

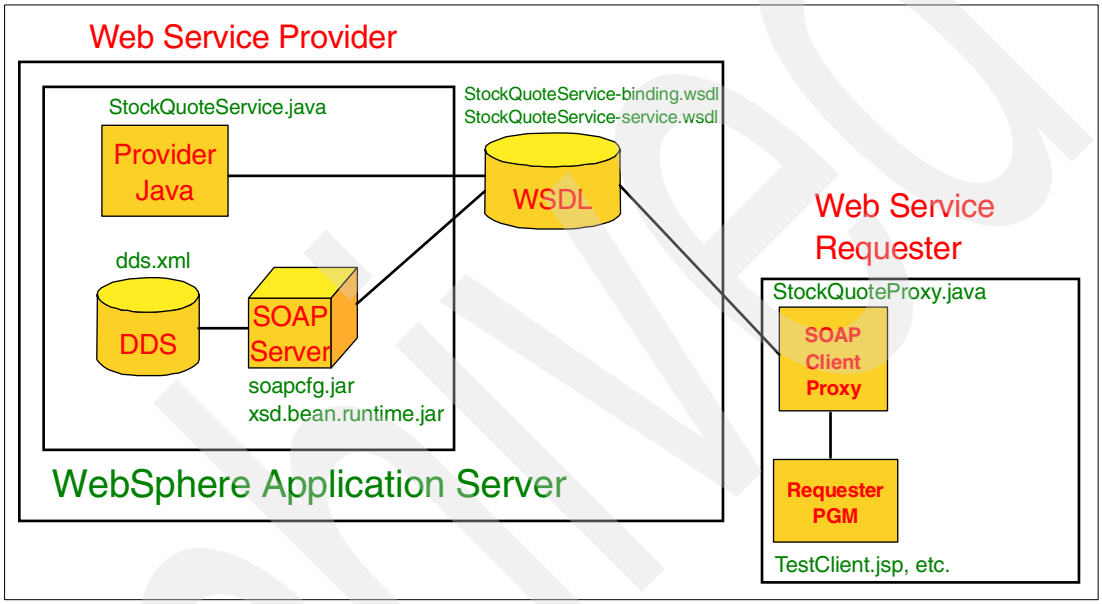


Figure 2-47 Actual components of StockQuoteApp Web Service application

2.3.1 Provider Java program: StockQuoteService.java

This program StockQuoteService performs the core functionality of StockQuote Web Service. As seen here, stock quotes are requested through this URL:

```
http://www.xmltoday.com/examples/stockquote/getxmlquote.vep?s=symbol
```

Symbol here is a variable which contains various NASDAQ listed symbols of the companies. Since the URL for stock information "www.xmltoday.com" returns stock quotes in the form of an XML DOM document, the price details are extracted using DOM (document object model) parser. These price details are returned to a calling program in Java float primitive data type. Refer to Figure 2-48.

```
public class StockQuoteService {

  public float getQuote (String symbol) throws Exception {
    // get a real (delayed by 20min) stockquote from
    // http://www.xmltoday.com/examples/stockquote/. The IP addr
    // below came from the host that the above form posts to ..
    URL url = new URL
("http://www.xmltoday.com/examples/stockquote/getxmlquote.vep?s="+symbol);
    InputStream is = url.openStream ();
    DOMParser parser = new DOMParser();
    parser.parse(new InputSource(is));
    Document d = parser.getDocument();
    Element e = d.getDocumentElement ();
    NodeList nl = e.getElementsByTagName ("price");
    e = (Element) nl.item (0);
    String quoteStr = e.getAttribute ("value");
    try {
      return Float.valueOf (quoteStr).floatValue ();
    } catch (NumberFormatException e1) {
      // mebbe its an int?
      try {
        return Integer.valueOf (quoteStr).intValue () * 1.0F;
      } catch (NumberFormatException e2) {
        return -1.0F;
      }
    }
  }
}
```

Figure 2-48 StockQuoteService Provider Java Program for the Web Service

2.3.2 Deployment Descriptor XML: DDS.XML

Apache SOAP uses XML documents called *deployment descriptors* to provide information to SOAP runtime about services made available. A deployment descriptor which exposes a service is implemented via core functionality of Java Bean/Class as shown here in Figure 2-49. The service URI is the Unique Resource Identifier (URI) that you want to give to StockQuote Web Service. The provider type is a Java Class called StockQuoteService as shown in Figure 2-49. The exposed method by this provider class is getQuote. The option attribute *checkMustUnderstands* may be set to “True” or “False” depending on whether you want the server to throw a *fault*. A fault is SOAP exception. The *static* attribute refers to the method getQuote being static or not in Java class. The *scope* attribute denotes implementing a Java class instantiation scope. An application scope indicates that the object will last until the servlet which is servicing request is terminated.

Archived

```
- <root>
- <isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
id="http://tempuri.org/StockQuoteService" checkMustUnderstands="false">
- <isd:provider type="java" scope="Application" methods="getQuote">
  <isd:java class="StockQuoteService" static="false" />
  </isd:provider>
  </isd:service>
  </root>
```

Figure 2-49 DDS.XML for StockQuote Web Service

2.3.3 SOAP Server components: soapcfg.jar, xsd.bean.runtime.jar

The Apache SOAP server is comprised of XML based configuration files and transport listener Java classes. The server will look into the current working directory for a file called soap.xml. Since this StockQuote Web Service application is deployed on WebSphere, the WASRPCRouterServlet.class and WASMessageRouterServlet.class are components of soapcfg.jar file. The RPCRouterServlet acts as a transport listener for all the requests for the Web Service in SOAP server. The other jar file xsd.bean.runtime.jar provides runtime classes for the SOAP server for StockQuote Web Service.

2.3.4 WSDL documents

Two WSDL documents are created in this scenario: StockQuoteService-service.wsdl and StockQuoteService-binding.wsdl. This section describes these.

WSDL is an XML format that describes the SOAP Web Service as an information exchange channel. Our StockQuote Web Service information exchange channel uses procedure-oriented information called Remote Procedure Call (RPC). Here are two WSDL documents generated by WDSc for StockQuote Web Service. Figure 2-50 shows the WSDL service document which defines a Web Service *port* named StockQuoteServicePort and *soap address* pointing to rpcrouter servlet.

The other WSDL document StockQuoteService-binding.wsdl as depicted in Figure 2-51 on page 55, is an XML format that describes message format and protocol details for a message exchange by StockQuoteServicePort. The request *message* attributes define *getQuoteRequest* and *getQuoteResponse* as stock quote message exchange types. A string data type is sent as a request for getQuote in the form of stock symbol, and float data type is returned as response for getQuote in the form of stock price.

Archived

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="StockQuoteServiceService"
    targetNamespace="http://idomws1:9080/WebProject/wsdl/StockQuoteService-service.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"

xmlns:binding="http://www.stockquoteservice.com/definitions/StockQuoteServiceRemoteInter
face"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://idomws1:5500/WebProject/wsdl/StockQuoteService-service.wsdl">
    <import
        location="http://idomws1:5500/WebProject/wsdl/StockQuoteService-binding.wsdl"
namespace="http://www.stockquoteservice.com/definitions/StockQuoteServiceRemoteInterface
"/>
    <service name="StockQuoteServiceService">
        <port binding="binding:StockQuoteServiceBinding" name="StockQuoteServicePort">
            <soap:address location="http://idomws1:5500/WebProject/servlet/rpcrouter"/>
        </port>
    </service>
</definitions>
```

Figure 2-50 StockQuoteService-service.wsdl

Archived

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="StockQuoteService-binding"

targetNamespace="http://www.stockquoteservice.com/definitions/StockQuoteServiceRemoteInt
erface"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:tns="http://www.stockquoteservice.com/definitions/StockQuoteServiceRemoteInterface
" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <import location="http://www.w3.org/2001/XMLSchema"
namespace="http://www.w3.org/2001/XMLSchema"/>
    <message name="getQuoteRequest">
        <part name="symbol" type="xsd:string"/>
    </message>
    <message name="getQuoteResponse">
        <part name="result" type="xsd:float"/>
    </message>
    <portType name="StockQuoteService">
        <operation name="getQuote" parameterOrder="symbol">
            <input message="tns:getQuoteRequest" name="getQuoteRequest"/>
            <output message="tns:getQuoteResponse" name="getQuoteResponse"/>
        </operation>
    </portType>
    <binding name="StockQuoteServiceBinding" type="tns:StockQuoteService">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="getQuote">
            <soap:operation soapAction="" style="rpc"/>
            <input name="getQuoteRequest">
                <soap:body
                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                    namespace="http://tempuri.org/StockQuoteService" use="encoded"/>
            </input>
            <output name="getQuoteResponse">
                <soap:body
                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                    namespace="http://tempuri.org/StockQuoteService" use="encoded"/>
            </output>
        </operation>
    </binding>
</definitions>
```

Figure 2-51 StockQuoteService-binding.wsdl

2.3.5 StockQuoteServiceProxy.java: Web requester Java client

WDSc generates a StockQuoteServiceProxy Java class as client for StockQuote Web Service. The Java code shown in Figure 2-52 and Figure 2-53, tells how the proxy client serves up as a Web Service Requester. To send a request, the proxy uses a SOAP call and sets the endpoint to this URL:

```
http://idomws1:5500/WebProject/servlet/rpcrouter
```

The parameters for getQuote method are set with the stock quote symbol as the input parameter. After setting the required parameters and encoding styles, proxy invokes the call to SOAP server. The SOAP server processes the request and sends back the SOAP

response. This response is decoded by proxy for any errors. If there are no errors, the stock quote price is returned to the calling requester program as float data type.

```
package proxy.soap;

import java.net.*;
import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.messaging.*;

public class StockQuoteServiceProxy
{
  private Call call = new Call();
  private URL url = null;
  private String stringURL = "http://idomws1:5500/WebProject/servlet/rpcrouter";
  private SOAPMappingRegistry smr = call.getSOAPMappingRegistry();

  public StockQuoteServiceProxy()
  {
  }

  public synchronized void setEndPoint(URL url)
  {
    this.url = url;
  }

  public synchronized URL getEndPoint() throws MalformedURLException
  {
    return getURL();
  }

  private URL getURL() throws MalformedURLException
  {
    if (url == null && stringURL != null && stringURL.length() > 0)
    {
      url = new URL(stringURL);
    }
    return url;
  }
```

Figure 2-52 StockQuoteServiceProxy.java (1 of 2)

Archived

```
public synchronized float getQuote(java.lang.String symbol) throws Exception
  {
    String targetObjectURI = "http://tempuri.org/StockQuoteService";
    String SOAPActionURI = "";

    if(getURL() == null)
    {
      throw new SOAPException(Constants.FAULT_CODE_CLIENT,
      "A URL must be specified via StockQuoteServiceProxy.setEndPoint(URL).");
    }

    call.setMethodName("getQuote");
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setTargetObjectURI(targetObjectURI);
    Vector params = new Vector();
    Parameter symbolParam = new Parameter("symbol", java.lang.String.class, symbol,
Constants.NS_URI_SOAP_ENC);
    params.addElement(symbolParam);
    call.setParams(params);
    Response resp = call.invoke(getURL(), SOAPActionURI);

    //Check the response.
    if (resp.generatedFault())
    {
      Fault fault = resp.getFault();
      call.setFullTargetObjectURI(targetObjectURI);
      throw new SOAPException(fault.getFaultCode(), fault.getFaultString());
    }
    else
    {
      Parameter refValue = resp.getReturnValue();
      return ((java.lang.Float)refValue.getValue()).floatValue();
    }
  }

}
```

Figure 2-53 StockQuoteServiceProxy.java (2 of 2)

2.3.6 Requester Programs: TestClient.jsp, Result.jsp

Here we describe the client requester programs for the Web Service that are accessed through a Web browser. TestClient.jsp (refer to Figure 2-54 on page 58), is a JSP (Java Server Pages) program created by WDSc to test the Web Service. This jsp requester calls other jsps like Method.jsp, Result.jsp, and Inputs.jsp to list the available requester methods in the Web Service and furnishes the results when input is submitted.

Archived

```
<HTML>
<HEAD>
<TITLE>Web Services Test Client</TITLE>
</HEAD>
<FRAMESET  COLS="220,*">
<FRAME SRC="Method.jsp" NAME="methods" MARGINWIDTH="1" MARGINHEIGHT="1" SCROLLING="yes"
FRAMEBORDER="1">
<FRAMESET  ROWS="80%,20%">
<FRAME SRC="Input.jsp" NAME="inputs"  MARGINWIDTH="1" MARGINHEIGHT="1" SCROLLING="yes"
FRAMEBORDER="1">
<FRAME SRC="Result.jsp" NAME="result"  MARGINWIDTH="1" MARGINHEIGHT="1" SCROLLING="yes"
FRAMEBORDER="1">
</FRAMESET>
<NOFRAMES>
<BODY>
The Web Services Test Client requires a browser that supports frames.
</BODY>
</NOFRAMES>
</FRAMESET>
</HTML>
```

Figure 2-54 TestClient.jsp for StockQuote Web Service

Archived

```
<HEAD>
<TITLE>Result</TITLE>
</HEAD>
<BODY>
<H1>Result</H1>

<jsp:useBean id="StockQuoteServiceid" scope="session"
class="proxy.soap.StockQuoteServiceProxy" />

....
....
<%
String method = request.getParameter("method");
if (method == null) method = "";

boolean gotMethod = false;

try {
if (method.equals("setEndPoint")) {

        gotMethod = true;
        String url0id=  markup(request.getParameter("url5"));
        java.net.URL url0idTemp  = new java.net.URL(url0id);
        StockQuoteServiceid.setEndPoint(url0idTemp);
}else if (method.equals("getEndPoint")) {

        gotMethod = true;
        java.net.URL mtemp = StockQuoteServiceid.getEndPoint();
        String tempResultresult8 = markup(mtemp.toString());
        %>
        <%= tempResultresult8 %>
        <%
}else if (method.equals("getQuote")) {

        gotMethod = true;
        String symbol1id=  markup(request.getParameter("symbol13"));
        java.lang.String symbol1idTemp  = symbol1id;
        float mtemp = StockQuoteServiceid.getQuote(symbol1idTemp);
        String tempResultresult11 = markup(String.valueOf(mtemp));
        %>
        <%= tempResultresult11 %>
        <%
}} catch (Exception e) {
%>
exception: <%= e %>
<%
return;
}
if(!gotMethod){
%>
result: N/A
<%
}
%>
</BODY>
</HTML>
```

Figure 2-55 getQuote method in Result.jsp for StockQuote Web Service

In Figure 2-55, we show the a portion of Result.jsp file generated by WDSc as sample code. The code here instantiates the proxy.soap.StockQuoteServiceProxy bean and stores it in the form of StockQuoteServiceid. The required stock quote symbol is passed to getQuote method of StockQuoteServiceid and the stock quote price is obtained. This stock quote price is shown on Result.jsp page as result.

2.4 Tips and techniques

This section lists some tips and techniques related to the scenario:

- For troubleshooting and checking purposes, we recommend that the Web Service first be tested in the WebSphere Test Environment of WDSc.
- After deployment of the enterprise application on WebSphere Application Server, make sure that installation directory path is generated correctly by looking at: /QIBM/UserData./WebAsADv4 directory of WebSphere Application Server
- On WebSphere Application Server make sure that the appropriate node has a proper Web server plug-in generated against it.
- If you get a page not found error on servlet, jsp, or html pages, make sure the context root of application is properly set. Also check the URL entered in browser for such an error.
- Always make sure that the Web Service is running by pointing your browser to the SOAP admin page:

```
http://<yourserver>:<yourport>/<yourcontextroot>/admin/index.html
```

Archived

3

Transforming a Domino application to a Web Service application

This chapter discusses transforming the existing Domino applications into Web Service applications. Although there can be other ways of doing this, we stay with the framework we use throughout this book, which is serving the requests from the clients from WebSphere Application Server. For this, as we did in the previous chapter, we use WDSc for development and deployment tools. For an actual applications example, we use one application coming from Lotus Web Service Enablement Kit (LWSEK), which is available from IBM's alphaWorks Web site. For a detailed explanation of LWSEK, what it is and how it can help you, refer to 3.4, "Analysis of LWSEK" on page 93.

We walk you through the process of transforming the existing Domino application. This chapter consists of the following sections:

- Existing Domino application — before transformation
- How to port the existing Domino application into a Web Service Java application still accessing Domino database
- How to deploy the ported Java application as a Web Service application served under WebSphere Application Server using WDSc
- Analysis of LWSEK — what it is and how it can help you

© Copyright IBM Corp. 2002. All rights reserved.

3.1 Existing Domino applications — before transformation

We start our journey by investigating the existing Domino application currently serving Notes clients only. We have a LotusScript application named ExpertiseLocator. The Notes client user opens a form similar to Figure 3-1.

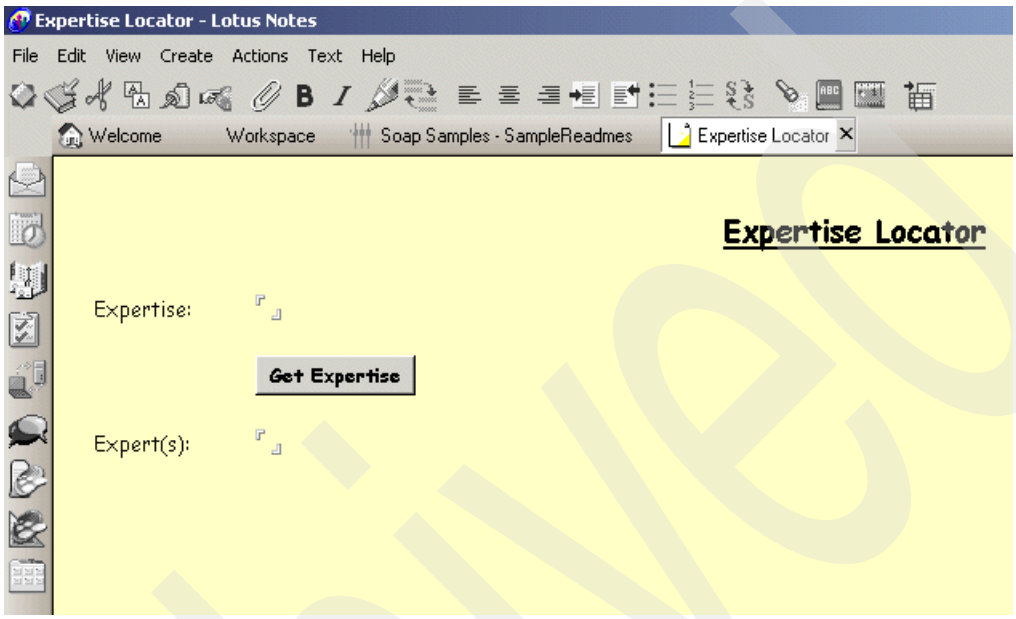


Figure 3-1 ExpertiseLocator LotusScript application input form example

For user keys in a specific product name, such as Domino in the Expertise field, when the Get Expertise button is clicked, the application returns the name of the person with the expertise for the product. The result window is illustrated on Figure 3-2.

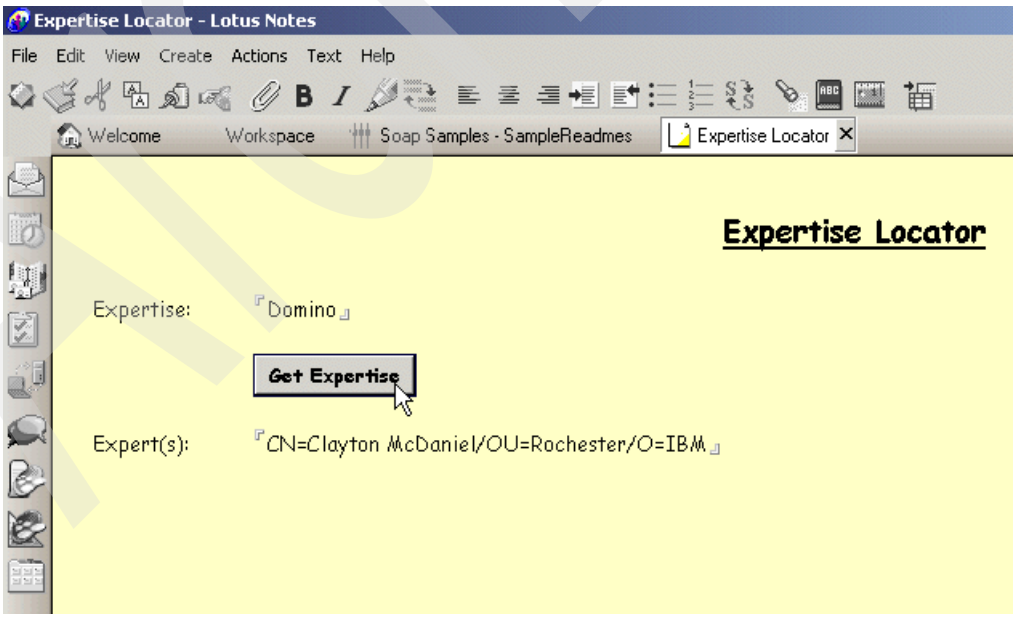


Figure 3-2 ExpertiseLocator LotusScript application return window example

All the data accessed by this application is in Domino database people.nsf.

Archive

3.2 Porting Domino application to Java application

Now we port this ExpertiseLocator LotusScript application into a Web Service Java application that is called ExpertiseLocator.java.

3.2.1 Total transformation flow

This section discusses the total process of the transformation of the Domino application into a Web Service Java application.

Figure 3-3 illustrates the technology view of the existing Domino application. ExpertiseLocator, as highlighted in the figure, is a LotusScript application serving the Notes client's requests through forms.

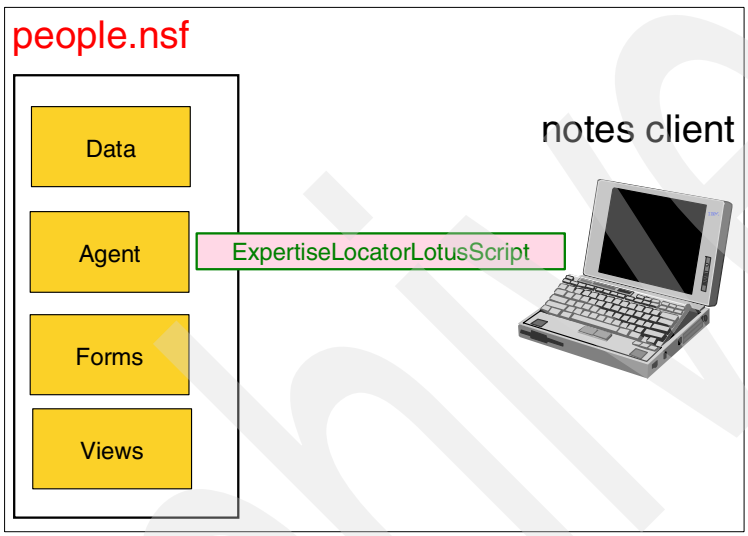


Figure 3-3 Technology view of pre-transformation ExpertiseLocator application

Our first task is to port the LotusScript application into a Java application as depicted by the dotted red arrows in Figure 3-4.

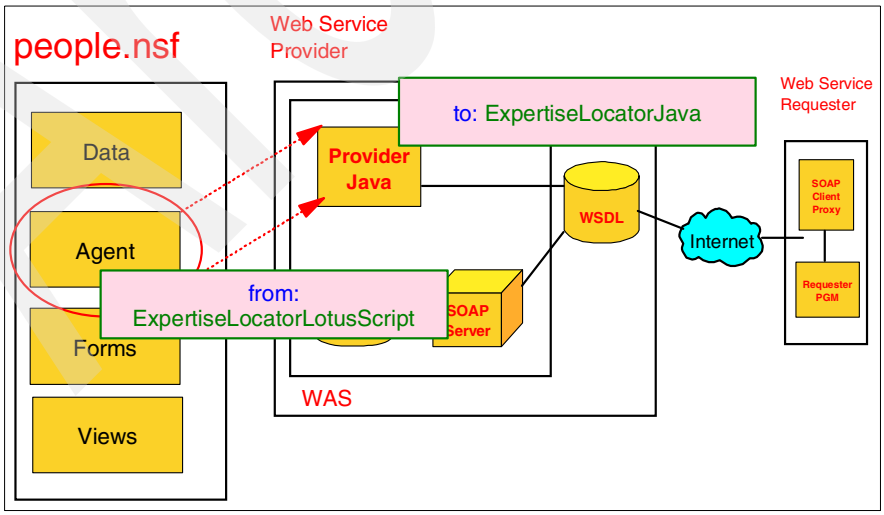


Figure 3-4 Porting of LotusScript application into Java application

3.2.2 Porting tips

We will discuss the practical porting tips in this section. First, Example 3-1 shows the example code of a LotusScript file, ExpertiseLocator in this case.

Example 3-1 ExpertiseLocator LotusScript application

```
Sub Click(Source As Button)
   Dim workspace As New NotesUIWorkspace
   Dim uidoc As NotesUIDocument

   Dim db As NotesDatabase
   Dim profileDoc As NotesDocument
   Dim searchFormula As String
   Dim dc As NotesDocumentCollection
   Dim dateTime As New NotesDateTime("12/01/94")
   Dim TaxonomyTerm$
   Dim experts( ) As String
   Dim i As Integer

   Set uidoc = workspace.CurrentDocument
   Set session = New NotesSession Tip 1
   Set db = session.GetDatabase("idomws1", "lek\people.nsf") Tip 2

   TaxonomyTerm$ = """ & uidoc.fieldGetText("Expertise") & """ Tip 3
   searchFormula$ = "@IsMember(" & TaxonomyTerm$ & "; AffinityPublishedLeaf)"
   Set dc = db.search(searchFormula$, dateTime, 0) Tip 4
   if Not dc.count = 0 then
      Redim Preserve experts(dc.count-1)
      Set profileDoc = dc.getFirstDocument() Tip 5
   else
      Goto finish
   End If

   If profileDoc Is Nothing Then
      errorCode = 1
      errorMessage = "Taxonomy term could not be found"
      Goto finish
   Else
      While Not profileDoc Is Nothing
         experts(i) = profileDoc.FullName(0)
         i=i+1
         Set profileDoc = dc.getNextDocument(profileDoc)
      Wend
      i=0
   End If

   Forall persons In experts
      returnMSG = returnMSG & persons & Chr(10)
   End Forall

   Call uidoc.fieldClear("expert")
   Call uidoc.fieldAppendText ("expert", returnMSG)
   End
finish:
   messageBox errorMessage, MB_OK + 16, “Error Encountered”
End Sub
```

Archived

Example 3-2 ExpertiseLocator Java application

```
package com.lotus.kds.ws;

/**
 * Insert the type's description here.
 * Creation date: (6/6/01 9:46:20 AM)
 * @author: Administrator
 */
import lotus.domino.*;
import java.util.*;
import com.lotus.kds.ws.*;

public class ExpertiseLocator {
/**
 * ExpertiseLocator constructor comment.
 */
public Expert getExpertise(String taxonomyTerm) {

    try {


        //output parameter
        int errorCode = 0;
        String errorMessage = "";

        //declararion
        Vector experts = new Vector();
        Session mainSession = Tip 1
            NotesFactory.createSession(
                "9.5.92.44",
                "Vinit Saraswat/itso",
                "password");

        Database peopleDB = Tip 2
            mainSession.getDatabase(mainSession.getServerName(), "lek\\people.nsf");

        String formula = "@ismember(\"" + taxonomyTerm + "\";AffinityPublishedLeaf)";Tip 3

        DocumentCollection dc = peopleDB.search(formula); Tip 4

        Document profileDoc = dc.getFirstDocument();Tip 5

if (profileDoc == null) {
            errorCode = 1;
            errorMessage = "Taxonomy term could not be found";

        } else {
            while (profileDoc != null) {
                experts.add(profileDoc.getItemValue("FullName").elementAt(0).toString());
                profileDoc = dc.getNextDocument(profileDoc);
            }
        };

        if (experts.equals("")) {
            errorCode = 2;
            errorMessage = "No expert could be found for the taxonomy term " +
taxonomyTerm;
        };

        Expert expert =
```

Archived

```
            new Expert(
               experts.toString().substring(1, experts.toString().length() - 1),
                errorCode,
                errorMessage                );
        mainSession.recycle();
        return (expert);

    } catch (NotesException n) {
        System.out.println("NotesError: " + n.text);

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

    }

    return null;
}
}
```

Session creation description

Tip 1 on Example 3-1 on page 64 shows how you create session in LotusScript. Let us repeat the exact statement here:

```
Set session = New NotesSession
```

Tip 1 on Example 3-2 on page 65 shows how you create session in Java:

```
Session mainSession =
   NotesFactory.createSession
      “iDomWS1”,
      “Vinit Saraswat/itso”,
      “password”);
```

You don’t need any further steps than setting the session in LotusScript, because all access is local in the database. This is not the case in Java. In the Java class we need to create a Notes session externally. So we use NotesFactory class to create a session and for it we specify: “idomws1”, as the Domino server address, “Vinit Saraswat/itso”, as the Lotus user name, “password”, as the Lotus user password.

Get database access

As highlighted by **Tip 2** in Example 3-1 on page 64, the following example shows how to get database access in LotusScript. You specify the Domino server name, “idomws1” and the directory path of the database with the database name itself, “lek\people.nsf”.

```
Set db = session.GetDatabase("idomws1", "lek\people.nsf")
```

Now in Java, we get the Domino server name from a previously created session as shown in the following example, or as highlighted by **Tip 2** in Example 3-2 on page 65. The directory path and database name description remains similar.

```
Database peopleDB =
        mainSession.getDatabase(mainSession.getServerName(), "lek\\people.nsf");
```

Specify search parameters

We then set up the search parameters. As highlighted by **Tip 3** in Example 3-1 on page 64, we create a text string for TaxonomyTerm then use it to formulate the search string.

```
TaxonomyTerm$ = """" & uidoc.fieldGetText("Expertise") & """"
searchFormula$ = "@IsMember(" & TaxonomyTerm$ & "; AffinityPublishedLeaf)"
```

In Java, this can be done a little simpler as shown in the following example, which is a highlighted **Tip 3** in Example 3-2 on page 65.

```
String formula = "@ismember(\"" + taxonomyTerm + "\";AffinityPublishedLeaf)";
```

By the way, every person in the database has a profile document. AffinityPublishedLeaf is the field in each profile document, which contains the expertise for a given person.

Search database

The following example, which is highlighted by **Tip 4** in our LotusScript example, shows database search statement.

```
Set dc = db.search(searchFormula$, dateTime, 0)
```

The following example, which is highlighted by **Tip 4** in our Java example, shows a database search statement in Java.

```
DocumentCollection dc = peopleDB.search(formula);
```

Get result document

The following example, which is highlighted by **Tip 5** in our LotusScript example, shows the statement to get the result document.

```
Set profileDoc = dc.getFirstDocument()
```

The following example, which is highlighted by **Tip 5** in our Java example, shows the statement to get the result document.

```
Document profileDoc = dc.getFirstDocument();
```

Package result document

Now we have the result document and we will package it. The following example below, as highlighted by **Tip 6** in our LotusScript example, shows how this can be done in LotusScript.

```
While Not profileDoc Is Nothing
      experts(i) = profileDoc.FullName(0)
      i=i+1
      Set profileDoc = dc.getNextDocument(profileDoc)
   Wend
```

The portion above will populate the experts array with the details stored in profileDoc document.

```
Forall persons In experts
   returnMSG = returnMSG & persons & Chr(10)
   End Forall
```

Then the portion above will package the names of experts as one string and return it.

The following example below then, as highlighted by **Tip 6** in our Java example, shows how this can be done in Java.

```
while (profileDoc != null) {
            experts.add(profileDoc.getItemValue("FullName").elementAt(0).toString());
            profileDoc = dc.getNextDocument(profileDoc);
        }
```

The portion above will populate the experts vector array with the details stored in profileDoc document.

Archived

```
Expert expert =
         new Expert(
            experts.toString().substring(1, experts.toString().length() - 1),
             errorCode,
             errorMessage                );
     mainSession.recycle();
     return (expert);
```

The above portion is quite different from the way we did in LotusScript. Instead of returning the results as string, we use a separate Java class file called ‘expert’. We don’t have to do it this way but the example application we use, ExpertiseLocator, just happens to use a class file to pass the values. For this technique, we need to use the customized Java to XML mappings, also known as customized serializer.

3.3 Deploying ported Java application using WDSc

Now we have ported ExpertiseLocator LotusScript application into a Java application. Next task will be deploying this ported Java application as WebSphere’s Web application. We will use WDSc as deployment tools.

The steps covered in this section are very similar to those covered in 2.2, “Developing and deploying the applications” on page 14. The only difference here is we use the customized Java to XML mappings, also known as customized serializer.

3.3.1 ExpertiseLocatorApp creation — step-by-step

We will walk you through the process of creating a new Web application out of a newly ported ExpertiseLocator Java application.

Create a new project

Here we describe how to create a new project for Expertise Locator Web Service.

1. Start your WDSc and create a new project as shown on Figure 3-5.

Archived

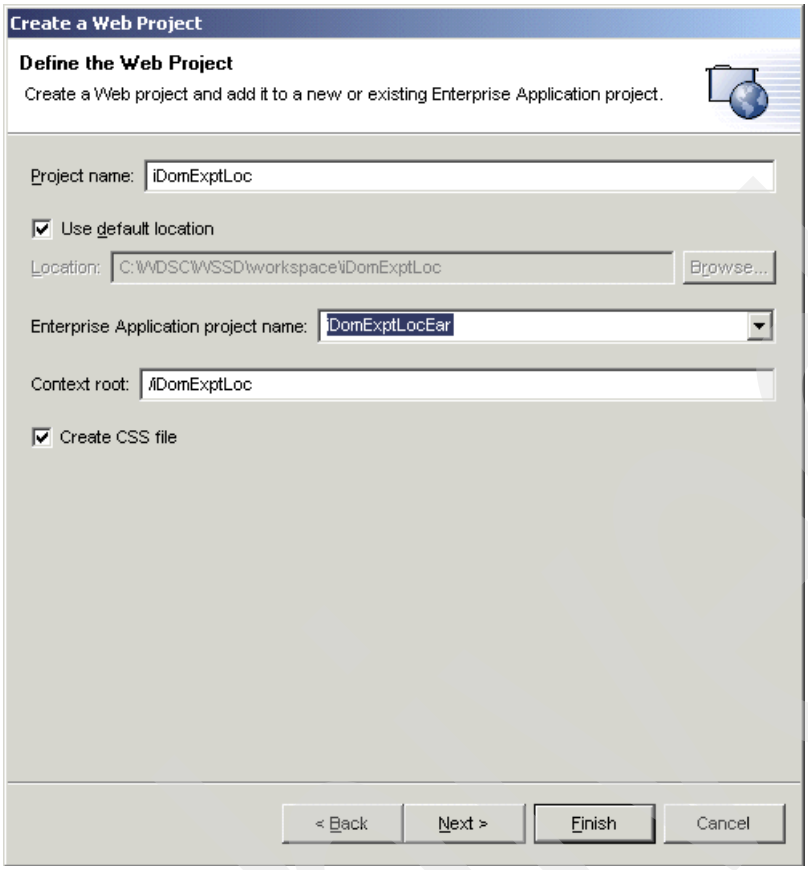


Figure 3-5 Specifying project name, context root and EAR project file in WDSc

Enter this information:

- Project Name -> iDomExptLoc
- Enterprise Application Project Name -> iDomExptLocEar
- Context Root -> /iDomExptLoc

Check the boxes for **use default location** and **create CSS file**.

Note: The context root has to be entered with **"/"** because this will map to the directory path in your Web service when you deploy it on the WebSphere Application Server. The context root will make up the URL of the Web Service.

2. Make sure you add the required external jar files like NSCOW.jar, soap.jar etc. to iDomExptLoc project, as shown in Figure 3-6. Click Finish to complete the creation of the WebProject.

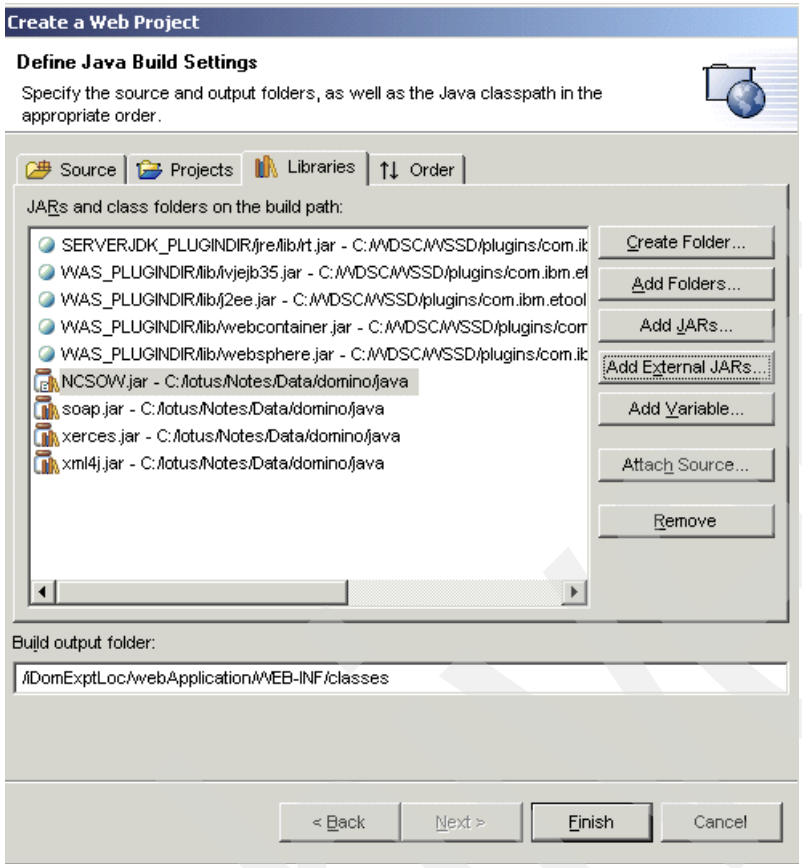


Figure 3-6 Import required Jar files in iDomExptLoc project in WDSc

3. After creation of the Web Project, click **Perspective -> Open -> Web** and expand the iDomExptLoc Project to look at the directory structure as depicted in Figure 3-7 and Figure 3-8.

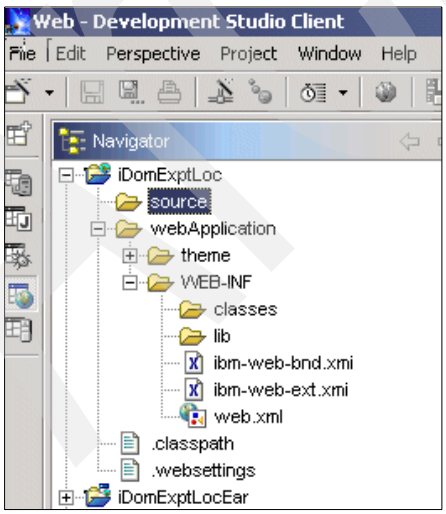


Figure 3-7 Directory Structure of WebProject in iDomExptLoc

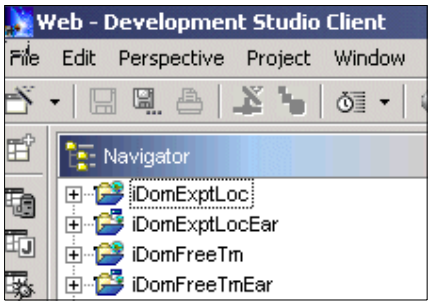


Figure 3-8 Source folder in iDomExptLoc directory in WDSc

Import required Java files in iDomExptLoc Project

In the source folder of iDomExptLoc we will import our source files for creating a Web Service application. As discussed earlier, we use some of the applications and files provided by LWSEK for our case study. Figure 3-9 shows the folder structure of LWSEK and what you get from it. We will copy some of the Java files.

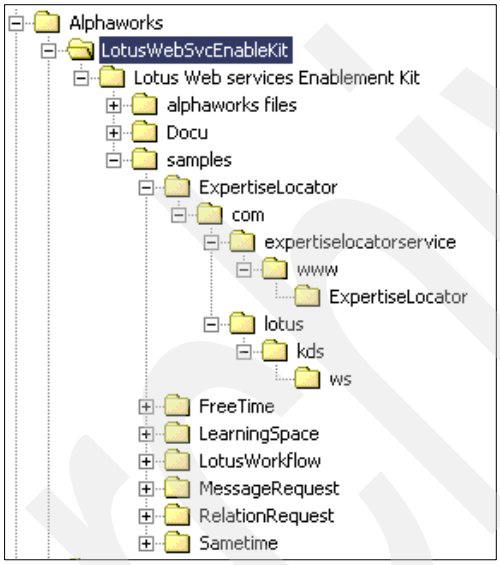


Figure 3-9 ExpertiseLocator samples directory structure in LWSEK

Note: We are not showing all the windows required in these steps. For a detailed look, refer to 2.2, “Developing and deploying the applications” on page 14.

1. On your WDSc main window, select **File -> Import** to get the window shown in Figure 3-10.

Archived

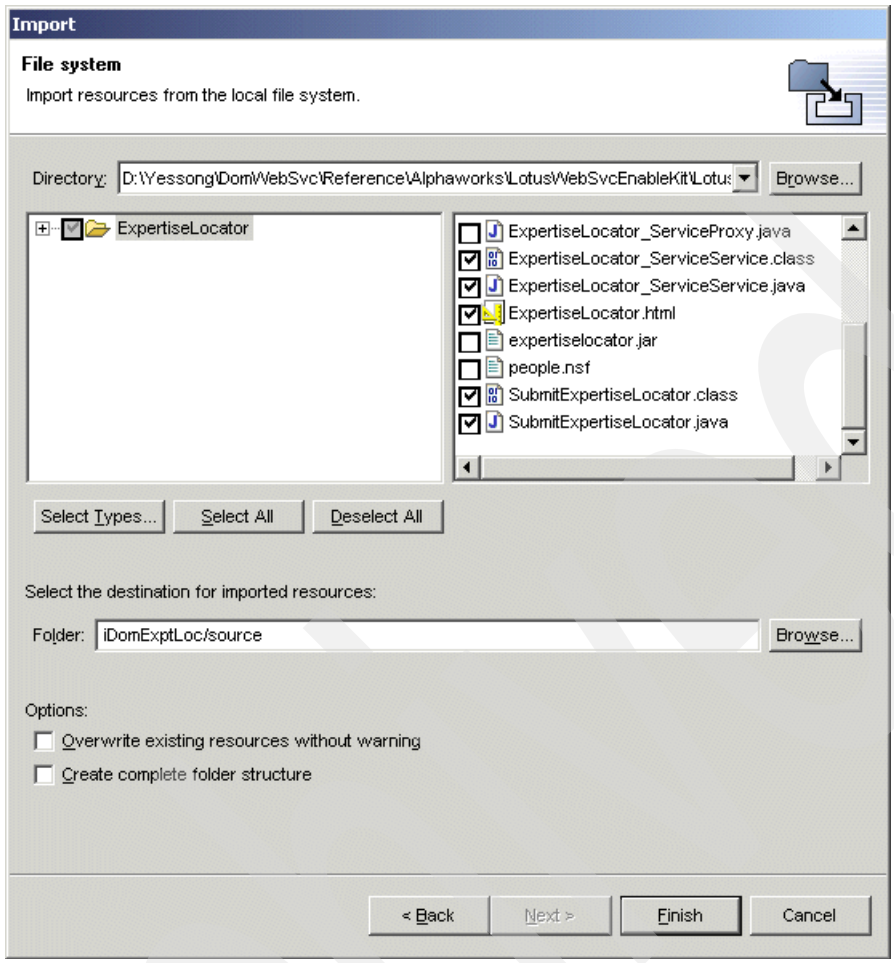


Figure 3-10 Check on ExpertiseLocator_ServiceSerivce.java and other files to be imported in WDSc

a. Let's say your local drive where you downloaded LWSEK is **X:**. Select LWSEK folder, for example:

   - **X:\LotusWebSvcEnableKit\Lotus Web services Enablement Kit\samples\ExpertiseLocator**

b. Click the following files to import:

   - **ExpertiseLocator_ServiceService**, both .java and .class files: This is the controller class which calls ExpertiseLocator Java application which we ported in 3.2, "Porting Domino application to Java application" on page 63.
   - **SubmitExpertiseLocator**, both .java and .class file: This is the servlet which accepts the request from the Web Service requester and calls the controller program, ExpertiseLocator_ServiceService.
   - **ExpertiseLocator.html**: This file lives on the server side and is sent to the browser upon their Web Service's request.

2. Now we will import some files required to handle the customized serialization, which is another way of saying customized Java to XML mapping. We need these classes apart from the core functionality class ExpertiseLocator_ServiceService, because the Web Service returns an "Expert" Java object as return type. For Apache SOAP to decode the response from this "Expert" Java object, we need to use ExpertSerializer class. See "Serialization concept for Apache SOAP" on page 92 for more details. Figure 3-11 illustrates the selection of the required files.

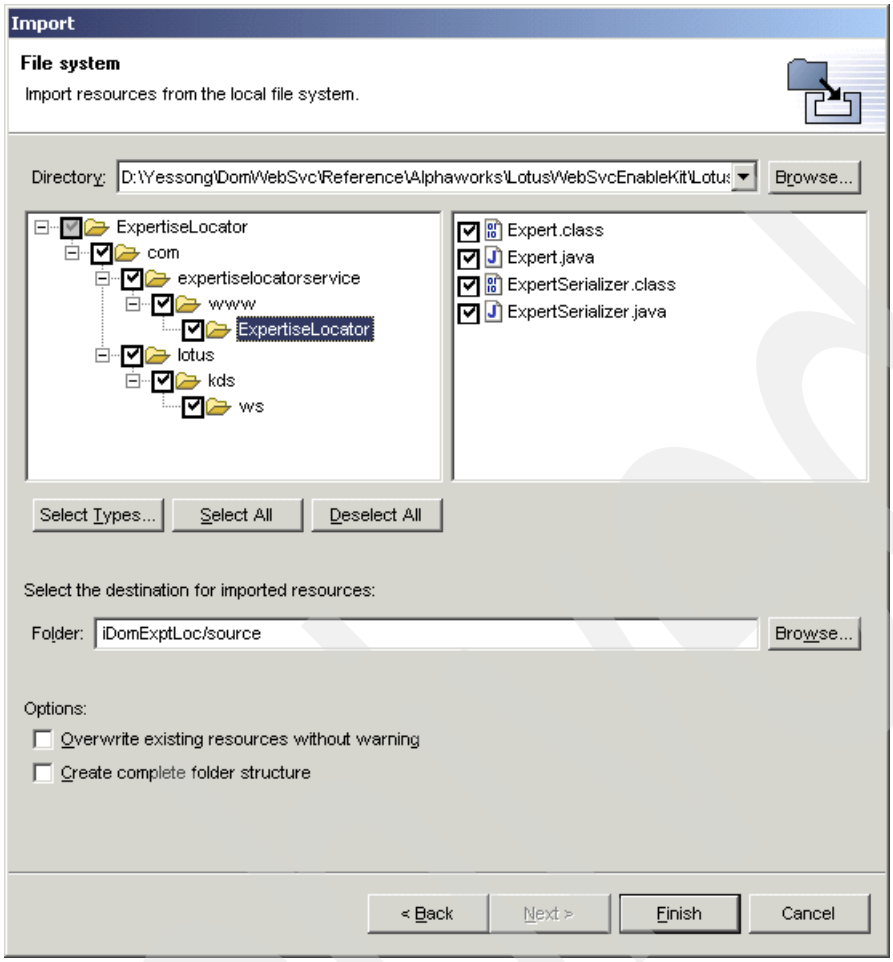


Figure 3-11 Import ExpertSerializer and Expert classes in iDomExptLoc project in WDSc

a. As illustrated in Figure 3-11, select the following files to import them in iDomExptLoc project:

- **com.expertiselocatorservice.www.ExpertiseLocator.Expert** .java and .class files
- **com.expertiselocatorservice.www.ExpertiseLocator.ExpertSerializer** .java and .class files

b. We also import the similar set of files from **lotus.kds.ws** package, as shown in Figure 3-12. Select:

- **lotus.kds.ws.Expert** .java and .class files
- **lotus.kds.ws.ExpertiseLocator .**java and .class files

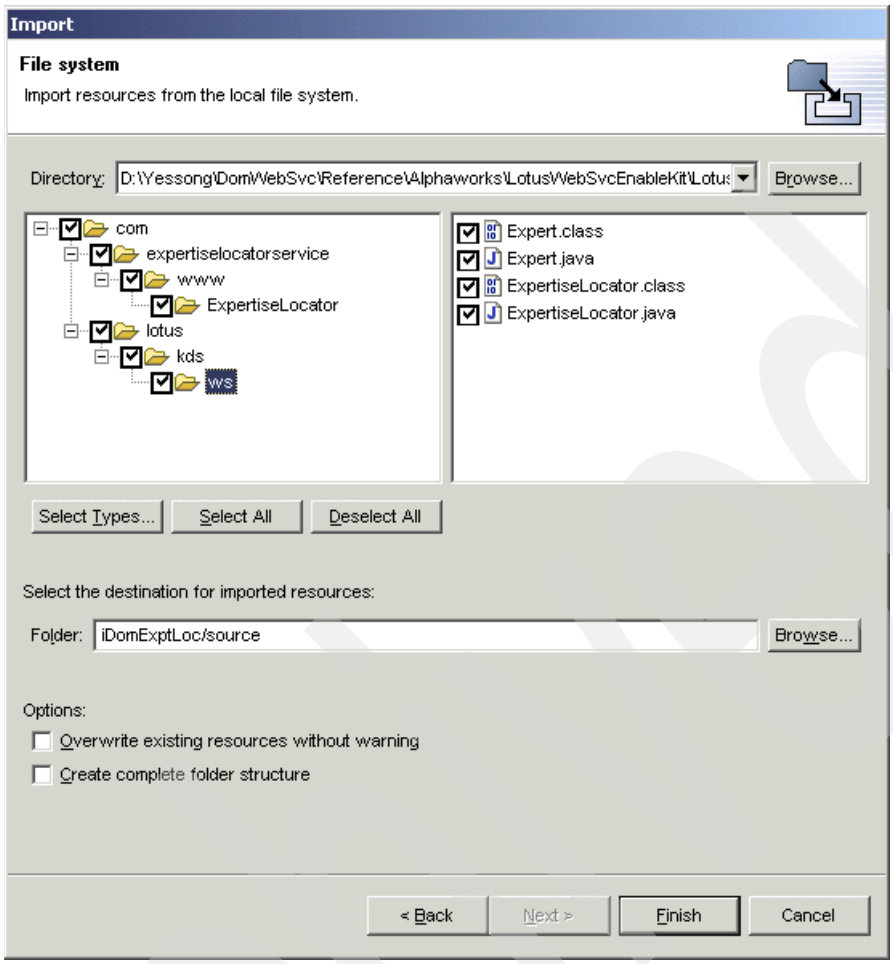


Figure 3-12 Import from com.kds.ws package in WDSc

Create a new Web Service

Now we have completed the Web Project creation. It's time to create a new Web Service from this. Again, we show some sample window shots. For a full cycle walkthrough of the Web Service creation, refer to 2.2, "Developing and deploying the applications" on page 14.

1. On your WDSc main window, select **File -> New -> Other -> Web Services -> Web Service** to create a Web service. In the Web Service Create window, click **Next** to go to the Web Service options page as shown on Figure 3-13.

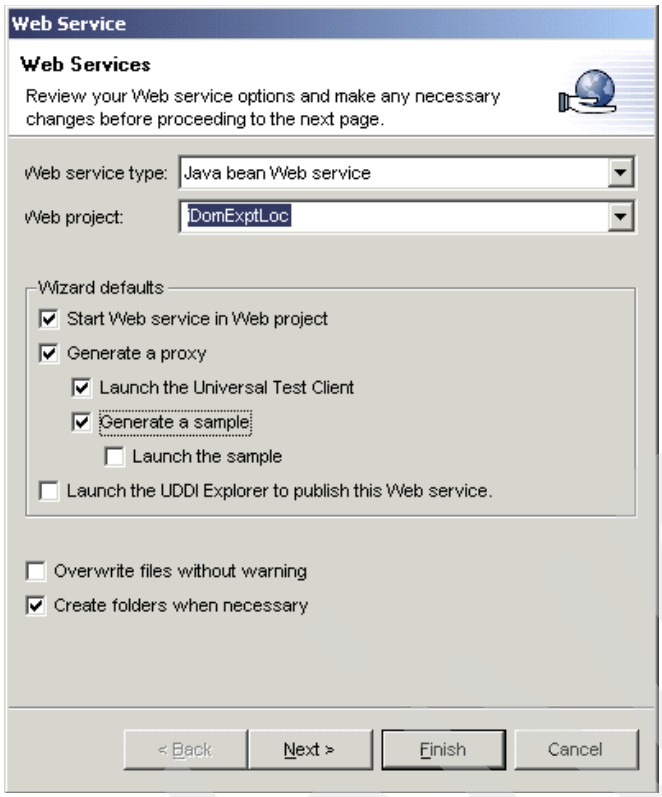


Figure 3-13 Create Web Service from ExpertiseLocator_ServiceService.java and other Java programs

2. As given in Figure 3-13, for the Web Service create options page, make sure **Web service type** is **Java Bean Web Service**. Also the Web project name should be correct. Check on options **Wizard defaults** - **Start Web service in Web project**, **Generate Proxy**, **Launch Universal Test Client**, **Generate the sample**. Also check the **Create folders when necessary**. Leave the other options unchecked and click **Next** to get the Web Service Java Bean Selection window.
3. On the Web Service Java Bean Selection window, browse classes to get the window shown in Figure 3-14. Enter **ExpertiseLocator_ServiceService** Java bean name in browser to select the class.

Archived

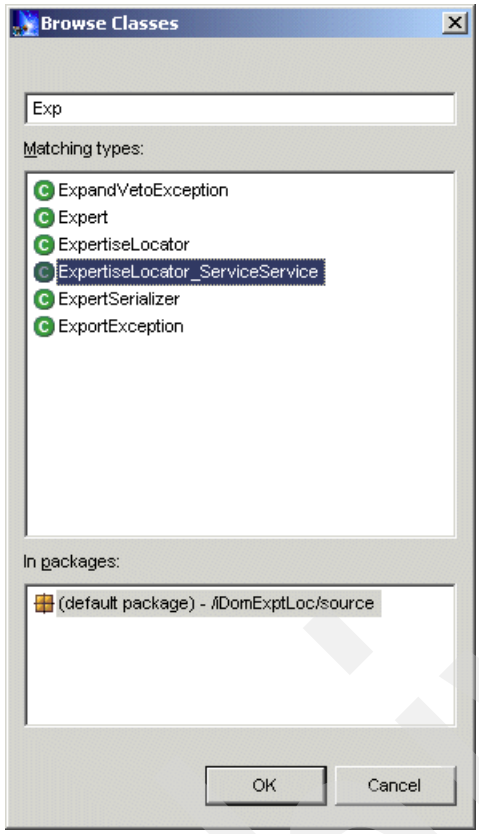


Figure 3-14 Select ExpertiseLocator_ServiceService Java bean in browse classes on WDSc

4. As depicted in Figure 3-15, select a name for the Web Service URI on the option window of the Web Service Java Bean Identity. This URI can be in the form of URL **http://tempuri.org/ExpertiseLocatorService** or it can be **urn:StockQuoteService**. Here we choose notation **urn:StockQuoteService**. There is no special reason for that. Notations are identified to Web Service components like proxy client as XML namespace representation. You will also notice a few WSDL files in Figure 3-15. These WSDL files define the framework of ExpertiseLocator Web Service in XML files such as ExpertiseLocator_ServiceService-service.wsdl. Leave these options as defaults and click **Next**.

Archived

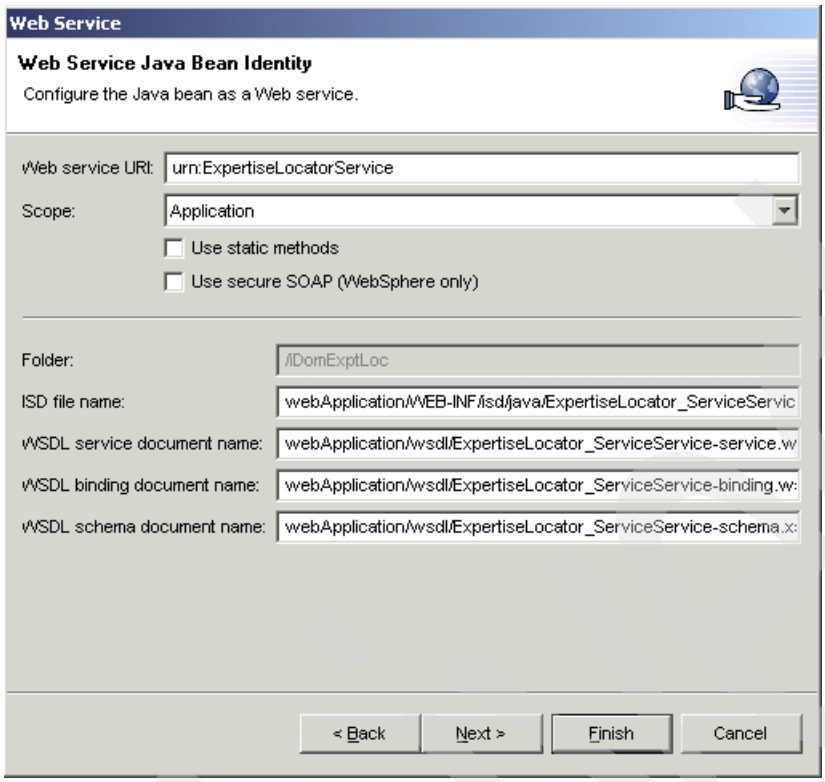


Figure 3-15 Specify the Web Service Java Bean identity on WDSc

5. You will see methods of your Web Service Java Bean appear in the selections in Figure 3-16. Make sure that **Input encoding for getExpertise** method and **Output encoding for getExpertise** method are both checked for SOAP encoding. This is because we are building the Web Service on SOAP encoding envelope. Check **Show Server (Java to XML) type mappings** and click **Next**.

Archive

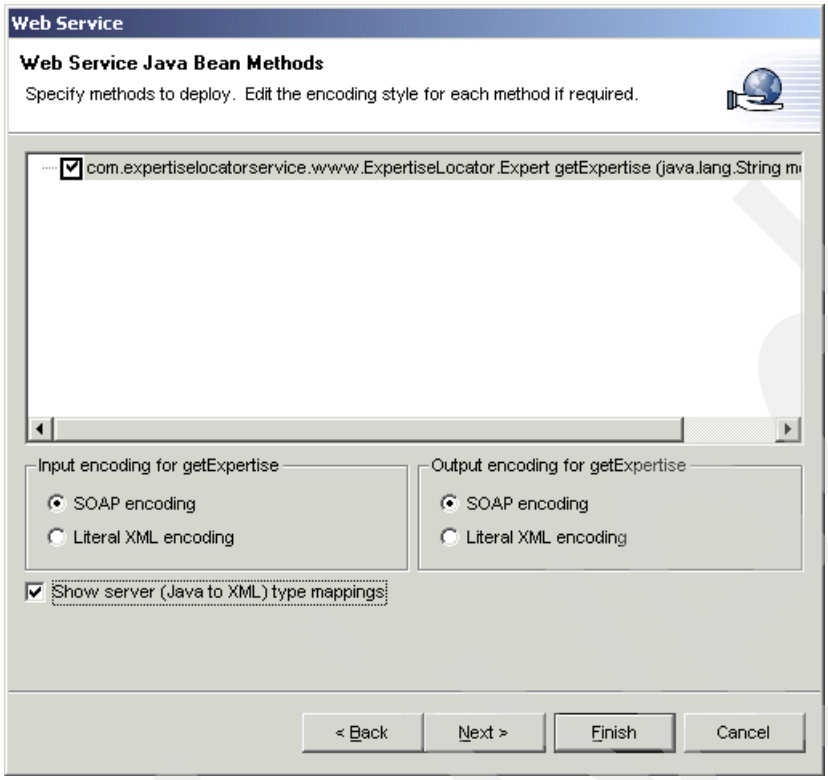


Figure 3-16 Specify encoding for Web Service Java Bean Methods in WDSc

6. Figure 3-17 shows the details of the ExpertiseLocator_ServiceService Java bean method argument and return type with its Web Service encoding style. In this case **com.expertiselocatorservice.www.ExpertiseLocator.Expert, SOAP encoding** depicts the return type and **java.lang.String, SOAP encoding** depicts call argument type for StockQuote Web Service. For java.lang.String SOAP encoding we can have the radio button **Show and use the default Java bean mapping** checked. But for Expert, SOAP encoding you need to have customized mappings by clicking on **Edit and use a customized mapping**. Customized mappings are specified when you want to use specialized classes in a call argument or return type method of the Web Service. For all other primitive data types like int, float, and predefined Java runtime object class like java.lang.String you can use default mappings.

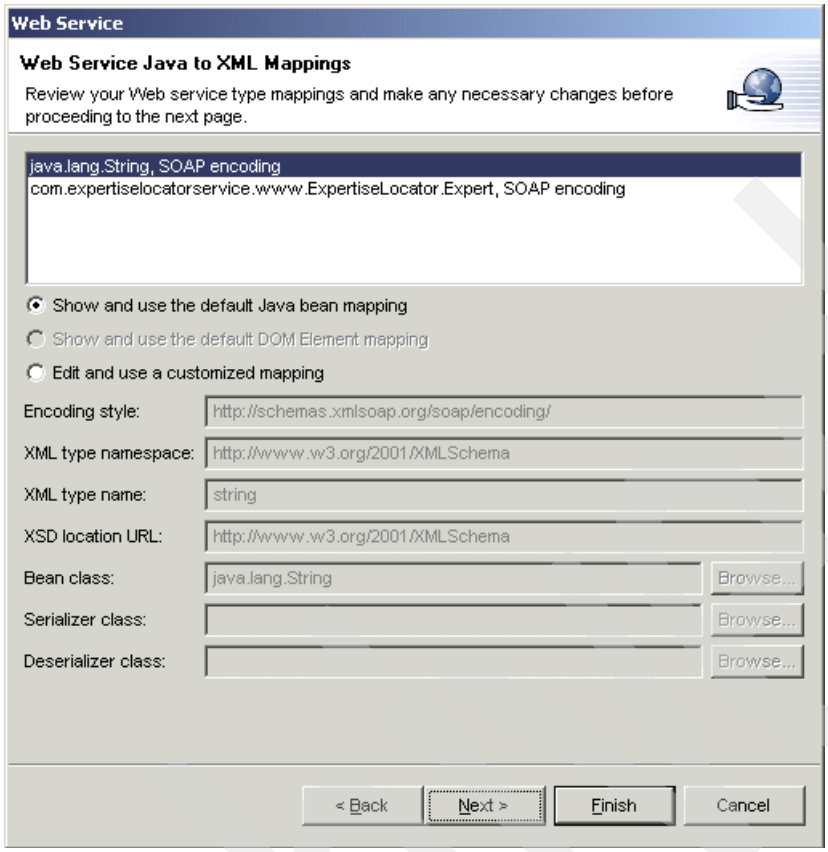


Figure 3-17 Verify Java to XML Mappings for Web Service on WDSc for SOAP request

Note: By clicking **Edit and use a customized mapping** for Expert SOAP encoding, browse and select ExpertSerializer class as shown in Figure 3-18. This important step should not be forgotten otherwise you will get SOAP fault exceptions for seralization/deserialization.

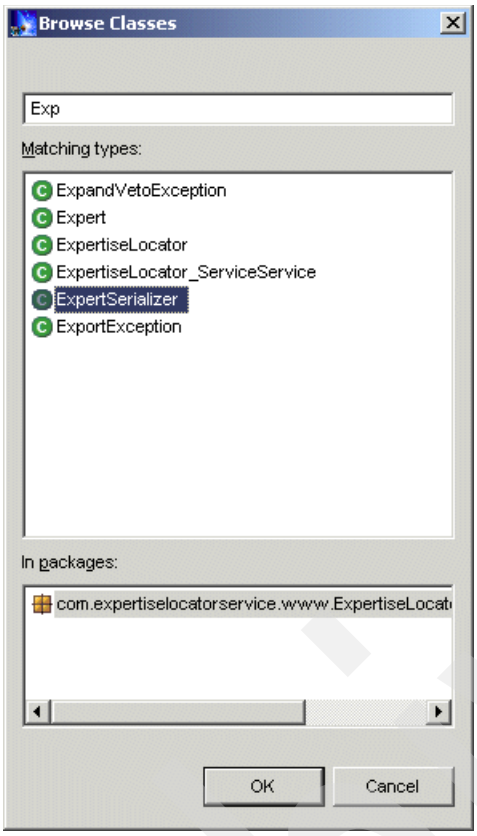


Figure 3-18 Select ExpertSerializer class for customized SOAP encoding in WDSc

7. As you can see in Figure 3-19, select the deserializer class also as **ExpertSerializer.class**, as set on SOAP encoding customized mapping.

Archived

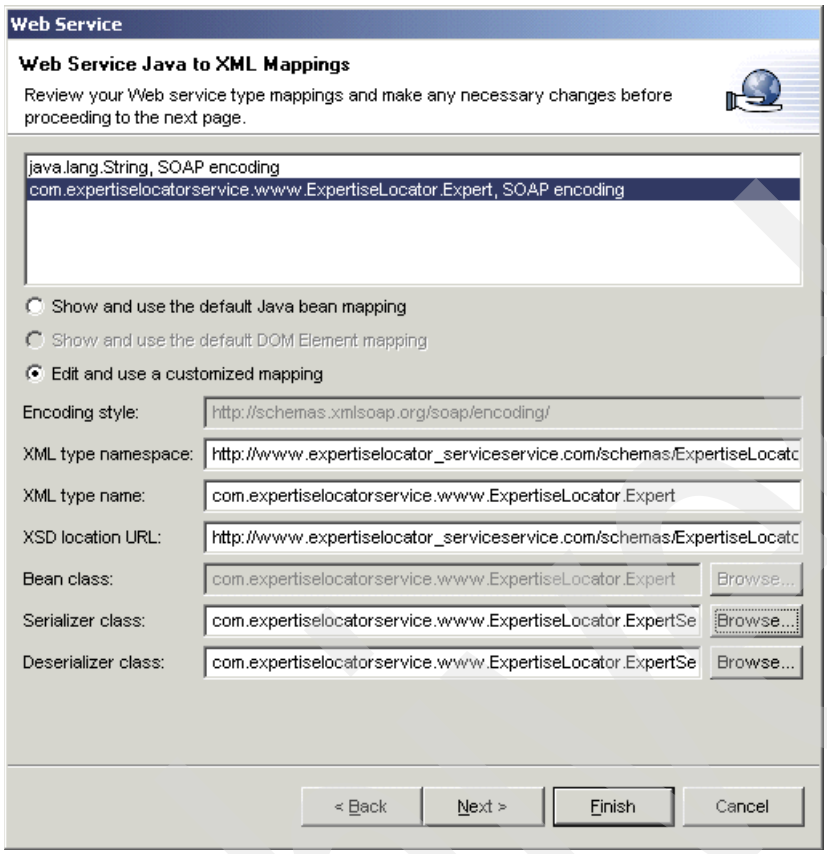


Figure 3-19 Customized mappings after selection of Serializer and Deserializer classes in WDSc

8. In Figure 3-20, you will see the binding type that will be generated for the Expertise Locator Web Service. This binding will be stored in WSDL files and make the framework of the Web Service. To consume the Web Service, a proxy client is also generated in **iDomExptLoc/Source** folder and named as class **proxy.soap.ExpertiseLocator_ServiceServiceProxy**. Leave these options as defaults and check the **Show mappings** box, click **Next** to continue.

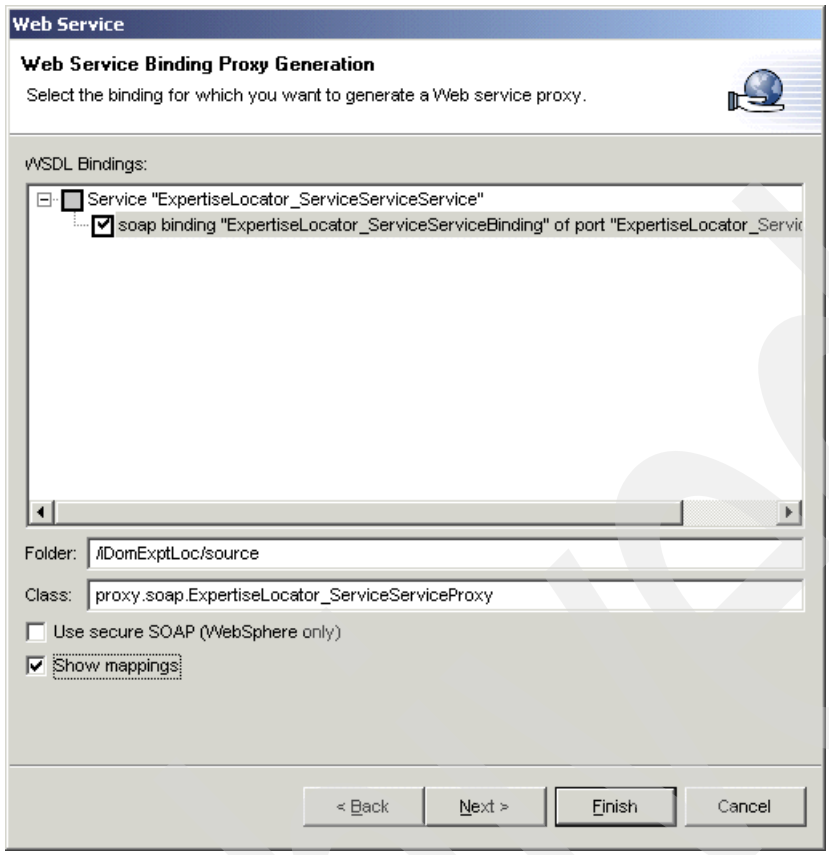


Figure 3-20 Verify Web Service Binding Proxy Generation option in WDSc

9. This Figure 3-21 let's you verify whether the XMLSchema mapping and Java Class mapping for a particular input or output parameter is appropriate. Also here you get a chance to verify the mappings. In our case, we have a return type SOAP method call set to Expert class. Make sure that these settings are not default data type if you are using customized mappings and click **Next**.

Archived

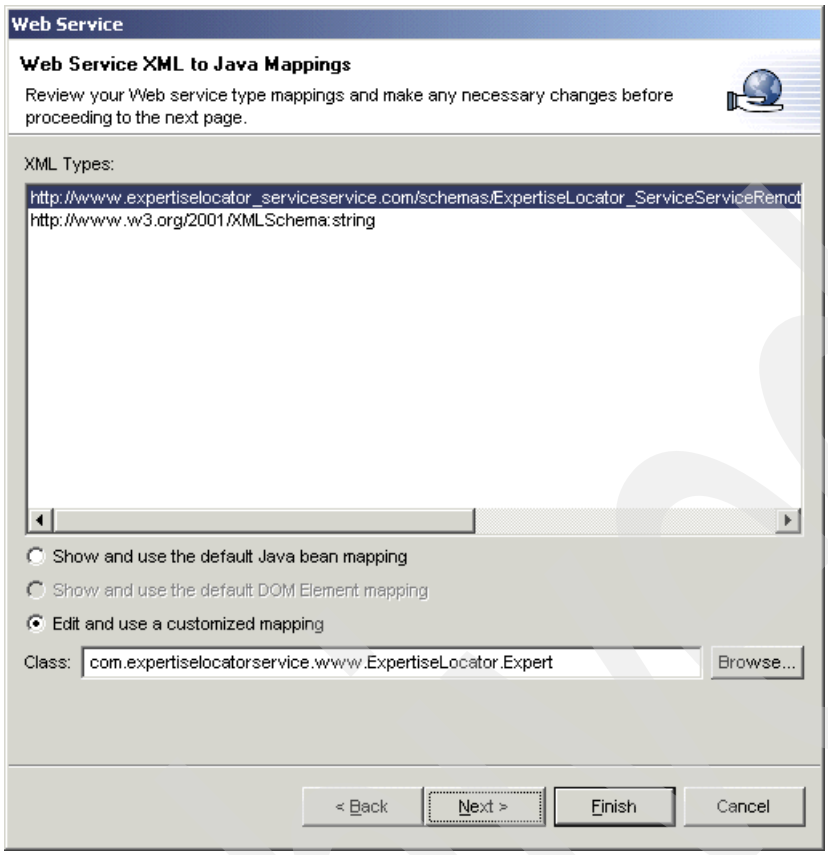


Figure 3-21 Verify the input and output parameter Schema mappings for Web Service in WDSc

10. Figure 3-22, shows XML to Java mappings for SOAP parameter java.lang.String. In this case default Java bean mapping is used.

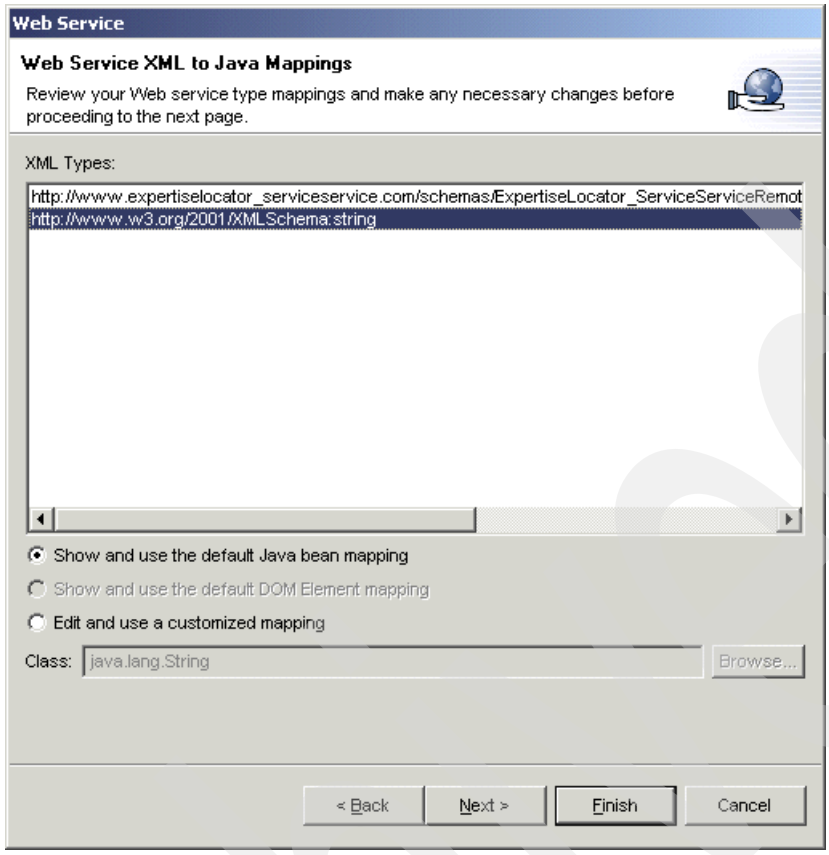


Figure 3-22 Verify the input and output parameter Schema mappings for Web Service in WDSc

11. Figure 3-23 is another figure for cross verifying the configuration with SOAP encoding types for parameters and return types in the Web Service method. In our case, since our return parameter is specialized class we populate Serializer and Deserializer class fields. Leave everything as it is and click **Next**.

Archived

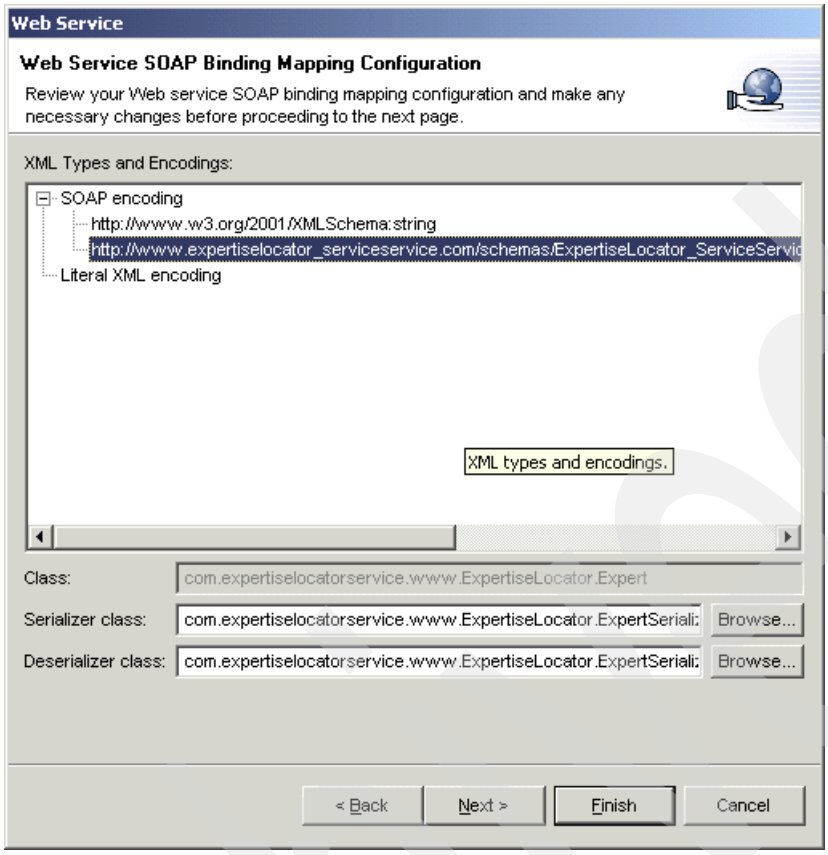


Figure 3-23 Web Service SOAP Binding configuration in WDSc

12. In Figure 3-24, you specify the sample generation option by checking **Generate sample**, and the sample TestClient.jsp will be generated in **JSP folder**, as specified in Figure 3-24. Click **Next** and then **Finish** to generate the Web Service. This will complete the steps of the Web Service creation.

Archived

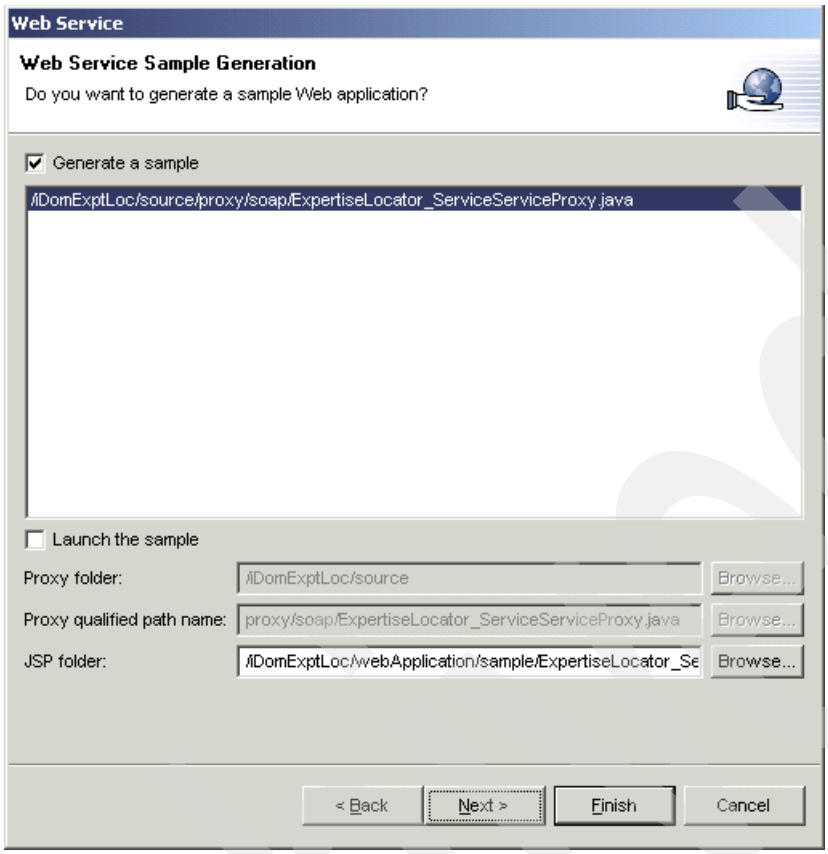


Figure 3-24 Web Service Sample Generation option for Web Service in WDSc

Change the Web Service configuration files for deployment on WAS

Now the Web Service is created. The next step is to change the Web Service configuration files for deployment on WebSphere Application Server.

1. In Figure 3-25, you will see the how to customize the newly created Expertise Locator Web Service for deployment on WebSphere Application Server. Click the **WSDL** folder in the **Web perspective** view of iDomExptLoc project. Double-click the **ExpertiseLocator_ServiceService-service.wsdl** file from the **WSDL** folder. Expand the **definitions**, **import** and **service** portion of WSDL file. Change the **targetNameSpace**, **xmlns:tns**, **import > location**, **service > soap:address > location** from **localhost** to **idomws1:5500**. Here **idomws1** is the Domino HTTP Server on iSeries and **5500** is the port number on which our instance of application server is running on WebSphere Application Server. After completing changes in WSDL files change save it by clicking **File -> Save**.

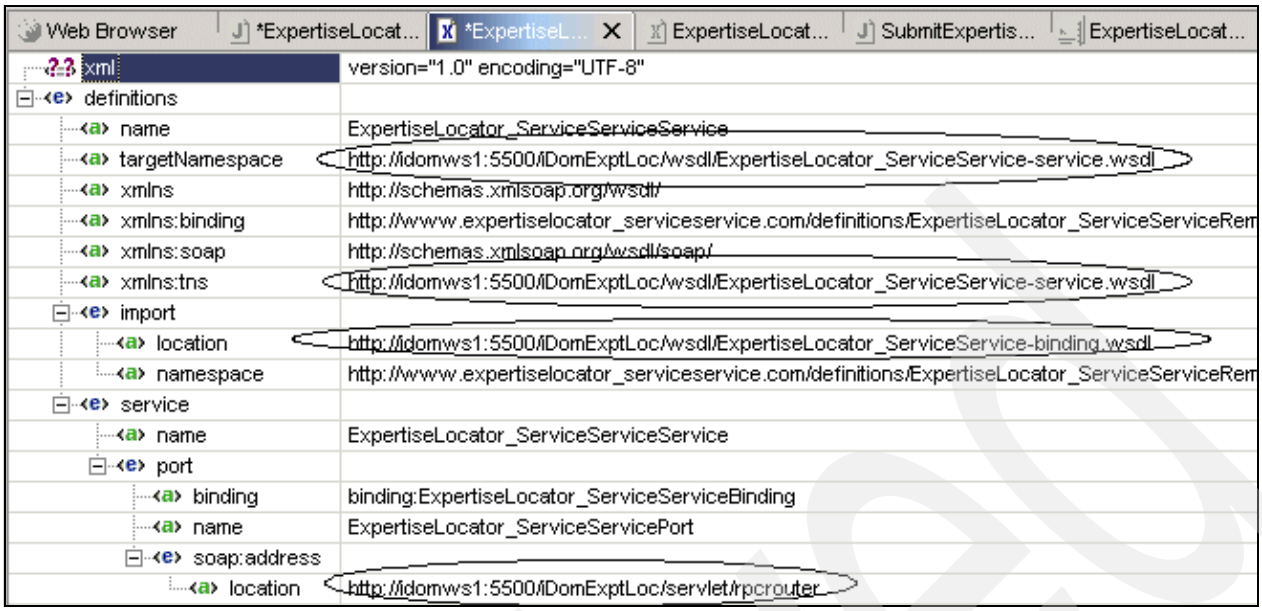


Figure 3-25 Make Changes in WSDL files on WebProject to point to WebSphere Application Server

2. Also change the ExpertiseLocator_ServiceServiceProxy.java in proxy/soap folder under source, as shown in Figure 3-26. In the StockQuoteServiceProxy code make sure the **http://localhost/iDomExptLoc/servlet/rpcrouter** is changed to **http://idomws1:5500/iDomExptLoc/servlet/rpcrouter**.

```
public class ExpertiseLocator_ServiceServiceProxy
{
  private Call call = new Call();
  private URL url = null;
  private String stringURL = "http://idomws1:5500/iDomExptLoc/servlet/rpcrouter";
  private SOAPMappingRegistry smr = call.getSOAPMappingRegistry();

  public ExpertiseLocator_ServiceServiceProxy()
  {
  }

  public synchronized void setEndPoint(URL url)
  {
    this.url = url;
  }

  public synchronized URL getEndPoint() throws MalformedURLException
  {
    return getURL();
  }
```

Figure 3-26 Change ExpertiseLocator_ServiceServiceProxy to point to your WAS instance in WDSc

3. As shown in Figure 3-27, browse the source of **com.kds.ws.ExpertiseLocator** in Expertise Locator Web Service and change it to point to your Domino server where you have installed people.nsf database.

Archive

```
: Expert getExpertise(String taxonomyTerm) {

:y {


   //output parameter
   int errorCode = 0;
   String errorMessage = "";

   //declararion
   Vector experts = new Vector();
Session mainSession =
      NotesFactory.createSession(
           "iDomWS1",
           "Vinit Saraswat/itso",
           "password");

   Database peopleDB =
      mainSession.getDatabase(mainSession.getServerName(), "lek\\people.nsf");

   String formula = "@ismember(\"" + taxonomyTerm + "\";AffinityPublishedLeaf)";

   DocumentCollection dc = peopleDB.search(formula);
```

Figure 3-27 Change the com.kds.ws.ExpertiseLocator to point to your Domino server in WDSc

Export the Web Service for deployment on WAS

Now we are ready to export the Web Service for deployment on WebSphere Application Server. As Figure 3-28 shows, click the **EAR file** as export type for iDomExptLoc project Enterprise Application. Make sure that the option **what resources do you want to export** has **WebProjectEar** selected on the **Where do you want to export resources to?** option. Browse and select the IFS directory to iSeries WebSphere Application Server install, for example, **y:/QIBM/UserData/WebAsAdv4/iwebsvc/installableApps/iDomExptLoc.ear**, where **y:** is the drive mapped to your iSeries. Also check the **Export source files** option. Click **Finish** to export.

Archived

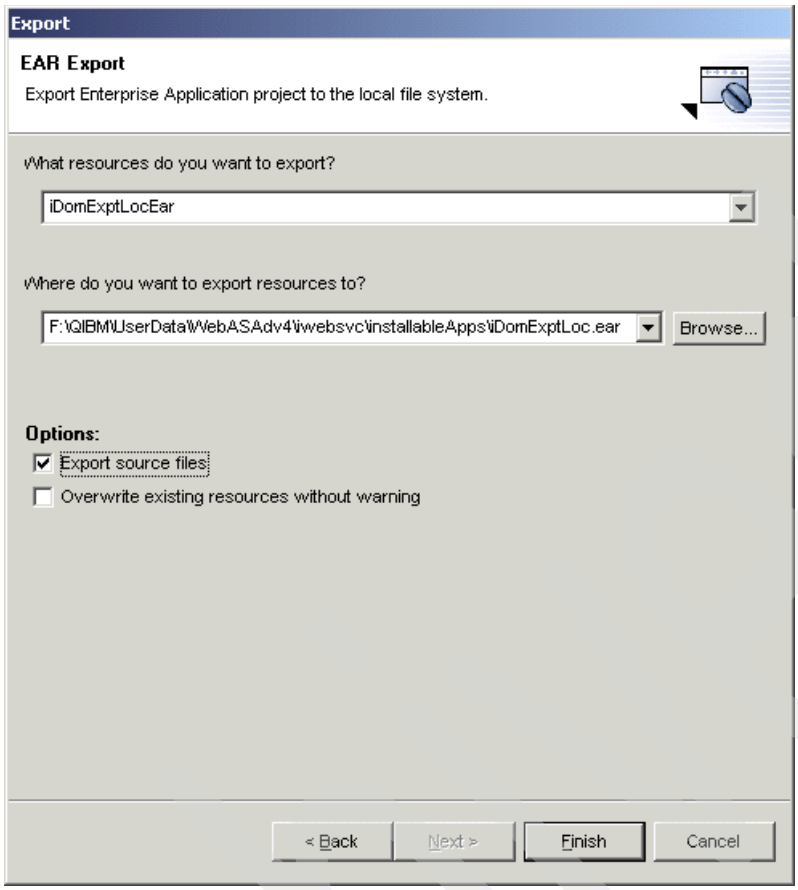


Figure 3-28 Select library path to export the EAR file in WDSc

Now to install this **iDomExptLoc.ear** file on WebSphere Application Server, we need to start the admin console for WebSphere Application Server. Go to a DOS command line prompt of your Windows machine and change directory to **WebSphere/AppServer/bin**. On this directory prompt type in the command **adminclient <node name> <port number>**, where **node name** points to the IP address of WebSphere Application Server machine and **port number** is the bootstrap port of WebSphere Application Server instance you are using. This will start the WebSphere Application Server Java based admin console.

Deploy the Web Service on WebSphere Application Server

As seen in Figure 3-29, browse to this directory to select the EAR file for deployment:

QIBM/UserData/WebAsAdv4/iwebsvc/installableApps/iDomExptLoc.ear

Enter the Application name as **ExptLocApp**.

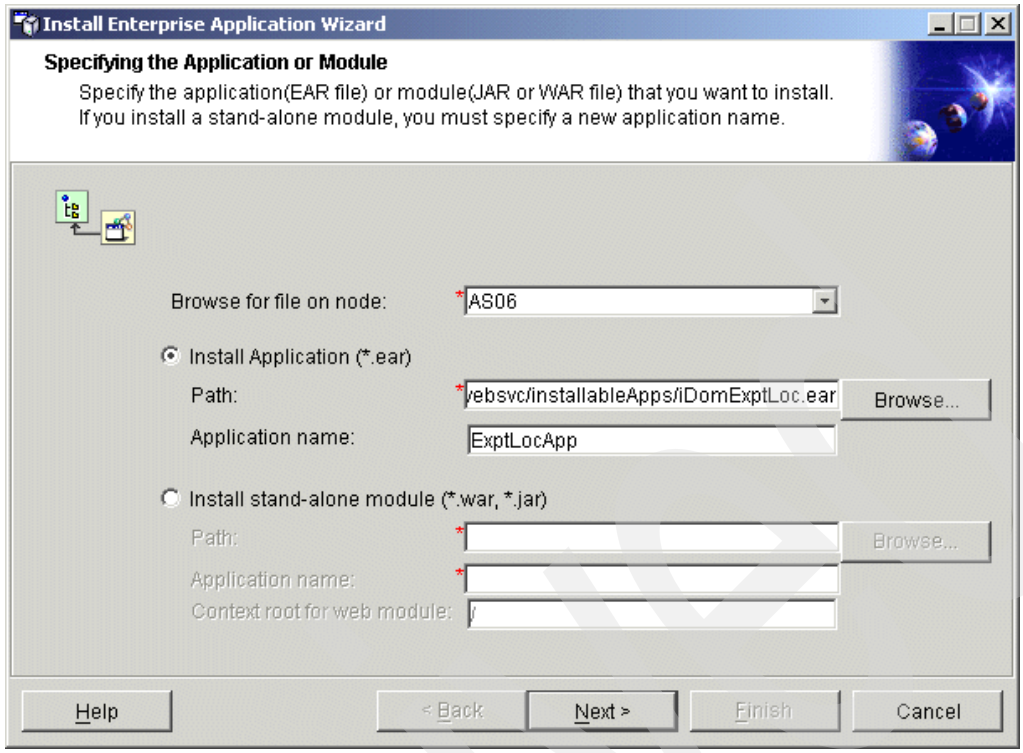


Figure 3-29 Specify path of EAR file on Install of Enterprise Application on WAS

After taking defaults on other windows, make sure that install directory path is correct on the final window as shown in Figure 3-30 and click **Finish**.

Figure 3-30 Install Enterprise Application Finish window on WebSphere Application Server

After deploying Expertise Locator Enterprise Application, place Apache soap.jar and NCSOW.jar files in the directory path **/QIBM/UserData/WebAsAdv4/iwebsvc/lib/ext**.

Important: Make sure that the WebSphere Application Server instance on which the Expertise Locator Web Service is deployed is stopped and restarted. Restarting the WebSphere Application Server instance makes the application specific jar files like NCSOW.jar available in classpath.

Start the StockQuoteApp application by right-clicking **ExptLocApp**. Press **Start**.

Test the WebSphere Application Server deployment of Web Service

Point your browser to:

```
http://idomws1:5500/iDomExptLoc/admin/index.html
```

As shown in Figure 3-31, you will see active services by clicking on **List all services**. In our example the Web Service is identified by URN (Uniform Resource Namespace): **ExpertiseLocatorService**.

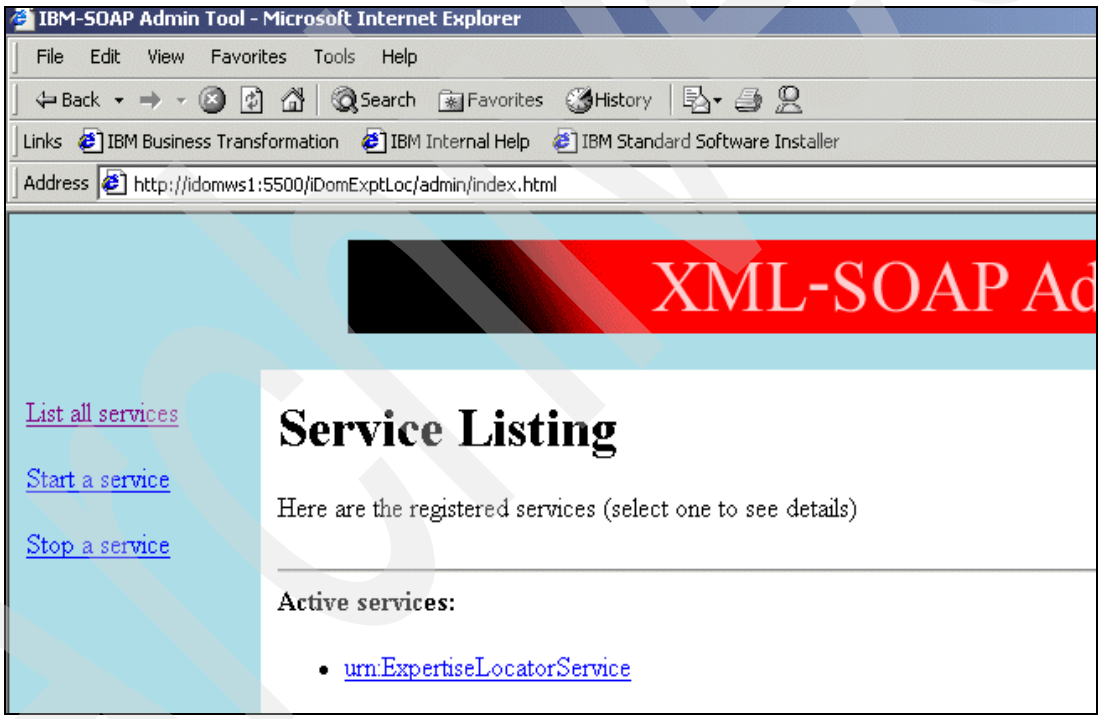


Figure 3-31 Verify the Web Service is running using XML-SOAP Admin client on Web Browser

Point your browser to this URL to run the sample test client:

```
http://idomws1:5500/iDomExptLoc/sample/ExpertiseLocator_ServiceService/TestClient.jsp
```

Click **getExpertise** method and enter **LotusScript** as taxonomy term. If you get a response like Figure 3-32, then *Congratulations!* You have successfully developed and tested your first LWSEK Domino iSeries Web Service application on WebSphere Application Server.

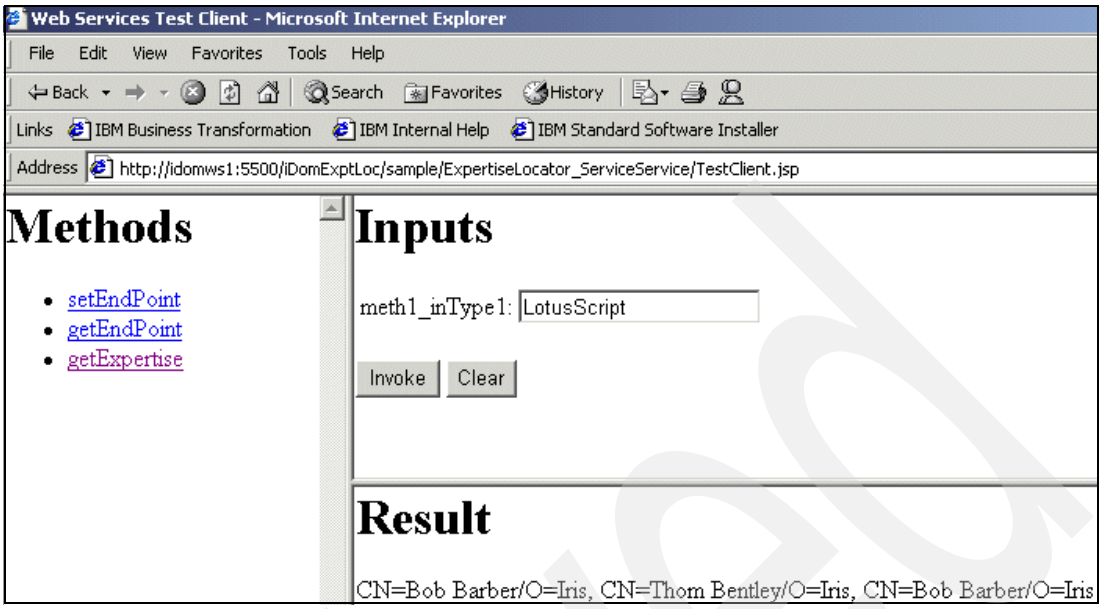


Figure 3-32 Test the Web Service using TestClient.jsp on Web browser

3.3.2 Tips on troubleshooting

Here some tips on troubleshooting or prevention measures to save you from problems.

Serialization concept for Apache SOAP

In a Java program, when you pass an object as parameter, the parameter is passed by reference. With SOAP, all the parameters are copied and passed with the SOAP call to the server. If the parameters are modified on the server side, the modifications are never returned to the client. Method return values are passed back from the server to the client. The method parameters and return values must be packaged (or marshaled) between client and server, because client and server are both running different Java Virtual Machines.

Serialization is the encoding of objects and objects defined in within them, into a stream of bytes. To marshall parameters and return values for specialized classes (that is, other than Java primitive data types and Java objects that implement serializable interface), SOAP implementation must know how to serialize the classes. Once a serializer class is defined in the configuration file DDS.XML, the SOAP envelope can generate XML representation. For example, in our Expertise Locator Web Service, the Apache SOAP generated XML representation of response object “Expert.class” uses the encoding style of “ExpertSerializer.class”.

Miscellaneous tips

This section includes some various tips for troubleshooting.

- For SOAP exceptions like “No Serializer/deserializer found to serializer a ‘XYZ’ using encoding style ‘FOO’, make sure that customized XML to Java mappings are set correctly for a Web Service in WDSc.
- For Serializer exceptions also make sure that proxy.soap.KKKKProxy.java file has SOAP Mapping Registry set to serializer deserializer classes.
- To create a Notes session from a Java class deployed on WebSphere Application Server, make sure you use NotesFactory and other classes from NCSOW.jar.

- For Domino server to provide an external Notes session to a Java program, DIIOP server task must be running on port 63148 of Domino server.
- The server document security setting for Run Restricted Java/COM agents should be set to Administrators and not left blank. See Figure 3-33.

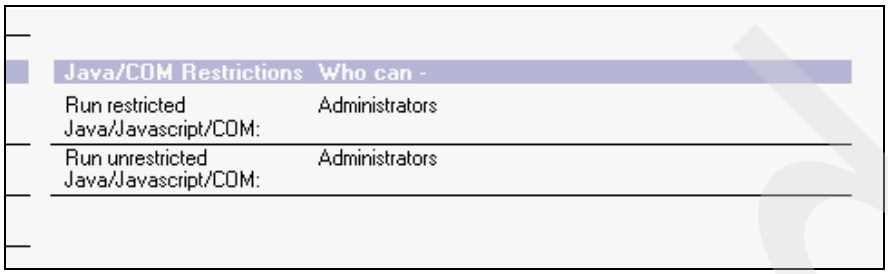


Figure 3-33 Setup Java/Com restrictions on server document

- If you get a class not found exception make sure that Apache soap.jar and NCSOW.jar files are properly installed on the Web Service application lib/ext directory.

3.4 Analysis of LWSEK

LWSEK stands for Lotus Web Service Enablement Kit. This kit is a blueprint including samples, demo and source code to enhance Domino applications with a Web Services framework. It is to provide some guidelines to bring out Domino applications to the Internet as a Web Service. LWSEK can be downloaded from this Web site:

`http://www.alphaworks.ibm.com`

3.4.1 What you get from it

This section describes what you get from LWSEK when you download it. Figure 3-34 shows the folder view of the kit package.

Archived

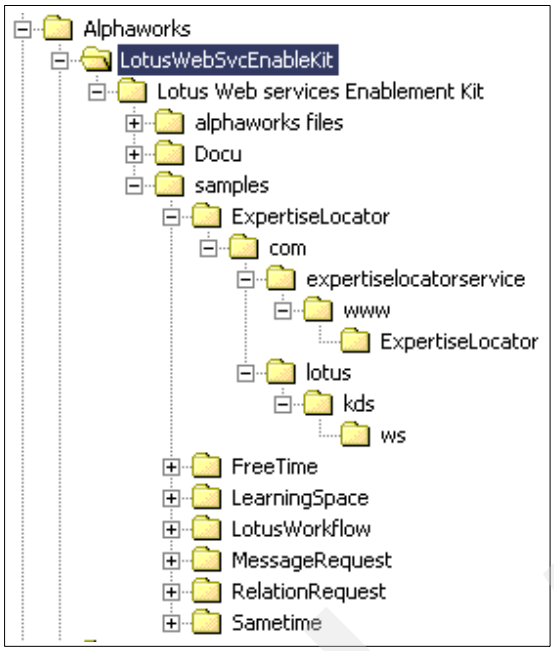


Figure 3-34 Directory view of LWSEK contents

LWSEK contains classes

These are Java class files with source code, access Domino objects and provide the functionality of various applications:

- Locate an Expert from “people.nsf” database.
- Search for Free Time of the Expert.
- Send a Message to the Expert.
- Initiate WorkFlow to request a service from expert, etc.

LWSEK contains databases

The sample Domino database “people.nsf” contains a searchable list of experts on Knowledge Discovery Server.

LWSEK contains consumer clients and WSDL files

Since LWSEK uses Web Services Tool Kit (WSTK) for Web Service enablement, it comes with all the required WSDL, configuration files and proxy consumer client to deploy the Web Service.

3.4.2 What we did with LWSEK

During this redbook project, we have:

- Analyzed LWSEK.
- Tested a couple of applications of its five applications.
- Deployed and ran it with the framework of WDSc as development/deployment tools and WebSphere Application Server as Web application server.

Archived

LWSEK uses WSTK, but we don't

This kit assumes use of WSTK to enable Domino applications for Web Service enablement. WSTK is used to generate WSDL files, Java source and class files for serializer objects and other deployment descriptors.

We used WDSc for iSeries

In this book we use WDSc for iSeries to generate all the required WSDL files and deployment descriptor files. We pick up only the very minimum required Java classes from LWSEK for core functionality. These Java classes are imported to WDSc. In WDSc the Java classes are enabled for the Web Service and deployed on WebSphere Application Server on iSeries. The Domino server's accesses to the databases is also running on iSeries.

3.4.3 Applications we tested

Here are descriptions on two applications we chose to test.

ExpertiseLocator Service

Expertise Locator Service is the Web Service that accesses Notes database people.nsf to find experts on a particular subject. People.nsf contains the personal profiles of employees, including their affinities to particular taxonomy terms. These taxonomy terms are used as search arguments by the Web Service requester client program. A Java class provides backend connectivity to the people.nsf on the Domino server and returns a list of experts found for a particular taxonomy term.

MessageRequest Service

Message Request Service is the Web Service that utilizes basic messaging in Domino. In this application, the backend Java class connects to Domino server as administrator. The administrator account mailbox on Domino is used to send messages. The request object as input are: SendTo, CopyTo, Subject, Body (only text format) and Principal. The response object as Output are errorcode and errorMessage.

Archive

Archived

4

Consuming Web Services from a Domino application

Now that we have described the basics in developing Web Services, how are you going to consume them? In this chapter we detail some of the different techniques that you may use in consuming Web Services within your Domino applications. In other words, we discuss the various methods of transforming an existing Domino application into a Web Service consumer.

There are two main parts in this chapter:

- Part 1: Concepts learning
  - We demonstrate what Web Service consumption is and how we can transform an existing Domino application into a Web Service requester application. We use SOAPConnect, which is a kit available from Lotus Developers Domain, as a learning guide. You will not be required to write a single line of the code since everything is packaged in the kit.
  - These sections are covered in Concepts learning:
    - Section 4.1, “Web Service consumption tutorial” on page 98
    - Section 4.2, “SOAPConnect TempfinderSample” on page 101
    - Section 4.3, “Consuming StockQuoteService using SOAPConnect” on page 107
- Part 2: Developing a Web Service consumer
  - With the knowledge you gained from Part 1, now it’s your turn to write a new Domino Web Service requester application, or transform your existing Domino application into a Web Service requester application. We walk you through this process, step-by-step.

© Copyright IBM Corp. 2002. All rights reserved.

4.1 Web Service consumption tutorial

In today's tough and demanding market, to be on top of the game means to be integrated as much as possible. Many times, the success of your sale depends on how seamless the transaction appears to your customer or how integrated you are with everyone who is involved. Integration is the name of the game when it comes to e-business. Now through the power of a Web Service this all becomes possible.

Wouldn't it be nice to be able to integrate your existing Domino application with your suppliers. With the consumption of Web Services, you can add a whole new dimension of power and functionality to your new or existing Domino applications by integrating them with others. Although the possibilities are endless, these the techniques can be quite rudimentary.

There are two sides of the Web Service: the Web Service provider and the consumer. In the previous chapter, we discussed how to convert your existing Domino applications into Web Service providing applications. In this chapter, we discuss how to do the opposite: Transform the Domino application into a Web Service consumer.

4.1.1 Web Service consumer introduction

As discussed in Chapter 2, "Writing my first Web Service application" on page 9, the information passed between a Web Service provider and consumer is done through SOAP messages. In our examples, the SOAP messages are generated through Java. Therefore, whatever program languages you use around this SOAP layer, we need to have a Java layer when it comes to converting the requesting information into SOAP messages.

Since our applications are based in Domino, one would likely use LotusScript all the time. For consuming Web Services, we now need to introduce Java into our application as mentioned above. Let's now discuss the strengths and weaknesses of LotusScript and Java in Notes.

LotusScript is a BASIC programming language that is embedded within Domino. LotusScript is a powerful object-oriented language that gives the developers access to two powerful Domino Object Model classes: Front-End Classes and Back-End Classes. When using the supplied Front-End Classes, the developer has the capability to add appealing and powerful user interfaces quickly and efficiently to their application. When using the Back-End Classes, LotusScript delivers a wide range of functionality for server side database programming. LotusScript is very simple and easy to learn, and this is what makes it the language of choice to so many developers.

Currently in Notes/Domino R5, LotusScript is unable to utilize the various Java platform software toolkits commonly distributed on the Internet. Java in Domino allows the programmer to use the same Back-End objects available to LotusScript, core Java language classes and other various powerful software libraries. When using these non-Domino classes in combination with the core Domino objects, our possibilities are virtually endless to what we can do within our application.

Domino Web Service consumer runtime flow

When trying to understand the way a Web Service consumer works, it is best to describe this with the use of a diagram. Figure 4-1 illustrates the whole runtime flow of Domino Web Service consumer.

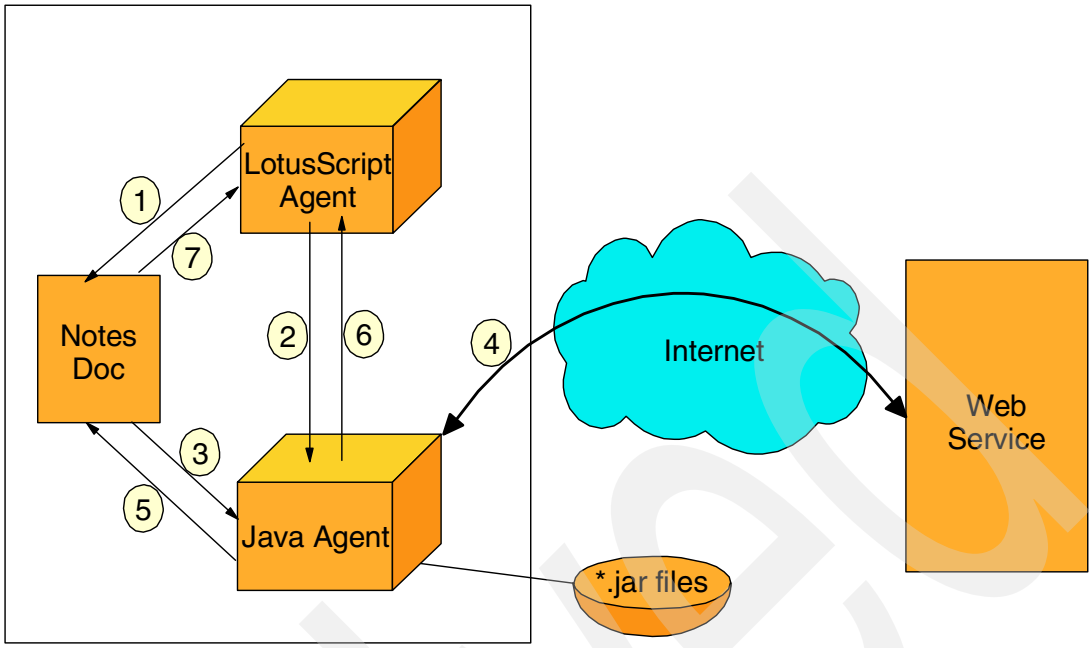


Figure 4-1 Domino Web Service consumer runtime flow diagram

1. The LotusScript agent is executed and all of the required parameters (endpoint, namespace, input parameters, and so on) are stored to a NotesDocument, retaining the document ID.
2. The LotusScript agent then calls the Java agent and passes the document ID for it to find its required parameters.
3. The Java agent finds the document and reads off all of its item values.
4. The Java agent then creates a SOAP message, via soap.jar (which will be explained later), and sends a request to the Web Service's SOAP server, in other words, the Web Service provider, via the Internet. This request is accepted, processed and then a response is returned.
5. When the data is returned, the Java agent parses the data and saves the values to the same NotesDocument as in Step 1.
6. A completion notification is then sent to the original LotusScript agent (in Step 2) telling it the SOAP message request has been completed successfully and the return data is waiting on the NotesDocument.
7. Finally the LotusScript agent reads the returned data from the NotesDocument and is now available for further processing.

4.1.2 Basic requirements to be a Web Service consumer

When consuming Web Services, we must first create a SOAP client. This client is used to form a SOAP wrapper around our Domino request. The client is also used to speak with the Web Service's SOAP server. This wrapper utilizes the functionality of special classes in the Java archive file soap.jar (and all of its dependents). Since LotusScript is unable to access Java objects, we need to import these .jar files into your current project and facilitate the SOAP client as a Notes Java agent. Throughout the remainder of this chapter, we will explore several examples and detail the functionality of each.

4.1.3 What is SOAPConnect?

SOAPConnect, sometimes spelled like SoapConnect, is a package developed by Lotus developers that will enable and allow simple LotusScript applications running on Domino, to consume Web Services. It is a partial implementation of the SOAP V1.1 standard for the LotusScript language. SOAPConnect provides us with new LotusScript classes used in creating SOAP messages, invoking SOAP services, and parsing the returned data for your application's use.

You may use SOAPConnect "out of the box" for consuming the Web Service that returns any of the Java primitive data types, such as int, float, or predefined Java runtime object classes, such as java.lang.String. In some cases, Web Services may use a custom serializer class to return the data packaged as a specialized class. SOAPConnect must be modified to allow for the consumption of this service. We must use their serializer/deserializer class on the return data in order for our application to use it. We will discuss this in more detail later in this chapter.

The SOAPConnect package can be downloaded from the Sandbox located at Lotus Developer Domain Web site, formerly known as Notes.net:

http://www-10.lotus.com/ldd

You will need to click Search the Sandbox to find the package. For our search, we used 'SOAPConnect' as the search keyword.

SOAPConnect components

The SOAPConnect is packaged as a zip file with several files included. Below is a list of the files included along with a brief description of their function.

- SoapSamples.nsf — The "front-end" of the application. This is where all of the Web Service requests are initiated and returned.
- SoapRunner.nsf — This is where all of the Java is stored and executed to perform the required SOAP message call.
- Help_SoapConnect.nsf — A very comprehensive resource for help. This database contains the definitions for all of the included LotusScript classes and debugging tips for common known problems.
- SoapConnect.lss — The LotusScript class file of SoapConnect. It contains SOAPClient, SOAPParameter, SOAPHTTPConnection and SOAPFault classes.
- Required .jar files — activation.jar, mail.jar, soap.jar, xerces.jar, xml4j.jar
- readme.htm — This readme file gives detailed instructions on where to install all of the components required by SoapConnect.

How we can use SOAPConnect

We will start by using SOAPConnect as our introductory example. You can download SOAPConnect and install it onto your Notes client or Domino server. SOAPConnect includes sample agents that allow you to start consuming your own or other's Web Services without writing a single line of code.

- We will examine the core functionality of this example to determine what is required to consume a Web Service of our own.
- Then we will customize the StockPortfolioSamples example, included in SOAPConnect, to point to the **StockQuoteService** Web Service we created in Chapter 2, "Writing my first Web Service application" on page 9.

- Finally we will write our own Web Service consumer from scratch. We will do this first by using a supplied *proxy* client and secondly by creating our own custom proxy client.

SOAPConnect built-in examples

SOAPConnect contains two very simple examples of how to consume Web Services through a Domino application to help you get started. The first example is TempfinderSample. This sample allows the user to enter in a zipcode of where they want to find the temperature. SOAPConnect then generates and sends a SOAP message request to a WebService for this information. The temperature is finally returned to a field on the form.

The second example included in SOAPConnect is StockPortfolioSample, which is very similar. This example also requires the user to input a stock ticker symbol onto a form. The SOAPConnect then also generates and sends a SOAP message request to a Web Service and the stock value is finally returned to a field on the form. This example then goes a step further by having an agent that loops through all of the “StockPortfolio” documents and updating their value periodically.

In the following sections, we will further investigate these two built-in examples.

4.2 SOAPConnect TempfinderSample

SOAPConnect illustrates to us the basics of what it takes to consume Web Services from within Domino applications. Here we will go into detail and explain how TempfinderSample works within SoapSamples.nsf.

4.2.1 Using TempfinderSample

Once you have completed the installation of SOAPConnect you will see we have two .nsf files to work with. SoapSamples.nsf contains the user interface and SoapRunner.nsf contains the back-end Java agent. This back-end Java agent class is also defined as the Web Service proxy code. It is where the actual SOAP message request is initiated from to connect to the Web Service.

To find the temperature of a specific location, you should open SoapSamples.nsf database and select **Create -> TempfinderSample -> Tempfinder**. This will bring up the form needed to consume this Web Service.

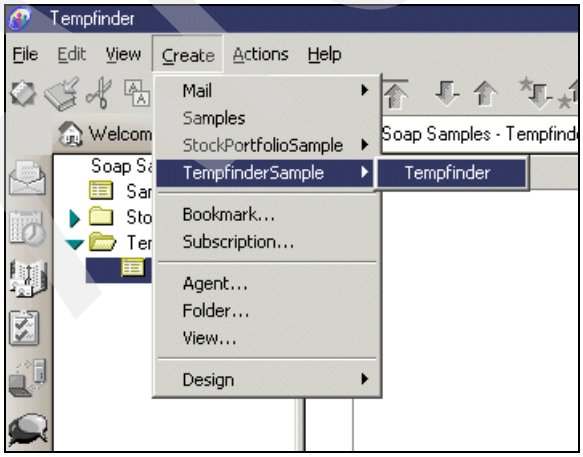


Figure 4-2 Calling Tempfinder from SoapSamples.nsf

Once the Tempfinder form is open, you see a blank field where you will want to enter the location's zipcode, as shown in Figure 4-3. The Web Service will return the temperature of this location. In my case I have typed in 55901 for Rochester, MN. Also on this form, you will see a button labeled "Get Temperature". By pressing this button, the agent will send a SOAP request to the Web Service with an input parameter of 55901.

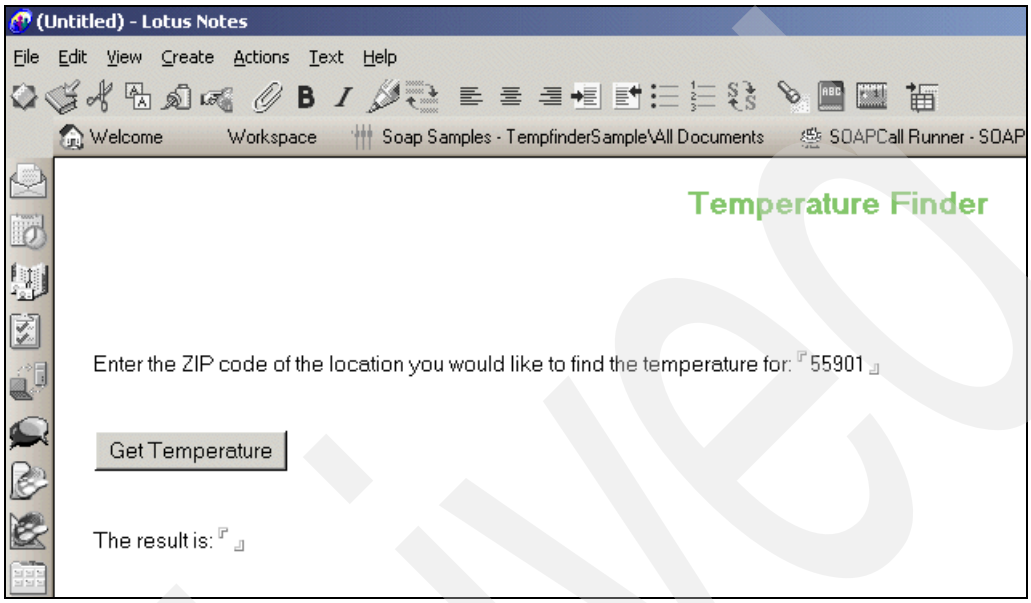


Figure 4-3 Tempfinder input parameter form

The provider's SOAP server accepts this request and parses through the XML SOAP message to see how it needs to be processed. The SOAP server routes the request to the corresponding Web Service method (in this case, getTemp method). The Web Service then processes and packages the response into a SOAP message and returns it to the consumer. From here the SOAP client extracts the data and it is then handed off to the original agent.

4.2.2 Understanding TempfinderSample

This section examines the technicalities of how TempfinderSample works. We look into the code behind this consumer example and emphasize what is required. You can refer to Figure 4-4 for a visual reference of the following information.

Archive

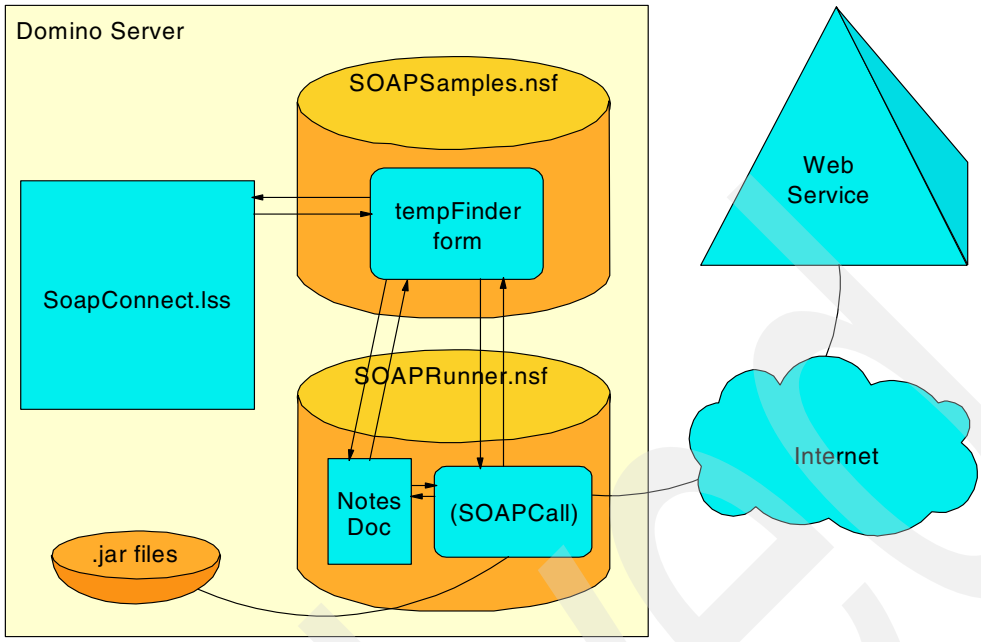


Figure 4-4 Tempfinder Web Service consumer technical flow diagram

You can see from the first line of the source code in Figure 4-5, the button uses SoapConnect's core functionality code located in SoapConnect.lss. This is included in the agent by adding the line %INCLUDE "SOAPConnect".

The first thing to point out is the variable called *EndPoint.* The *EndPoint* should be set to the uniform resource locator (URL) of the Web Service's SOAP rpcrouter. The SOAP rpcrouter is the Web Service's transport listener servlet. This servlet will listen for and process any service requests sent to the Web Service. As we will see later in this chapter, you can set this *EndPoint* to the URL `http://yourdomain:port/YourWebProject/servlet/rpcrouter`, the SOAP rpcrouter in your Web Service.

Next you will see that a LotusScript SoapClient object is declared as *TempService* by passing the *EndPoint* to its constructor. The script then calls the object's *invoke* method passing the *URI* or *NameSpace*, *MethodName* and the arguments. In this example, the *NameSpace* is urn:xmethods-Temperature. A NameSpace is defined as XML identifier to the Web Service application components. The *MethodName* is the method on the Web Services side that the consumer will be utilizing.

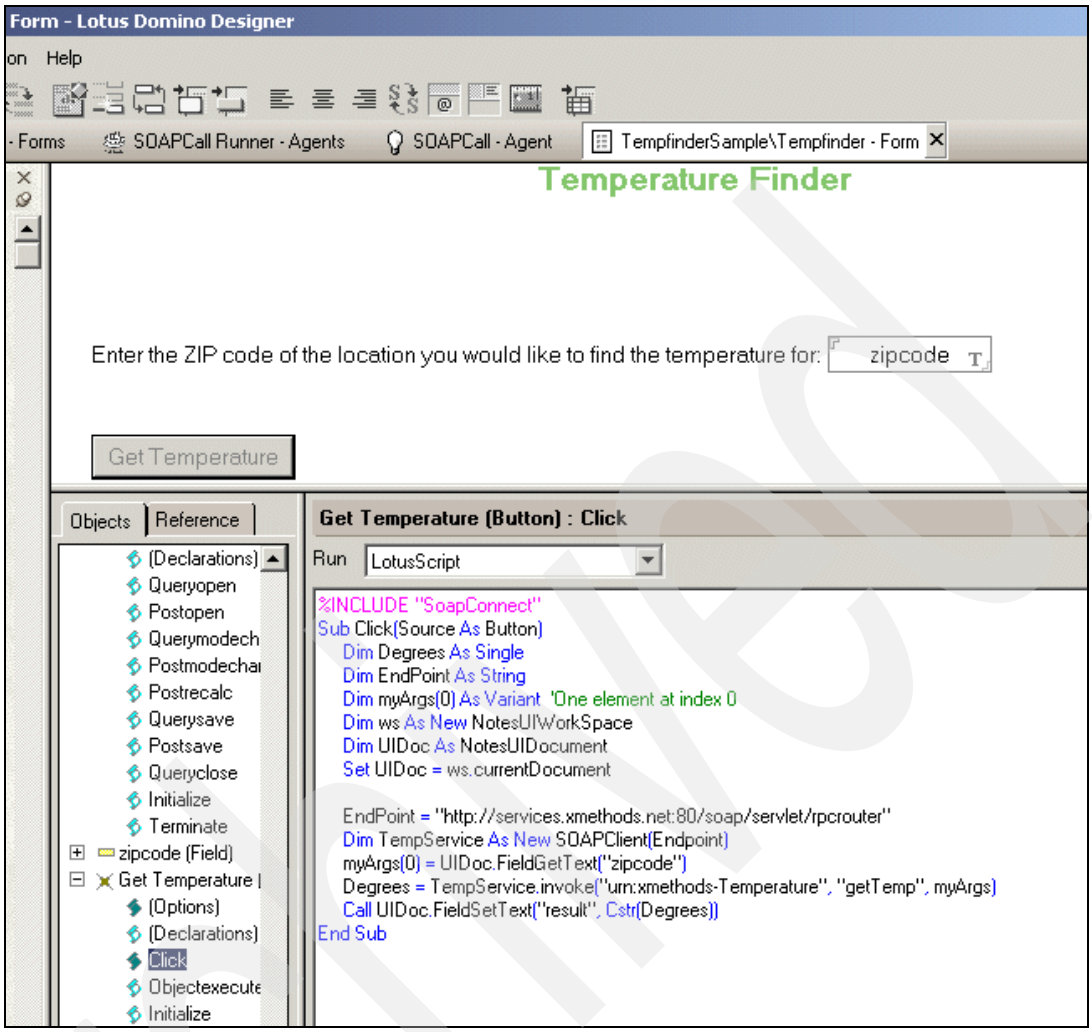


Figure 4-5 Source code behind Get Temperature button

The main purpose of the SoapClient *invoke* method is to package up the arguments into three separate arrays consisting of *ArgNames, ArgTypes* and *ArgValues*. Along with these, it saves the values of *EndPoint, NameSpace* and *MethodName* to a NotesDocument in SoapRunner.nsf. The method retains the document ID as a local variable for later use. Finally, *invoke* finds the (SOAPCall) Java agent in SoapRunner.nsf and passes it the document ID mentioned above.

This technique of utilizing a NotesDocument is required due to the limitations in Domino when agents call agents. Their ability to pass parameters between each other is quite limited. The only parameter that an agent is able to accept as an argument is a universal document ID (UNID).

Note: (SOAPCall) agent acts as the proxy client for our consumer in accessing the Web Service. This agent is the only place in the whole application that uses SOAP classes and methods. Having this portion of the application supplied to us decreases the amount of programming skill required in consuming Web Services.

Once the (SOAPCall) agent is executed, it first gets a handle to the NotesDocument by its UNID. It parses through the three parameter arrays described above, *ArgNames, ArgTypes*

and *ArgValues*, element by element, and packages each as a Java Parameter, storing them into a single array called parms, as shown in Figure 4-6.

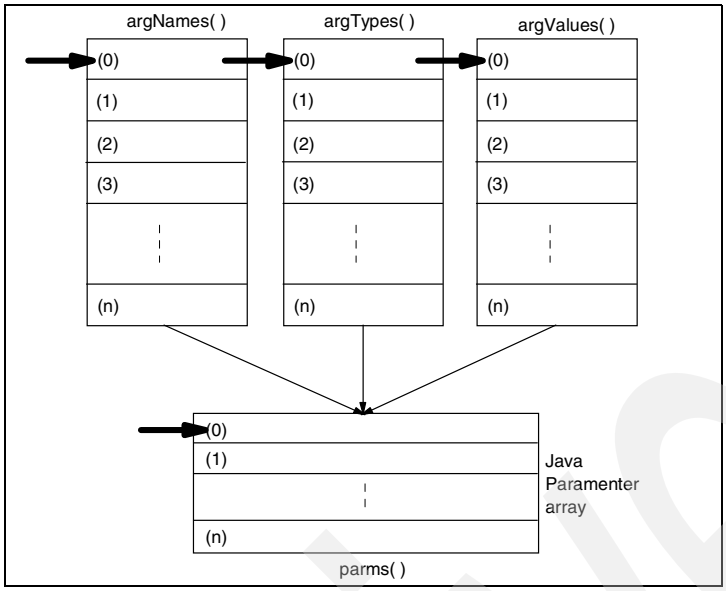


Figure 4-6 Condensing the three parameter arrays into a single Java Parameter array

In Example 4-1, you can see a section of code from the Java agent (SOAPCall). This is the code containing all of the essential SOAP programming needed in this consumer. First an object of the class *Call* is instantiated from soap.jar. Next, all of the items are taken from the NotesDocument and are set as shown in the sample code.

Example 4-1 Sample code setting required properties of the object Call from soap.jar

```
    ...
Call call = new Call();
    ...
call.setParams(params);
call.setTargetObjectURI(doc.getItemValueString("ServiceNamespace"));
call.setMethodName(doc.getItemValueString("MethodName"));
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
URL url = new URL(doc.getItemValueString("ServiceEndpoint"));
Response resp = call.invoke(url, "");
    ...
```

Once the SOAP request message is created, it is finally sent to the SOAP server by calling the *invoke* method. If there are no errors on the server-side, the server will return the requested information and set it to the object *resp* of type Response. Finally, the rest of the agent (SOAPCall) parses the returned data and sets the values to the corresponding fields on the same NotesDocument as above and saves it. Upon completion, (SOAPCall) notifies the SoapClient invoke method of its completion.

The LotusScript SoapClient *invoke* method then re-obtains a handle to the NotesDocument and gets the values and types of the returned data. This data can now be used and processed as needed. If there were any errors encountered in the consumption process, the error message would be displayed at this time.

Archived

4.2.3 TempfinderSample runtime flow summary

This diagram looks very similar to Figure 4-1 on page 99, yet different. The main difference is we now use the LotusScript class library, SOAPConnect.lss. Figure 4-7 provides a step-by-step summary of how the example TempfinderSample executes.

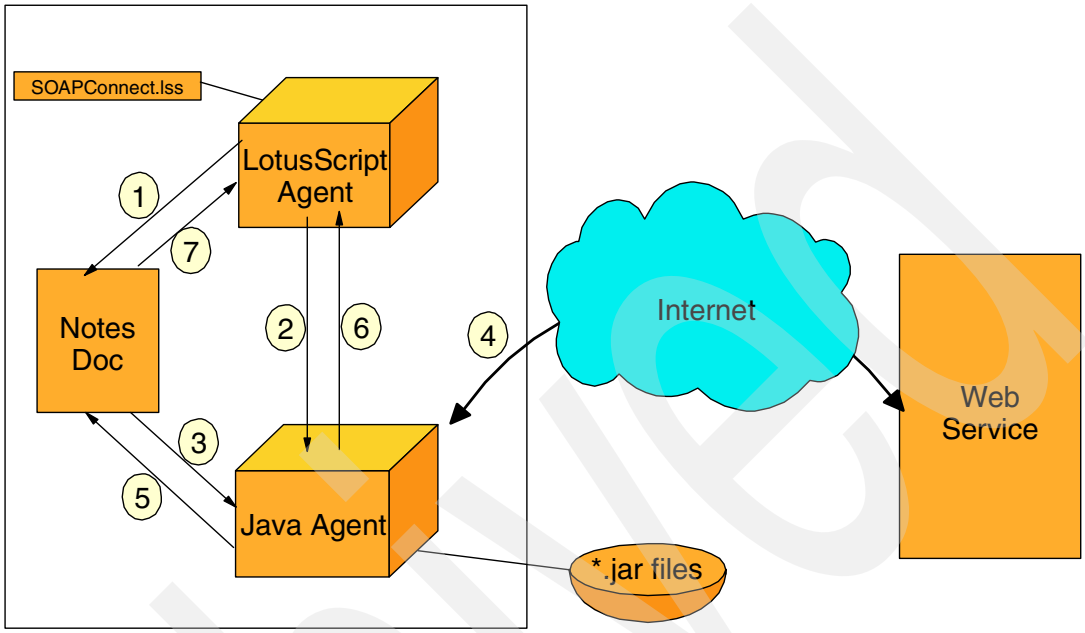


Figure 4-7 SOAPConnect summary flow diagram

1. The LotusScript button is pressed and creates an instance of the class SoapClient. The SoapClient stores all of the required parameters (endpoint, namespace, input parameters, and so on) to a NotesDocument, retaining the document ID.
2. The SoapClient then calls the Java agent and passes the document ID for it to find its required parameters.
3. The Java agent finds the document and reads off all of its item values.
4. The Java agent then creates a SOAP message, via soap.jar and its dependents, and sends a request to the Web Service's SOAP server, in our case, via the Internet. This request is accepted, processed and then returned.
5. When the data is returned, the Java agent parses the data and saves the values to the same NotesDocument as in Step 1.
6. A completion notification is then sent to the original LotusScript agent (in Step 2) telling it the SOAP message request has been completed successfully and the return data is waiting on the NotesDocument.
7. Finally the SoapClient reads the returned data from the NotesDocument and is now available for further processing.

Although this runtime flow is described according to the consumer TempfinderSample using SoapConnect, we can use the same idea in creating our own Web Service consuming Domino application.

Archived

4.3 Consuming StockQuoteService using SOAPConnect

In Chapter 2, “Writing my first Web Service application” on page 9, we described how you can create your own Web Service using WebSphere Development Studio client for iSeries (WDSc). This section describes how to customize the SOAPConnect detailed in 4.1.3, “What is SOAPConnect?” on page 100 to consume your StockQuoteService Web Service.

In this example we will illustrate the exact steps for changing the pre-existing code in StockPortfolioSample. It might be a good idea to back up the SoapSamples.nsf and SoapRunner.nsf database files, or to copy the design elements we will be changing and rename them to a unique name of your choice.

4.3.1 Customizing StockPortfolioSample

We start by opening the SoapSamples.nsf database. In Figure 4-8, you will notice by selecting the **Create -> StockPortfolioSample -> StockForm** menu, there is already an example to consume a stock quote Web Service. The included example points to an external Web Service that processes stock quote value requests. What we need to do is to change the required lines in this application to point this example to the StockQuoteService Web Service we created in Chapter 2, “Writing my first Web Service application” on page 9.

Important: As of the time of publication of the redbook, this example was required to use the **IBM soap.jar** included with the distribution of WebSphere Application Server 4.02 or WebSphere Development Studio client for iSeries 4.0.0

- WebSphere Application Server — /QIBM/ProdData/WebASADV4/lib/soap.jar
- WDSc — \WDSC\WSSD\plugins\com.ibm.etools.websphere.runtime\lib\soap.jar

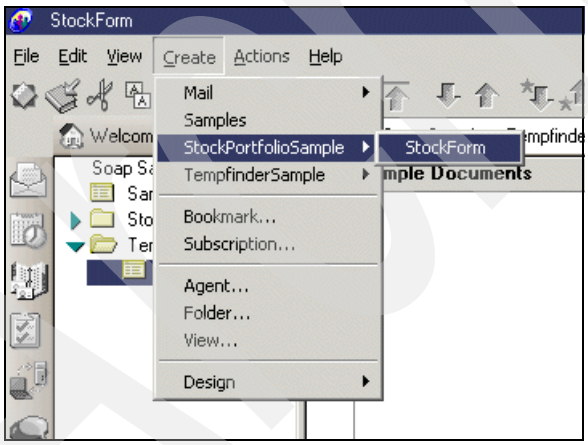


Figure 4-8 SoapConnect - StockPortfolioSample -> StockForm

When changing any example, it is always a good idea to create a backup. Here we will copy the existing form so we can customize it to our Web Service. You can achieve this by doing the following:

1. Open the SoapSamples.nsf in Domino Designer by selecting **View -> Design**. This will open the database in Domino Designer and place you in the Form elements area of the database shown in Figure 4-9.

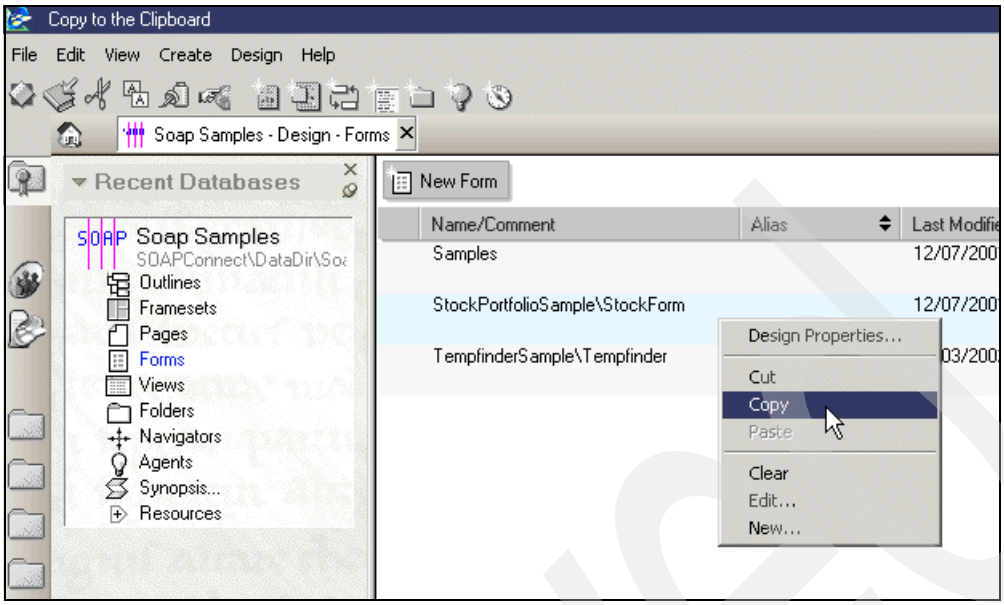


Figure 4-9 Form design elements in SoapSamples.nsf

2. On the right side of the client, shown in Figure 4-9, you will see the design element **StockPortfolioSample\StockForm**. We will want to copy this form and rename it with a unique name of your choice. You can do this by first selecting the form **StockPortfolioSample\StockfForm**. Right-click the form name and select **Copy**. Paste the form back into the database by right-clicking anywhere in the same region and selecting **Paste**. There will now be a new form named **Copy Of StockPortfolioSample\StockForm** as shown in Figure 4-10.

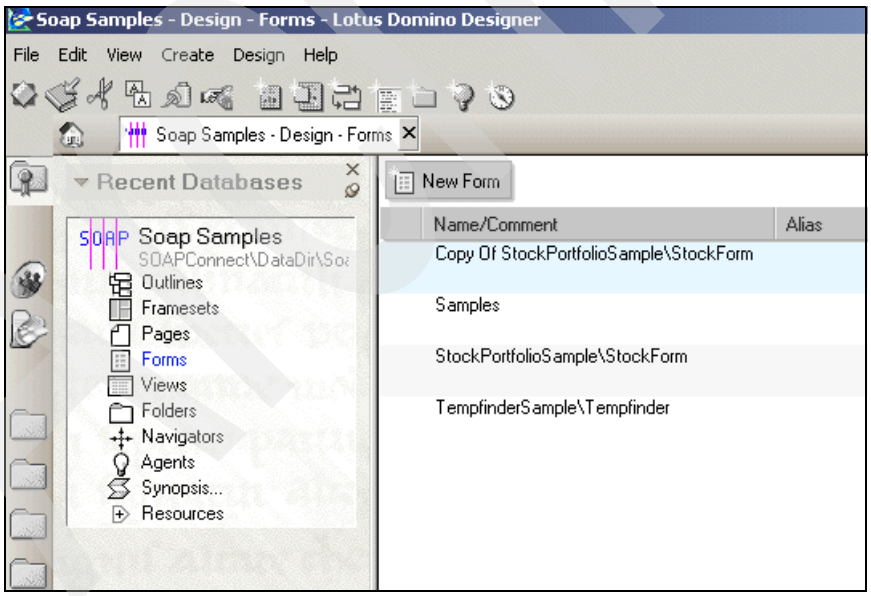


Figure 4-10 Form design elements in SoapSamples.nsf after copying StockForm

3. Open this new form and rename it in the Form properties dialog window, as illustrated in Figure 4-11, to a new unique name. In this example we renamed the form to **StockPortfolioSample\MyNewStockForm**.

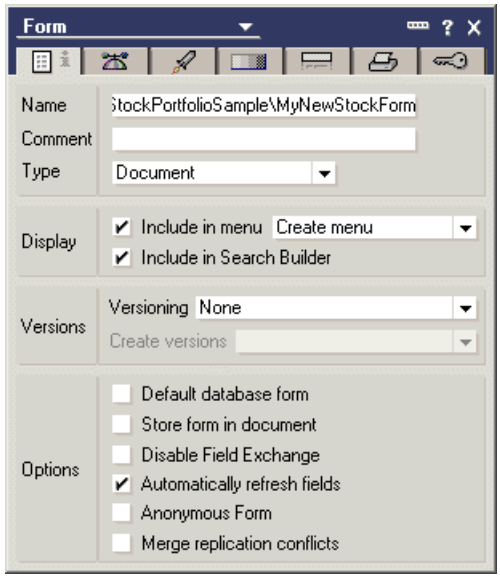


Figure 4-11 StockPortfolioSample\MyNewStockForm properties window

Tip: If you have created a function within an agent you feel that could be reused elsewhere, it is a good idea to copy this function into a new Script Library. Script Libraries are useful in Domino application programming because all of the functions it contains are available to all elements within the database.

If you want your functions or classes to be available to *all* applications on the Domino server, you should create a .lss file. The .lss file has the same functionality of a database's Script Library, yet is accessible server wide. Also when using .lss files, you can hide your code to the end user, when developing commercial applications.

This example uses a Script Library to hold a required function. We can edit this function by opening the Script Library design elements section of the database. This example uses a function that creates an instance of the class SoapClient. We will need to change the values in this function to match the *EndPoint*, *NameSpace* and *MethodName* values that are defined for our Web Service.

4. Open this by expanding the Resources section of the database design elements view, and by clicking on Script Libraries as shown in Figure 4-12.

Archived

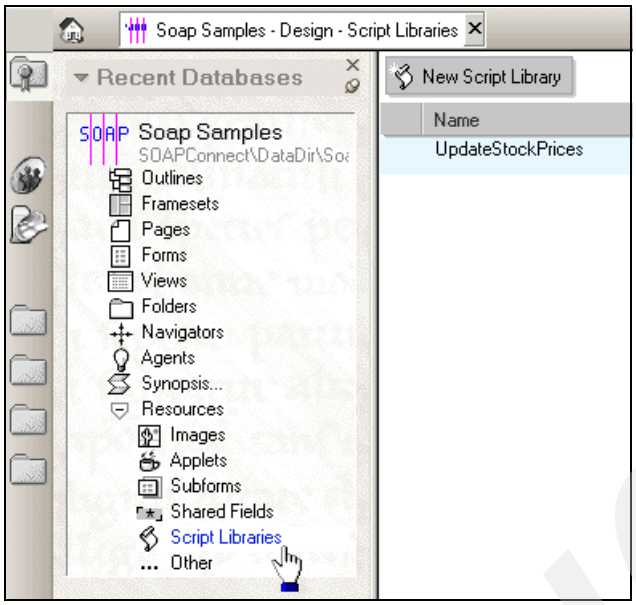


Figure 4-12 SoapSamples.nsf Script Library design elements

5. Figure 4-12 shows you the Script Library design elements in this database. You will see the Script Library "UpdateStockPrices". Copy and paste this library in the same fashion as described previously for the new form. You should see the new library appear in the design pane with the name of **copy of updateStockPrices**.
6. Open this new library and rename it to **"MyNewUpdateStockPrices"**. This is done by pressing **alt-enter** to display the Script Library properties window as shown in Figure 4-13.

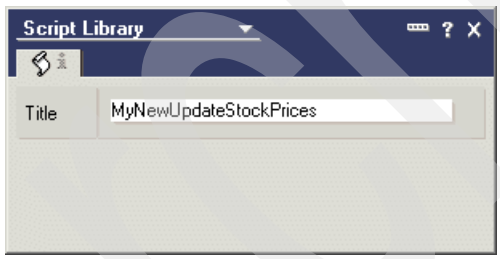


Figure 4-13 MyNewUpdateStockPrices Script Library properties window

7. Find and select the "updateStockWS" function within this library. In this function, we need to change a few lines to point to our own StockQuoteService Web Service. The line numbers correspond to the lines displayed in Example 4-2 on page 111.
   - Line 8 - `EndPoint = "http://idomws1:5500/WebProject/soap/servlet"`

     The URL of the SOAP rpcrouter, or request listener servlet, on your Web Service. If you have worked through the chapters earlier in the book, we set the root our StockQuoteService Web Service to WebProject.
   - Line 11 - `ServiceURI = "http://tempuri.org/StockQuoteService"`

     This is the XML NameSpace.
   - Line 14 - `Method = "getQuote"`

     This is the name of the Java method that actually does the work in the Web Service.
   - Line 19- `ServiceArgs(0) = symbol`

– Delete Line 21 - `ServiceArgs(1) = “USA”`

Example 4-2 updateStockWS in MyNewUpdateStockPrices script library after changes

```
1 Function updateStockWS(symbol As String) As Variant
2  ' Supply the following 3 values
3  Dim EndPoint As String
4  Dim ServiceURI As String
5  Dim Method As String
6
7  'Endpoint Server of Stock Quote
8  EndPoint = "http://idomws1:5500/WebProject/soap/servlet"
9
10 ' URI of the Stock Quote namespace
11 ServiceURI = "http://tempuri.org/StockQuoteService"
12
13 ' Stock Quote method
14 Method="getQuote"
15
16 ' Set up method parameters
17 ' This Web Service requires two parameters: stock symbol and
18 ' country name. It returns quote in that country's currency.
19 Dim ServiceArgs(0) As Variant
20 ServiceArgs(0) = symbol
21
22 Dim BNService As New SoapClient(EndPoint)
23 BNService.debug = True
24
25 updatedPrice = BNService.invoke(ServiceURI, Method, ServiceArgs)
26 updateStockWS=updatedPrice
End Function
```

After you have finished updating the function “updateStockWS”, it should appear as in Example 4-2.

8. Return back to the form, **MyNewStockForm**. Find and click the button on this form labeled “Retrieve Price”. In the pane below this form, you should notice the LotusScript code that runs behind that button will appear. In the “Options” section of the LotusScript code you will see the following:

   ```
   Use "UpdateStockPrices"
   ```

   You will want to change that line to point to the changes in our new script library:

   ```
   Use “MyNewUpdateStockPrices”
   ```

9. Finish this section by selecting **File -> Save**.

Note: You will notice that both Script Libraries contain a function called “updateStockWS”. This is alright since the button on our form will be pointing to the actual name of our *new* Script Library “MyNewUpdateStockPrices”.

In Figure 4-14, you can see how all of the scripts we have discussed above work together. We have the LotusScript behind the button, which uses a function from the Script Library MyNewUpdateStockPrices, which uses a class located in SOAPConnect.lss, which calls a Java agent (SOAPCall) located in SoapRunner.nsf. This final Java agent is the one that finally creates the SOAP message, and interacts with the SOAP server of the Web Service. This description may sound confusing, but as we have seen earlier in this section, the concept is quite simple.

Archived

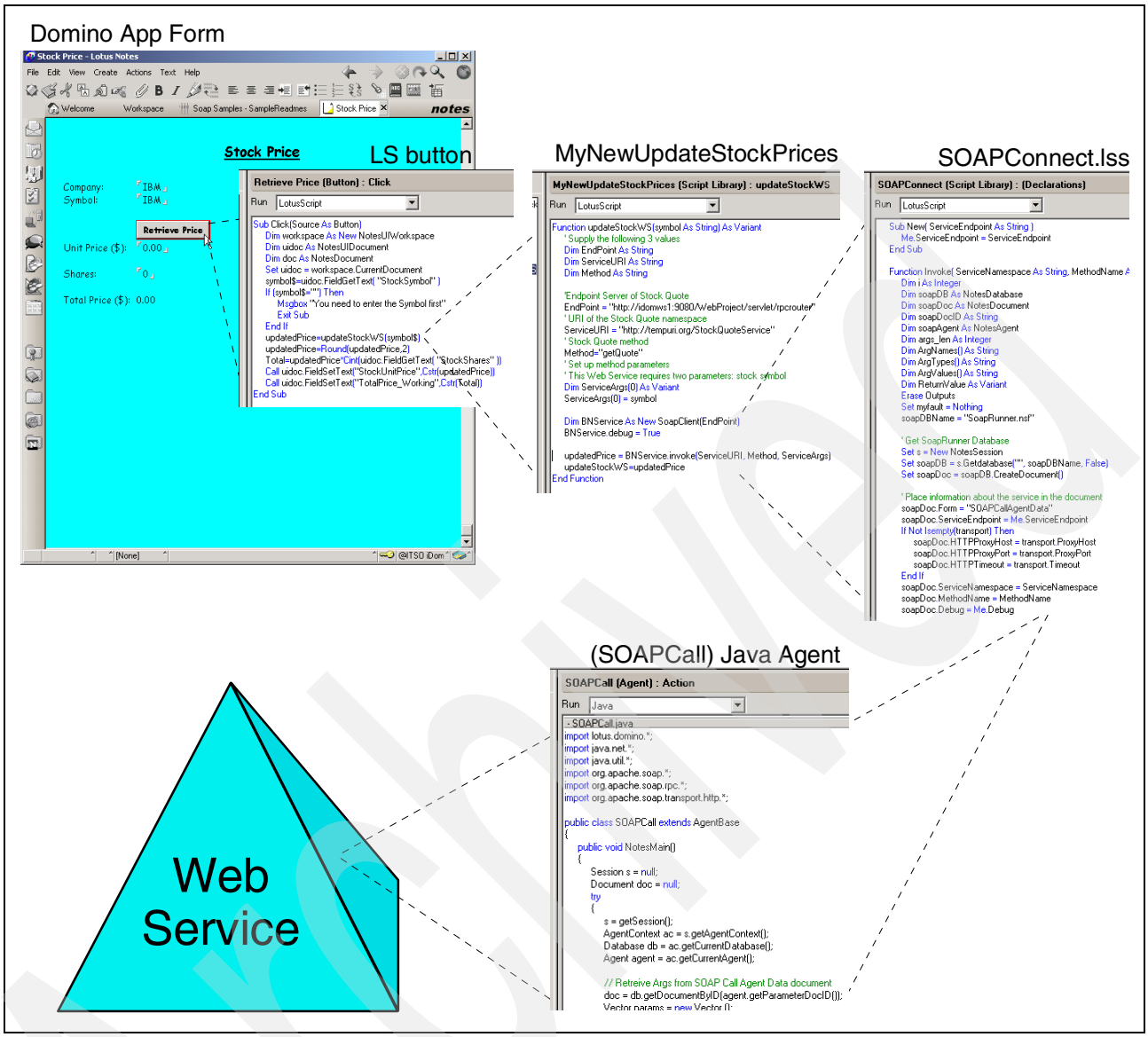


Figure 4-14 MyNewStockPortfolioSample script flow diagram

Congratulations! You have finished all the required steps to make the SoapConnect example, **StockPortfolioSample** consume your Web Service **StockQuoteService**. We will now switch gears and illustrate how to test your new Web Service consumer.

4.3.2 Testing your new consumer

Assure you have saved and closed all of the scripts and forms, and return to SoapSamples.nsf database. You can test your new consumer by selecting **Create -> StockPortfolioSample -> MyNewStockForm** as shown in Figure 4-15. This menu button will bring up the form as the user interface to our consumer.

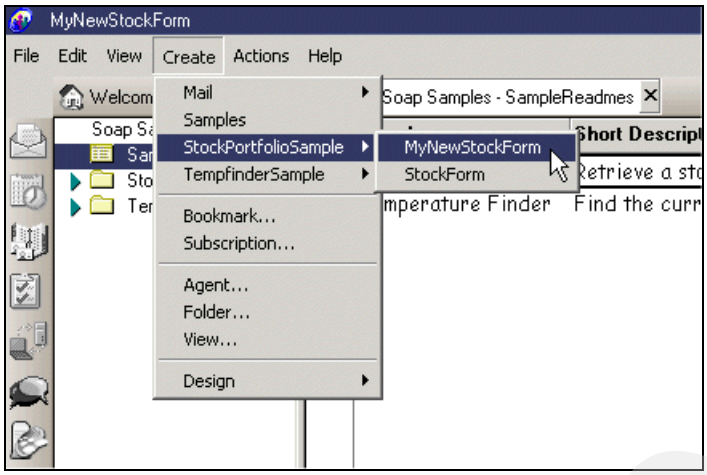


Figure 4-15 Testing your Web Service consumer

You can see on the form shown in Figure 4-16, the only required field is **symbol**. All of the other fields are for the organization of your portfolio. After you have typed in a stock symbol into the field, press the “Retrieve Price” button to activate your consumer. A few seconds later you will see a number value populated into the **Unit Price ($)** field. This number is the current value of the stock according to our Web Service provider.

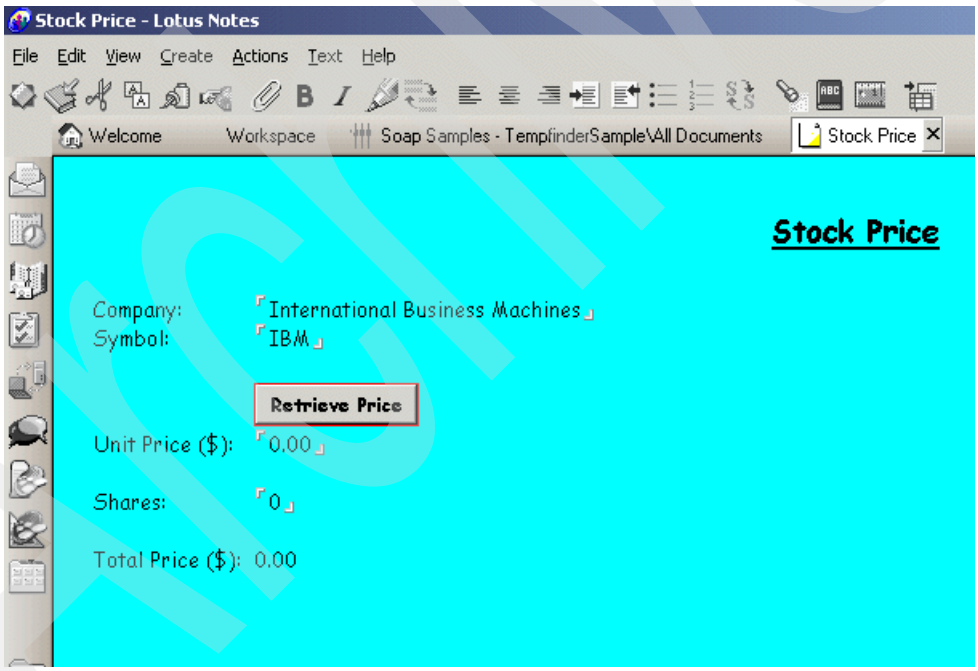


Figure 4-16 SOAPConnect form used to consume StockQuoteService Web Service

The remaining fields on this form do not pertain to the actual Web Service consumption, but are added features of this Domino application. You may type in the company name and the number of shares you have for when the form is saved. This creates a portfolio document for you with the current value for the given stocks. Also included in this application is an agent called **UpdateSelectedStockPrices**, that periodically updates the prices on all documents you have saved. This agent can be customized for use with minimal effort utilizing the knowledge you have gained in this section.

4.3.3 Debugging tips

As mentioned in the Help_SoapConnect.nsf database included with SoapConnect, we can use a special utility to view the actual SOAP messages sent across the “wire”. This utility is called *TCP Tunnel/Monitor* actually shows the XML that flows between the client and the server. The utility is included in the soap.jar that is distributed in the SOAPConnect package. For detailed documentation on this tool, refer to:

http://xml.apache.org/soap

Note: To use the TCP Tunnel/monitor utility, you must have a Java Development Kit (JDK) or Java Runtime Environment (JRE) installed on your local system.

TCP Tunnel/Monitor acts as a broker in the SOAP transaction between the client and the server. To set this up for your environment, you must first change the *EndPoint* in the LotusScript that invokes the SoapClient object. You need to set this *EndPoint* to a virtual URL, pointing to the TCP Tunnel/Monitor tool. Once the TCP Tunnel/Monitor tool receives the request, it then passes it along to the actual endpoint. Since this “tunnel” was established from the initial request, the response will travel back through the same tunnel, therefore capturing all the response data also.

As an example, the first *EndPoint* URL is the actual endpoint of the Web Service. The second *EndPoint* is the “virtual” URL of the Tunnel/monitor tool:

```
http://idomws1:5500/WebProject/servlet/rpcrouter
http://localhost:8082/WebProject/servlet/rpcrouter
```

We execute the TCP Tunnel/Monitor in the following way:

1. First open up a command prompt and change to the directory where you copied soap.jar in the installation of SOAPConnect. During the initial installation of SOAPConnect you should have also copied a file tunnel.bat to the same directory. In our case, both files are located in **c:\lotus\notes**.

   Tunnel.bat is a DOS batch file that requires the use of your installed JRE. If this fails, assure you have the path to your JDK/JRE in your system environment PATH setting. This is set in my case by typing the following in the command prompt window:

   ```
   set PATH=C:\Program Files\IBM\Java13\bin -or-
   set PATH=C:\WebSphere\AppServer\java\bin
   ```

   Note: Please note that by setting your PATH, this will only hold for the current “session” of your command prompt window. After closing or when using a different window, you must set the PATH again. You can set this permanently by first going to your “system properties” area on your Windows/32 system. Choose the Advanced tab, then select Environmental Variables. You will find “Path” under the System Variables area.

2. To execute the TCP Tunnel/monitor, type in the following command as shown in Figure 4-17.

   ```
   Usage: tunnel <listenport> <tunnelhost> <tunnelport>
   ```

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>cd lotus\notes

C:\lotus\Notes>tunnel 8082 idomws1 5500
```

Figure 4-17 Command prompt view when executing TCPTunnel/monitor

The actual call made in the tunnel.bat batch file is:

```
java org.apache.soap.util.net.TcpTunnelGui 8082 idomws1 5500
```

TCP Tunnel/Monitor has a nice user interface illustrated in Figure 4-18. Once specifying *EndPoint* in your LotusScript code to point to `http://localhost:8082/WebProject/servlet/rpcrouter`, the TCP Tunnel/Monitor will be listening for SOAP requests on the port 8082. Once a connection is established, it will capture the SOAP message and display it in the left pane of the interface. This message will then be directed to the same location on the specified <tunnelhost> and <tunnelport>.

When the server sends a response back, this SOAP message will also be captured and displayed in the right pane of the interface. These messages can be seen in Example 4-3 and Example 4-4.

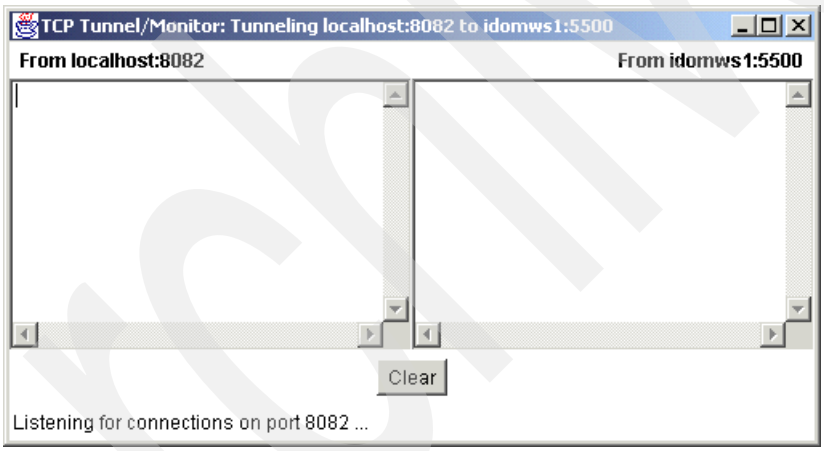


Figure 4-18 TCP Tunnel/Monitor user interface

You can see in Example 4-3, the message is sent in XML. The message is transformed into a SOAP message by the creation of the SOAP envelope. If you look closely at the message, you can see all of the information we supplied the consumer earlier. The XML NameSpace is defined as *ns1*, the method name getQuote is defined, and the symbol is defined as IBM of type *xds:string*. Using this tool can be helpful in debugging to assure that all of your necessary parameters are included within the SOAP envelope successfully.

Example 4-3 TCP Tunnel/Monitor — from localhost:8082

```
POST /WebProject/servlet/rpcrouter HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 469
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
   <SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Archived

```
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
      <ns1:getQuote xmlns:ns1="http://tempuri.org/StockQuoteService"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <symbol xsi:type="xsd:string">IBM</symbol>
      </ns1:getQuote>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

When looking at the response XML SOAP message in Example 4-4, the first you should notice is the <ns1:getQuoteResponce ....> tag. This shows you that the message coming back is officially a response of the message sent with the <ns1:getQuote ...> tag. This example shows us the return value is sent in the form of *<return xsi:type="xsd:float">81.3</return>*. If the data returned was a class, there would be multiple values listed in the return tags.

Example 4-4 TCP Tunnel/Monitor — from idomws1:5500

```
HTTP/1.1 200 OK
Server: WebSphere Application Server/4.0
Content-Type: text/xml; charset=utf-8
Set-Cookie: JSESSIONID=0000B104XNK44MKXYUDFDIK51OI:tng3sphr;Path=/
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Content-Length: 479
Content-Language: en
Connection: close

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getQuoteResponse xmlns:ns1="http://tempuri.org/StockQuoteService"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <return xsi:type="xsd:float">81.3</return>
      </ns1:getQuoteResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

4.4 Consuming Web Services with a supplied proxy client

When starting to write your first Web Service consumer from scratch, it is helpful to learn by examples. The examples shown previously in this chapter mainly concentrated on how you can make your coding efforts simple by using LotusScript. These examples were user friendly by having the proxy client Java agent transparent to the developer. The only component the user had to customize was the LotusScript interface. They were also needed to make sure all of the required values were passed to your proxy class.

Up until now we did not care about the proxy Java agent, but what if this agent was not supplied. This section will provide you with examples on what is required in this Java portion of the consumer to successfully create the SOAP message the server requires. We will take a backwards approach to describing this consumer starting with the proxy client Java class, then moving to the Notes Java agent that uses the proxy class and finally finishing up with the LotusScript agent that calls the Java agent.

4.4.1 What is a proxy client?

Once a Web Service is published on the Internet and listening for SOAP messages, we need to create a request SOAP message to send to it. Up until now, we have only mentioned throughout this redbook that we create the message by using the Java archive file soap.jar. Nothing has yet been said on *how* it is created. Even though the code required to generate such a message is basically minimal, it is not very intuitive.

What the proxy client gives us is a class that contains all of the function calls required to generate a SOAP message and to invoke the SOAP request. Now for you as the developer, all that is required for the consumption of the Web Service is to create an instance of this proxy client class, and call the consuming method (that is, getQuote) it contains.

Today, there are tools that will easily generate this proxy class for you. In the example we will be showing, we will use the StockQuoteServiceProxy class auto-generated by WebSphere Development Studio client for iSeries (WDSc). This class will be called from a basic Java agent in our consumer Domino application.

4.4.2 StockQuoteServiceProxy class

As mentioned above, the StockQuoteServiceProxy is auto-generated by WDSc. You can do this by selecting the **Generate Proxy** check box illustrated in Figure 2-20. The entire proxy client class Java code generate is shown in Example 4-5.

Example 4-5 StockQuoteServiceProxy.java

```
package proxy.soap;

import java.net.*;
import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.messaging.*;

public class StockQuoteServiceProxy
{
  private Call call = new Call();
  private URL url = null;
  private String stringURL = "http://idomws1:5500/WebProject/servlet/rpcrouter";
  private SOAPMappingRegistry smr = call.getSOAPMappingRegistry();

public StockQuoteServiceProxy()
  {
  }

public synchronized void setEndPoint(URL url)
  {
    this.url = url;
  }

public synchronized URL getEndPoint() throws MalformedURLException
  {
    return getURL();
  }
```

```
private URL getURL() throws MalformedURLException
  {
    if (url == null && stringURL != null && stringURL.length() > 0)
    {
      url = new URL(stringURL);
    }
    return url;
  }
public synchronized float getQuote(java.lang.String symbol) throws Exception
  {
    String targetObjectURI = "http://tempuri.org/StockQuoteService";
    String SOAPActionURI = "";


   if(getURL() == null)
    {
      throw new SOAPException(Constants.FAULT_CODE_CLIENT,
      "A URL must be specified via StockQuoteServiceProxy.setEndPoint(URL).");
    }
    call.setMethodName("getQuote");
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setTargetObjectURI(targetObjectURI);
    Vector params = new Vector();
    Parameter symbolParam =
       new Parameter("symbol", java.lang.String.class, symbol,
       Constants.NS_URI_SOAP_ENC);
    params.addElement(symbolParam);
    call.setParams(params);
    Response resp = call.invoke(getURL(), SOAPActionURI);

    //Check the response.
    if (resp.generatedFault())
    {
      Fault fault = resp.getFault();
      call.setFullTargetObjectURI(targetObjectURI);
      throw new SOAPException(fault.getFaultCode(), fault.getFaultString());
    }
    else
    {
      Parameter refValue = resp.getReturnValue();
      return ((java.lang.Float)refValue.getValue()).floatValue();
    }
  }
}
```

You will notice the main component of this class is the getQuote method. All of the necessary values required to create the SOAP message (that is, MethodName, NameSpace, and so on) are set in this method. The input parameter values are organized into an array of Java Parameters and set to the class. Finally the invoke method is called, which is where the sending and receiving of the SOAP message takes place. This is very similar to the Java agent (SOAPCall) described in 4.2, “SOAPConnect TempfinderSample” on page 101.

After the invoke method is called from the class Call, the server processes the SOAP request and sends the SOAP response. This response is set to the variable resp of type Response. If a fault was encountered anywhere in the SOAP transaction, a new SOAPException is created and processed. If the SOAP transaction completed successfully, the return value in resp is extracted and set to the variable refValue of type Parameter. This value is then returned to the calling function. From this point on, your Web Service consumer can utilize the return value as needed.

Archived

4.4.3 Web Service consumer Java agent

In the SOAPConnect example discussed in 4.1.3, “What is SOAPConnect?” on page 100, its core functionality was based within the (SOAPCall) Java agent. To create this agent, the developer must have a great deal of Java programming skills. This is the ultimate goal of the SOAPConnect project, to supply this advanced Java agent giving the large community of LotusScript developers the ability to consume Web Services easily. Here we will show you how to incorporate the proxy class in GetStockQuote Java agent. By doing this step, the level of required Java programming skill to develop a Web Service consumer will drastically decrease.

Customizing GetStockQuote Java agent

This section will cover the Java agent in a component-by-component fashion. First we discuss the Notes programming portion of the agent. Next we show how to perform the call to the proxy class. Finally we incorporate all the pieces together and show the agent’s runtime flow.

Java Notes programming

All of the parameters that are needed by this Java agent were previously saved to a NotesDocument. This document was created and saved by the initial LotusScript agent in this consumer. When the LotusScript calls the Java agent, it passes it the document ID of the parameter NotesDocument. Refer to 4.1.1, “Web Service consumer introduction” on page 98 for more details on this technique.

This agent first starts by creating a Session and defining the current database, as shown in Example 4-6. It then searches the database for the document that has the document ID passed to it. After it is found, the agent scans the document and stores all of the values needed to local variables.

Example 4-6 GetStockQuote - Java Notes programming

```
import lotus.domino.*;
import java.util.*;
import java.net.*;
import proxy.soap.*;

public class JavaAgent extends AgentBase {
   public void NotesMain() {
        Document doc = null;
      try {
         Session session = getSession();
         AgentContext ac = session.getAgentContext();
         Database db = ac.getCurrentDatabase();
         Agent agent = ac.getCurrentAgent();

         // Retreive Arg from Agent Data document
         doc = db.getDocumentByID(agent.getParameterDocID());
         Vector params = doc.getItemValue("Symbol");
         String symbol = (String) params.elementAt(0);

         // Create Proxy Instance from StockQuoteServiceProxy Class
         ...
         ...
         ...
         // Save the return value to the NotesDocument and save
         doc.replaceItemValue("quote", (new Float (result)));
         doc.save();
      } catch(Exception e) {
         e.printStackTrace();
```

Archived

```
        }
    }
}
```

The final step is to set the return value to the corresponding item on the Document and save it. If there is an Exception thrown, the agent prints the call stack to the Java console and exits.

Importing StockQuoteServiceProxy.class into your current project

"Java Notes programming" on page 119 depicts how the agent obtains the values for all of the required parameters. Here we will use the auto-generated proxy class to simplify our agent. We first must add this external Java class to our new agent. Importing is the process of telling the Java agent what "external" classes it is able to use. To do this, we will need to know a valid path (local or network) to the StockQuoteServiceProxy.class file.

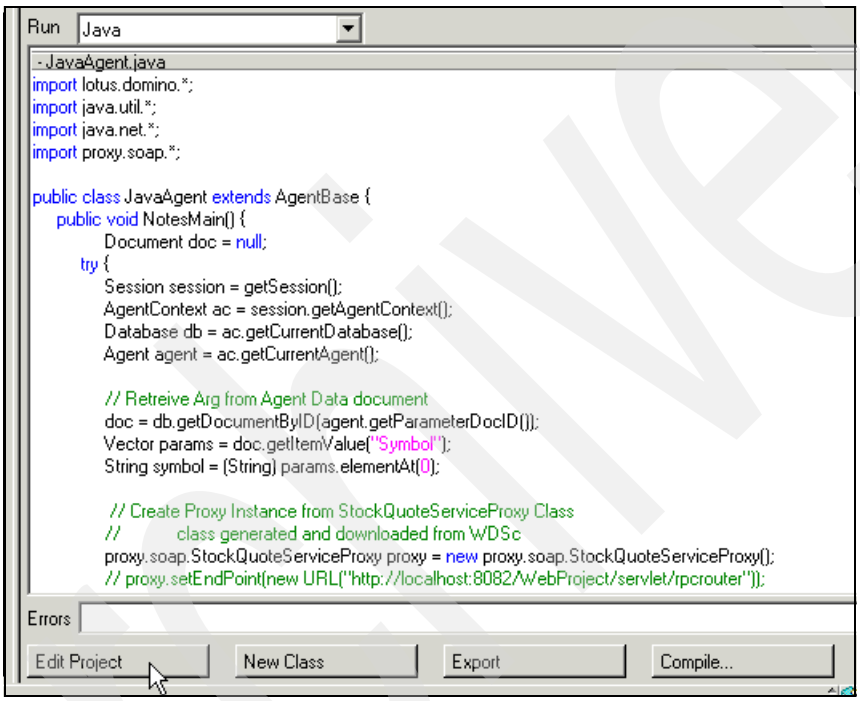


Figure 4-19 Edit Project button in programmer's pane

Tip: When you copy a .class to your local Windows PC, make sure you create the exact directory structure of the class package. For example, The StockQuoteServiceProxy.class has a fully qualified class name as proxy.soap.StockQuoteServiceProxy.class. When saving this .class file, you must save it to:

<drive letter>:\<root>\proxy\soap\StockQuoteServiceProxy.class

Follow these steps to import an external Java class file into your current project:

1. First start by selecting the **Edit Project** button, as show in Figure 4-19.
2. In the Base Directory drop-down box, select the <root> directory of your class package as shown in Figure 4-20. In this case, our <root> directory is c:\classes.
3. You will notice in the **Available Java Files** pane, the **proxy** directory. If we expand the entire directory structure, you will see a **soap** directory and finally the actual **StockQuoteServiceProxy.class** file. Highlight the file and click the **Add/Replace Files**

button. The file will show up on the right side of the window in the **CurrentAgentFiles** pane.

4. Select **OK** to officially import the package into your project.

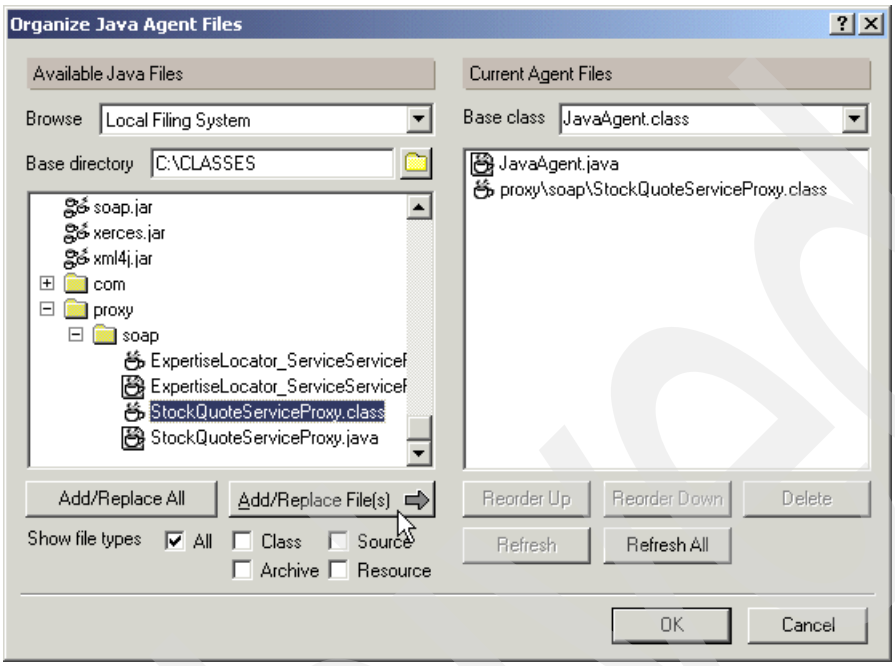


Figure 4-20 Importing class package into current Notes Java project

Pressing **OK** will take you back to the main Programmer's Pane in Domino Designer.

Tip: Instead of importing the required classes into your project, you can add these files to your classpath by directly specifying them in the notes.ini of your Notes client or Domino server. Doing this will **greatly** increase the performance of your Java agent.

If you require .jar files, specify the full path and filename; if you are using .class files, only the full path is needed. In both cases, add the following line to the end of your notes.ini:

JavaUserClasses = class1.jar; class2.jar; C:\classes

Calling the proxy class

After importing the file, your Java project now knows about this new class. There are two ways the developer can specify in the code which components need to be accessed. The first way to do this is to use the **import proxy.soap.*;** statement. You can see this in Example 4-7 on the 4th line of the code.

The other way to utilize a component of the package is by using the fully qualified class name when making your call. In the code examples, you will notice both ways are used. This is not required. We have illustrated the syntax of both techniques primarily for example purposes.

Example 4-7 Proper usage of the proxy client class

```
import lotus.domino.*;
import java.util.*;
import java.net.*;
import proxy.soap.*;
```

```
public class JavaAgent extends AgentBase {
   public void NotesMain() {
         Document doc = null;
      try {
         ...
         // Create session and get NotesDocument
         ...

         proxy.soap.StockQuoteServiceProxy proxy =
            new proxy.soap.StockQuoteServiceProxy();
         proxy.setEndPoint(
            new URL("http://idomws1:5500/WebProject/servlet/rpcrouter"));
         float result = (float) proxy.getQuote(symbol);

         ...
         // Set and Save return value to NotesDocument
         ...
      }
   }
}
```

The Java code in Example 4-7 shows us the proper usage of the proxy client class. We first create a new instance by using the fully qualified name of the StockQuoteServiceProxy class and call **proxy**.

```
proxy.soap.StockQuoteServiceProxy proxy = new proxy.soap.StockQuoteServiceProxy();
```

Once the instance of StockQuoteServiceProxy has been created, the EndPoint (location of the Web Service's SOAP rpcrouter) must be set.

```
proxy.setEndPoint(new URL("http://idomws1:5500/WebProject/servlet/rpcrouter"));
```

The StockQuoteServiceProxy has a method called **getQuote** as shown in Example 4-5 on page 117, which initiates the SOAP message request. To execute this, we call the method **getQuote** and set its return value to a variable called **result** of type float.

```
float result = (float) proxy.getQuote(symbol);
```

The agent will then continue by processing the return value as shown.

Complete customized GetStockQuote Java agent

In the past few steps we have defined sections of a Java agent that utilize a proxy client Java class. As you can see, this agent requires very minimal Java programming skill. In Example 4-8, we have put all of these pieces of the agent together.

Example 4-8 Complete GetStockQuote Java agent

```
import lotus.domino.*;
import java.util.*;
import java.net.*;
import proxy.soap.*;

public class JavaAgent extends AgentBase {
   public void NotesMain() {
         Document doc = null;
      try {
         Session session = getSession();
         AgentContext ac = session.getAgentContext();
         Database db = ac.getCurrentDatabase();
         Agent agent = ac.getCurrentAgent();
```

Archived

```
            // Retreive Arg from Agent Data document
            doc = db.getDocumentByID(agent.getParameterDocID());
            Vector params = doc.getItemValue("Symbol");
            String symbol = (String) params.elementAt(0);

            // Create Proxy Instance from StockQuoteServiceProxy Class
            //    class generated and downloaded from WDSc
            proxy.soap.StockQuoteServiceProxy proxy =
               newproxy.soap.StockQuoteServiceProxy();
            proxy.setEndPoint
               (new URL("http://idomws1:5500/WebProject/servlet/rpcrouter"));
            float result = (float) proxy.getQuote(symbol);

            doc.replaceItemValue("quote", (new Float (result)));
            doc.save();
         } catch(Exception e) {
            e.printStackTrace();
         }
      }
}
```

Initial LotusScript agent

Just as in the examples in SOAPConnect, we want to make the GetStockQuote Java agent transparent to the developer. By doing this, we allow the vast community of LotusScript developers to create Web Service consumer agents quickly and easily. The developer will only be “calling” GetStockQuote Java agent, he/she does not need to understand it.

The initial LotusScript agent that calls the GetStockQuote Java agent must only package the critical values onto a NotesDocument and pass the UNID to the Java agent when called. This example will be a very short and generic approach to creating a LotusScript agent that consumes a Web Service.

You first need to create a form within your consumer database to store your critical values. For this example, we create a form, stockQuote, with two fields on it: symbol and quote. The following LotusScript code saves the values to this form, then calls the GetStockQuote Java agent to retrieve the stock price value. The value is then saved to the quote field on the form, and the agent exits.

Example 4-9 LotusScript agent to consume StockQuoteService Web Service

```
Sub Initialize
   Dim s As New notessession
   Dim db As notesDatabase
   Set db = s.currentDatabase
   Dim symbol As String
   Dim doc As NotesDocument
   Dim stockAgent As notesAgent
   Set doc = New NotesDocument(db)
   doc.form= "stockQuote"

   If (symbol$="") Then
      symbol$ = Cstr(Inputbox$("Please enter the Stock symbol to lookup: ", "Stock Symbol
Lookup", "IBM"))
   End If
   doc.symbol = symbol$

   'Make the call (by calling a Java Agent that uses SOAP4J)
   If ( doc.Save( False, False ) = True ) Then
      docID = doc.NoteID
```

```
      Print "SOAPCall Document " & docID & " successfully saved"
   End If

   ' Get a handle on the Java agent, and invoke the call
   Set stockAgent = db.getAgent("GetStockQuote")
   Print "Calling Java StockAgent"
   If stockAgent.Run(doc.NoteID) = 0 Then
      Print "Agent 'GetStockQuote' call ran successfully"
   End If

End Sub
```

You can see from the code above, what is required is very minimal. We need to make sure the form, the LotusScript agent, and the Java agent GetStockQuote agents are all in the same database, and we execute the LotusScript agent from the Actions menu. This is only an example. You can integrate any part of this into your existing application to add more power and functionality quickly and easily.

4.5 Creating a proxy client from a WSDL file

Up until now, it has been assumed that we are able to use the proxy client supplied with SOAPConnect or your Web Service provider has supplied you with a client class. What if this proxy client is not supplied for you? How would you go about creating one yourself?

There are many ways to go about creating your own proxy client class. This redbook has shown us examples of what the developer is required to do when consuming a Web Service. You can also find hundreds of examples scattered on various Web sites located throughout the Internet. For intermediate to advanced Java developers, the syntax and functionality can be quite easy when using these examples as reference. But each Web Service is quite unique. Each Web Service has its own identity spelled out by its SOAP server address, to its URI namespace down to the name of the method called in its core, just as we have our own fingerprint. How does a developer uncover the Web Services fingerprint? The answer lies within the Web Service's, Web Service Definition Language (WSDL) file.

This section will continue by describing WSDL files and illustrating examples. We will also discuss where can we find WSDL files for given Web Services. And finally we will guide you step-by-step in creating your own proxy client class from a WSDL by hand and by using auto-generation tools, such as WebSphere Development Studio client for iSeries, WDSc.

4.5.1 What is a WSDL file?

As mentioned above, WSDL means Web Service Definition Language. This is a file that contains the fingerprint of the Web Service. This file is XML based that contains everything a consumer would need to know about the Web Service. In this section we detail what a WSDL file contains and how we can use it. In Example 4-10, you can see a WSDL file for a Country Population Lookup Web Service.

Example 4-10 WSDL file for Country - Population Lookup Web Service

```
<?xml version="1.0" encoding="UTF-8" ?>
   <wsdl:definitions
      targetNamespace="http://cs.uga.edu:8080/axis/services/urn%3acountryInfoLookup"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:intf="http://cs.uga.edu:8080/axis/services/urn%3acountryInfoLookup"
```

```
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:impl="http://cs.uga.edu:8080/axis/services/urn%3acountryInfoLookup-impl"
      xmlns="http://schemas.xmlsoap.org/wsdl/">
   <wsdl:message name="getPopulationResponse">
      <wsdl:part name="return" type="xsd:string" />
   </wsdl:message>
   <wsdl:message name="getPopulationRequest">
      <wsdl:part name="in0" type="xsd:string" />
   </wsdl:message>
   <wsdl:portType name="CountryInfoLookup">
      <wsdl:operation name="getPopulation" parameterOrder="in0">
         <wsdl:input message="intf:getPopulationRequest" />
         <wsdl:output message="intf:getPopulationResponse" />
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="urn%3acountryInfoLookupSoapBinding" type="intf:CountryInfoLookup">
   <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
   <wsdl:operation name="getPopulation">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input>
         <wsdlsoap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://cs.uga.edu:8080/axis/services/urn%3acountryInfoLookup" />
      </wsdl:input>
      <wsdl:output>
         <wsdlsoap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://cs.uga.edu:8080/axis/services/urn%3acountryInfoLookup" />
      </wsdl:output>
   </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="CountryInfoLookupService">
      <wsdl:port name="urn%3acountryInfoLookup"
         binding="intf:urn%3acountryInfoLookupSoapBinding">
      <wsdlsoap:addresslocation=
         "http://cs.uga.edu:8080/axis/services/urn%3acountryInfoLookup" />
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

From glancing at the WSDL file above, a few items should stand out that you have learned previously in this redbook. The first and foremost element is the targetNamespace. This is also known as the SOAP server or rpcrouter, of the Web Service. Next you should notice the getPopulationRequest and getPopulationResponse. These are the definitions for the actual parameters that will be passed to the Web Service and returned from the Web Service. In this case you can see that both parameters are of type **string**.

After this, you will come to the operations section of the WSDL file. This is the definition space of what operations are available to be performed by this Web Service. In our case we only have one operation “getPopulation”. Other Web Services may have many operations depending on the function they provide. The getPopulation operation refers to the methodName located within the Web Service that does the work. You will see subsections, we have input and output. These sections are just as they sound. The input section tells us the encoding style of the procedure and the NameSpace it utilizes. The output section here tells us the same information.

You as the developer can gain the following information from this file when writing a proxy client for your consumer application.

- targetNamespace — EndPoint URL of Web Service
- operation
  - methodName
  - soapAction
  - encodingStyle - input and output
  - namespace

For a more detailed explanation of the syntax and components of WSDL visit:

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

4.5.2 Finding a WSDL file

Since the WSDL file contains the key information to successfully utilizing a Web Service, we must somehow be able obtain this information. The standard way to find a Web Service to consume is through Universal Description, Discovery and Integration, or UDDI. UDDI is a "yellow pages" per se, for Web Services. Web Services are listed in this directory by category. You can learn more about UDDI and how to use it by reading Appendix B, "Using UDDI for both ends — publishing and requesting" on page 177.

For the following example in 4.5.3, "Generating a proxy client class with WDSc" on page 127, we will obtain the WSDL file from `http://www.xmethods.com` (Figure 4-21).

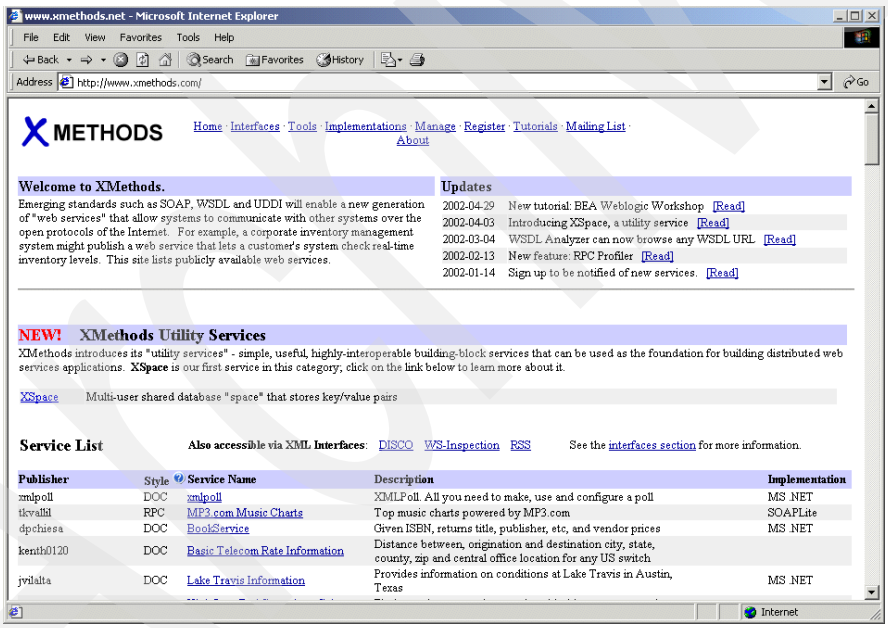


Figure 4-21 http://www.xmethods.com — Web site

Xmethods.com is a Web site that is a directory for Web Service providers to post their available services. On the site you will find many services along with contact information and documentation on the proper usage of the Web Service. Along with this information you will find a WSDL file for the given Web Service.

For the remainder of this chapter we will utilize the Web Service found on Xmethods.com:

`Country - Population Lookup Service - Gives the Population for a given country`

By visiting the details page for this Web Service, you are able to download the WSDL to your local PC. There are several tools that are available today that will auto-generate a proxy client

class for you by only supplying it the WSDL file of the Web Service you want to consume. We detail and provide a step-by-step illustration on how you can create your own proxy client using WDSc.

4.5.3 Generating a proxy client class with WDSc

One of the many powerful parts of WDSc is the ability to generate a proxy client class for the developer by only supplying it the WSDL file of the Web Service. For this demonstration, we assume that you have downloaded the CountryInfoLookup.wsdl file from the Web Service we mentioned above, Population Lookup Service.

The following steps are very similar to what we have shown in Chapter 2, “Writing my first Web Service application” on page 9.

Create a new Web Project

First of all, when doing basically anything in WDSc we need to create a **New Project**. You can do this by first selecting **File -> New -> Project**, as shown in Figure 4-22.

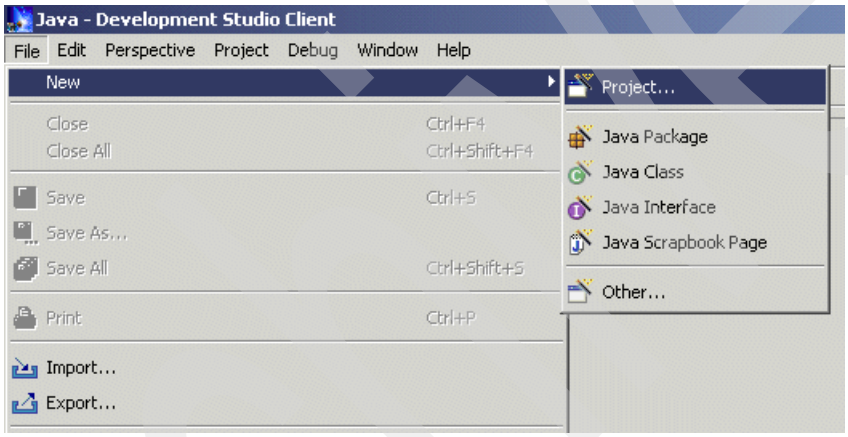


Figure 4-22 Menu to create a new project

Archived

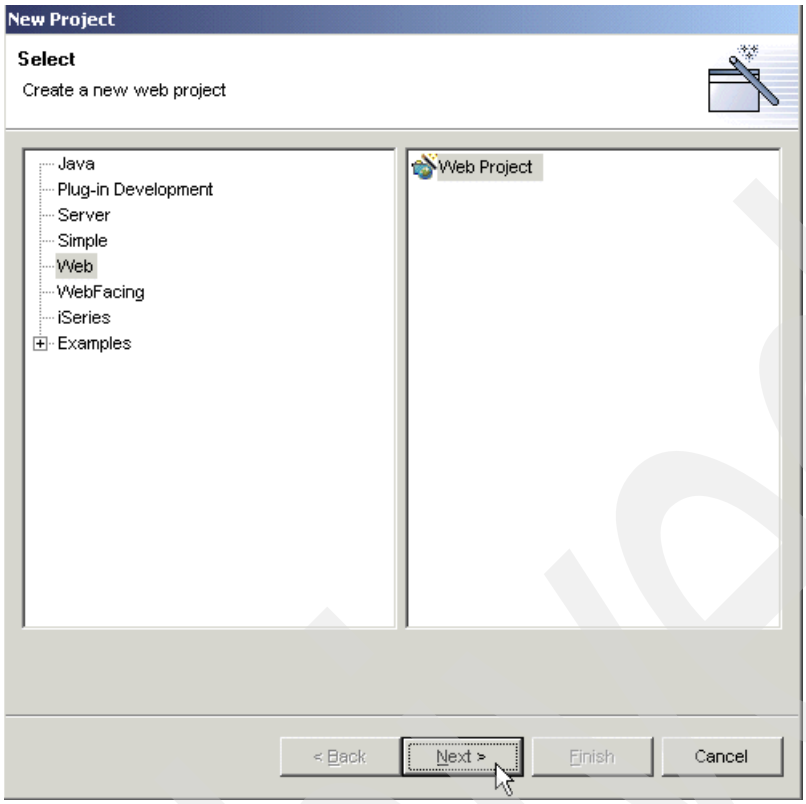


Figure 4-23 Web -> Web Project

After selecting to create a new project, you will get a window, shown in Figure 4-23. In this selection window we indicate that we want to create a new *Web Project*. This will let WDSc create the required elements for a Web application.

For a description of each type of project and their use, consult WebSphere Development Studio client for iSeries help.

Once you have chosen the correct type of project, continue by selecting **Next >**.

Archive

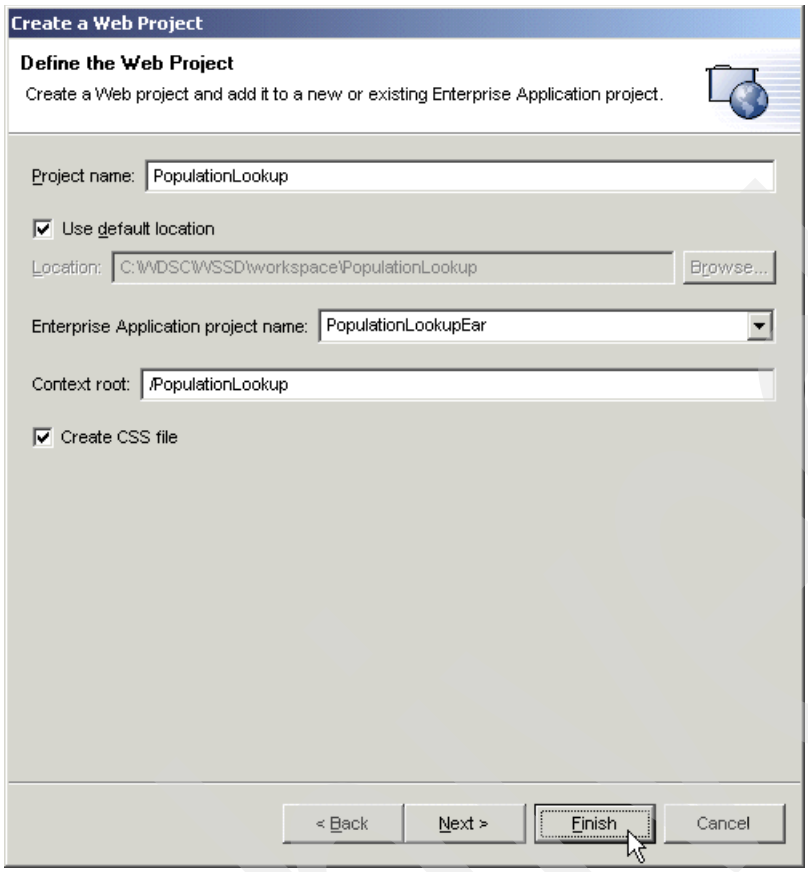


Figure 4-24 Define the Web Project window

In this wizard window, you will need to define a name to your new project. In this example we have named our new project **PopulationLookup**. If your project does not require to be stored to a specific location, assure the **Use default location** check box is checked. For the purpose of this section, generating a proxy class, we will not be deploying anything to our WebSphere Application Server. The remaining options do not apply to this section. For a description of their purpose, refer to 2.2, “Developing and deploying the applications” on page 14.

These are all the steps required to create a new Web Project in WDSc. Close this wizard by selecting **Finish**.

Importing WSDL file into project

When we create a new Project, this project can be thought of as a “container” for all of our files. It creates a default directory structure for us, and generates some basic default files. If we want to add a file to our project, we do this through a process called *importing* the file.

This section takes you step-by-step through the process of importing our downloaded WSDL file into our current Web Project.

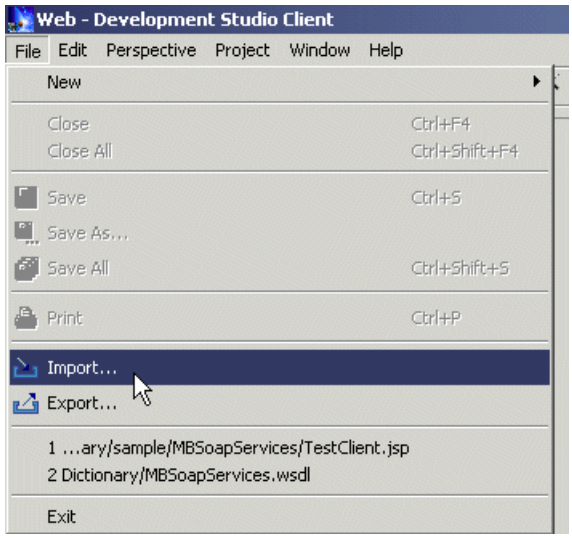


Figure 4-25 File -> Import menu

Since we have gathered the WSDL and saved it to our local PC, we will need to import this file into our project. This process is very easy. We first start by selecting **File-** > **Import**, as shown in Figure 4-25.

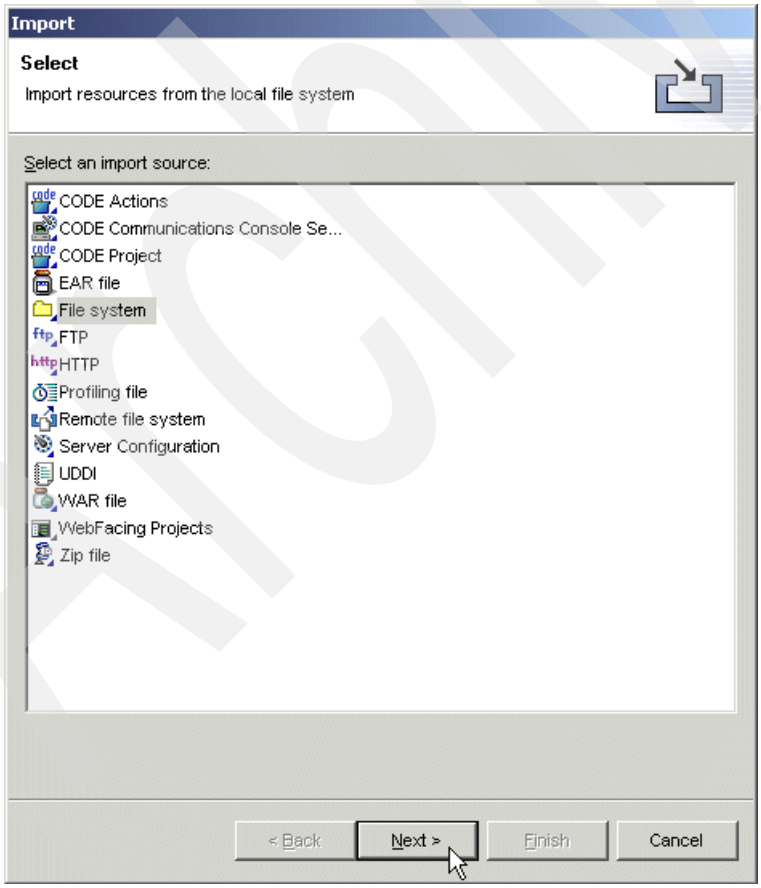


Figure 4-26 Import type selection window

Archived

This menu item will pop up the window illustrated in Figure 4-26. This window gives us several choices of the origination of the WSDL file. In our case it is stored on our local PC, so we will select File System.

As you can see there are several options, if we wanted to pull this file directly from the Web Server, we would select HTTP, if your source file is located on a FTP server, you can select the option to import your file directly from this server. You can even specify to directly get the WSDL file straight from UDDI. We will not detail the mentioned options for importing a file, but as always, you can get more information by consulting the WDSc help file.

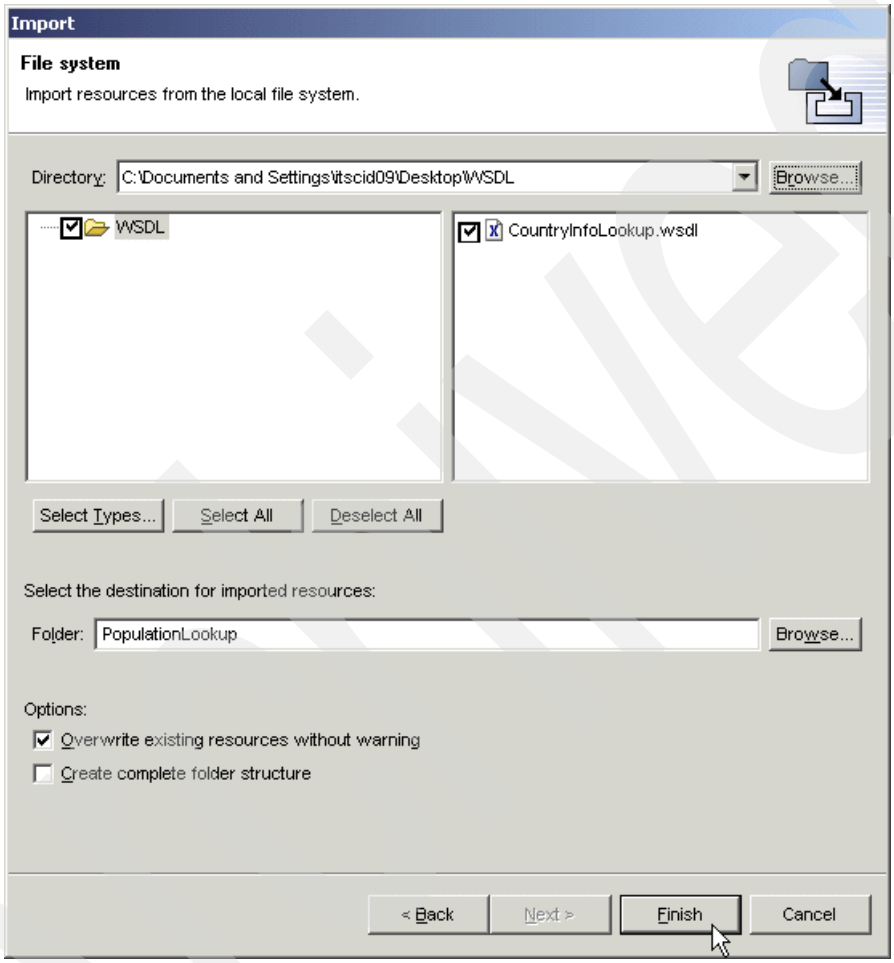


Figure 4-27 Choosing the location and destination of the WSDL file to import

Now that we have chosen to import our WSDL file from our File System, we need to browse to find the location of this file. Do this by selecting the Browse button at the top, right-hand side of Figure 4-27. In this particular example the file was located on the our Desktop in a WSDL folder. You can see this directory structure above.

Select the appropriate file in the right-hand pane of this window. Also, pay close attention to the destination folder. This should default to the folder of your current project, but if it does not, press the browse button in this section to choose the **PopulationLookup** folder/Web project.

The options section gives you the choice to overwrite any files as needed without warning, and allows you to retain the current directory structure on the location of the WSDL file. In our case, we will have a Documents and Settings\ ... ...\WSDL directory structure in our Web project. In our case, we only want to overwrite files without warning.

Creating the proxy client class

In this section we will walk though the WebServiceClient wizard. By completing this wizard, WDSc will automatically generate a proxy client class, and a Web-enabled TestClient by use of JSP.

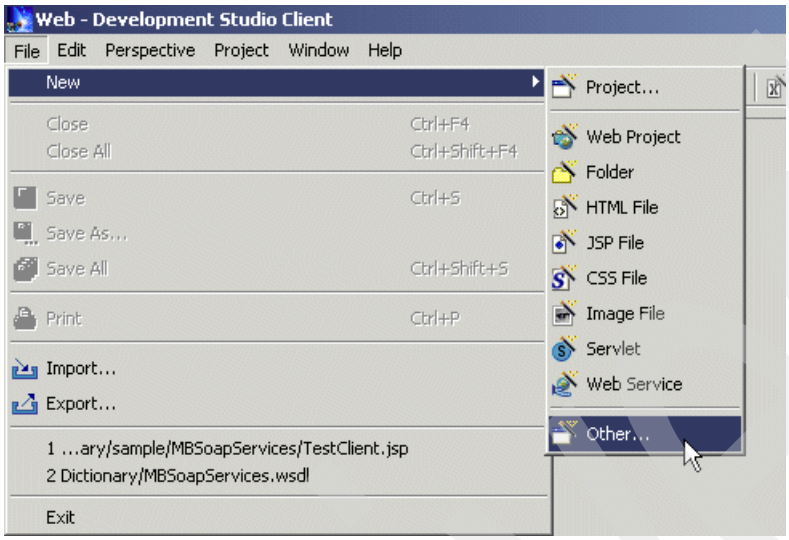


Figure 4-28 File -> New -> Other menu

We first start by selecting **File -> New -> Other** to chose what we want WDSc to create for us, as shown in Figure 4-28.

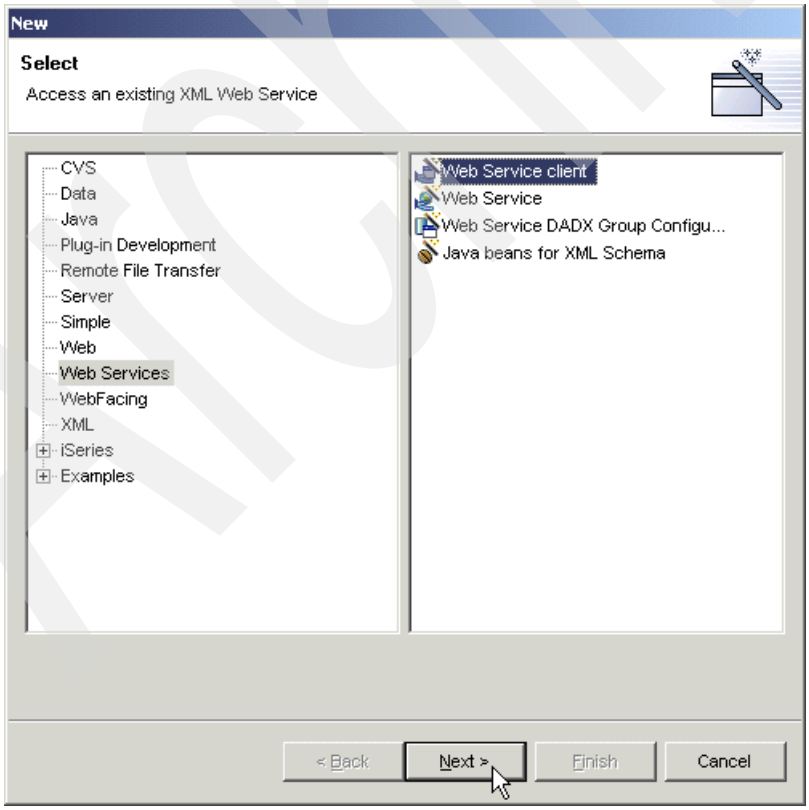


Figure 4-29 Selecting the creation of a WebServiceClient

In the type selection window, shown in Figure 4-29, we chose **Web Services** and Web Service client. By choosing this, we are only telling WDSc we want it as a *client* for Web Service.

Continue from this window by selecting **Next >**.

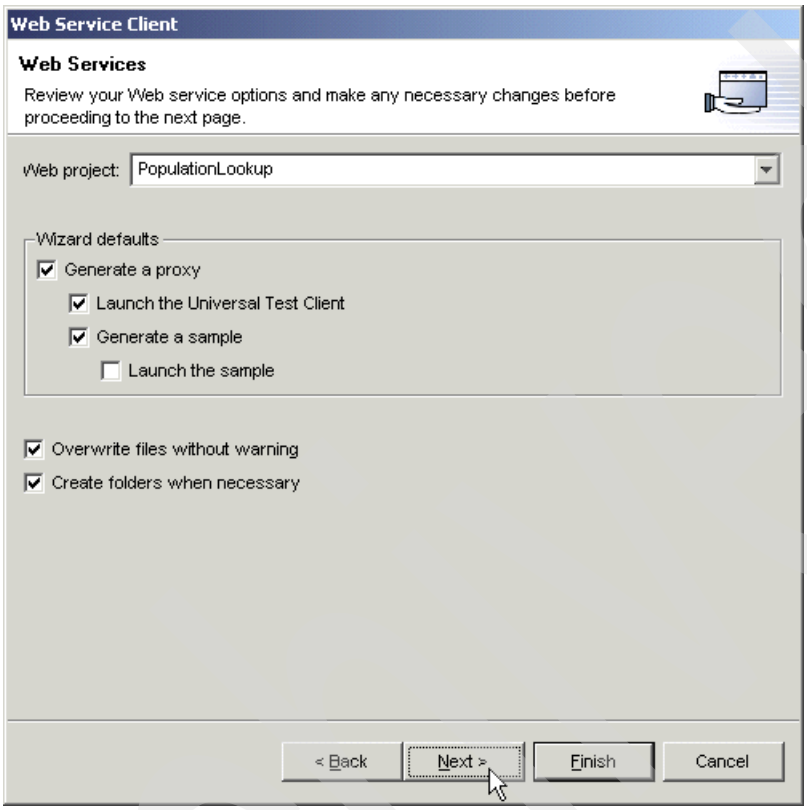


Figure 4-30 Selecting to Generate a proxy

The Figure 4-30 allows us to select several options of what we want this wizard to do for us. We first of all want to specify in the Web project field the current project where we want this client to be stored. If the field does not contain the current **PopulationLookup**, use the drop-down arrow and select it from the list.

Next we will want to tell the wizard we want it to **Generate a proxy**. This option will create the actual .java and .class file for our proxy client. Below that option you will see Launch the Universal Test Client and Create a sample. This will tell WDSc to generate the JSP pages to utilize the proxy class for testing.

The remaining options are purely your choice, such as overwrite files without warning, and create folders when necessary. For this example we have selected both options.

As always, continue by pressing **Next>**.

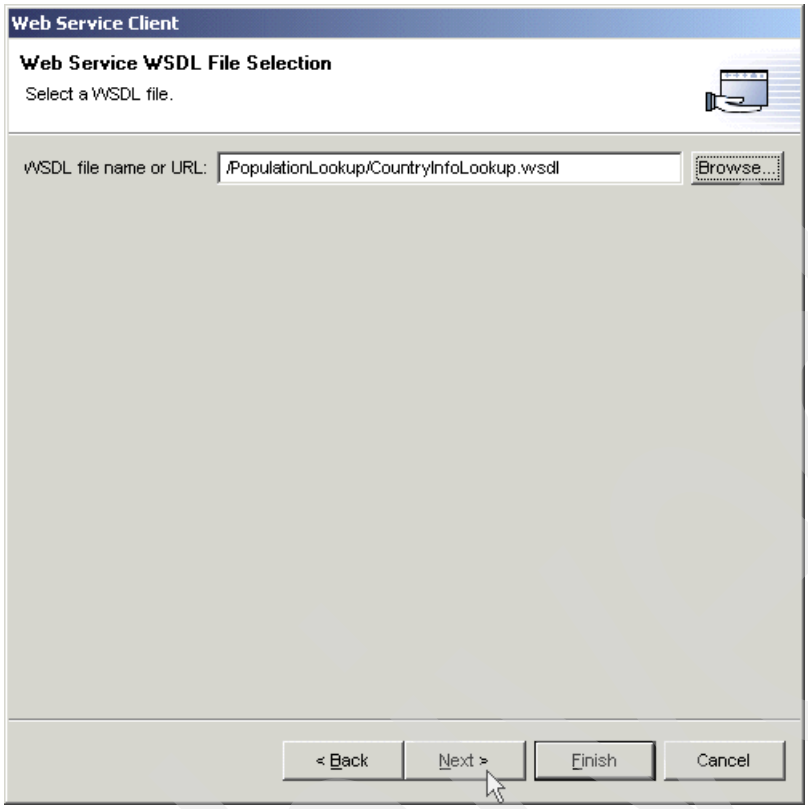


Figure 4-31 Locating the WSDL file of your WebService

Here is where we tell WDSc the specifications of the Web Service that we want to consume. We need to specify, in the field shown in Figure 4-31, the location of the WSDL file. This window gives us the option to select the file from our project or even from a URL. We could type in the URL of WSDL file from where we saved it originally, and WDSc will access the file from the Web and store it to our project. Or, we could choose the browse button to find the file in our current project. This is shown in Figure 4-32.

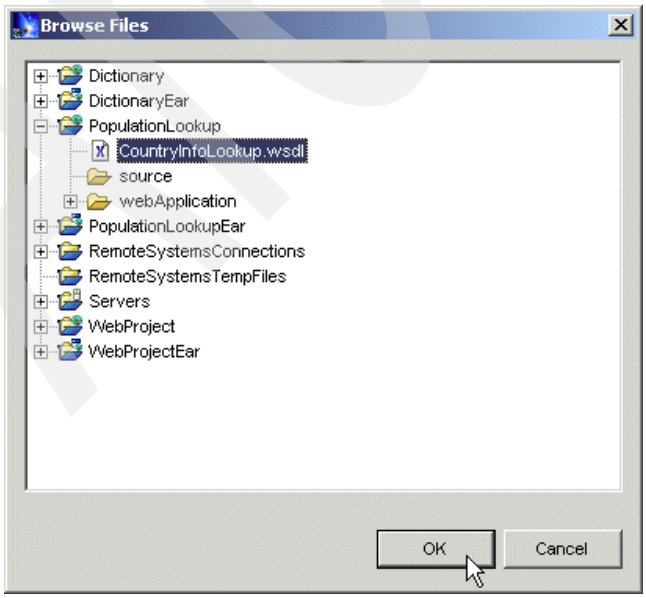


Figure 4-32 Browsing files for Web Service's WSDL file

After you have specified the WSDL file and its location, continue on by pressing **Next>**.

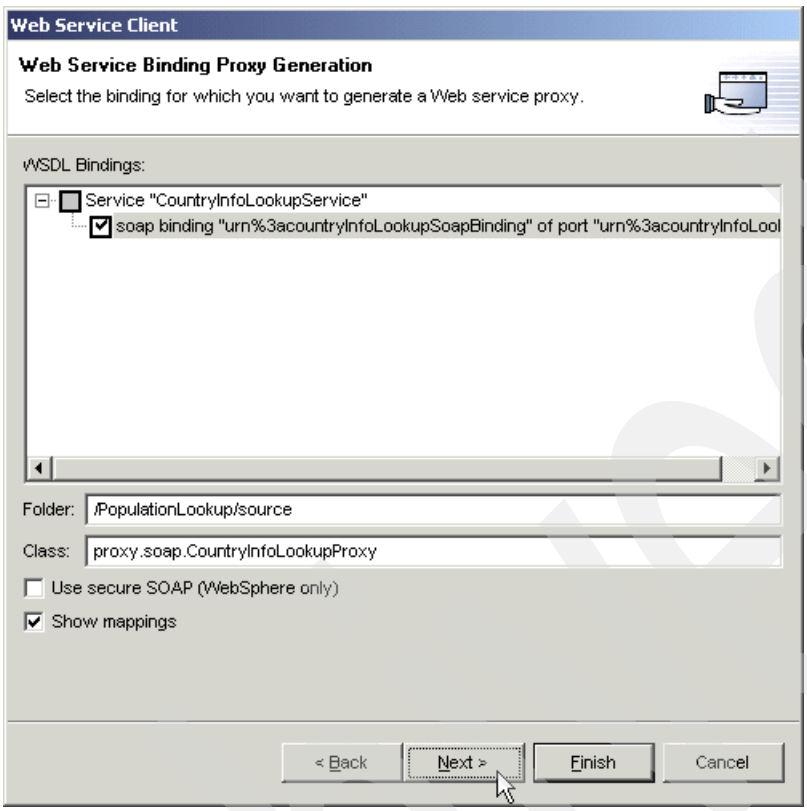


Figure 4-33 Web Service Binding Proxy Generation window

Figure 4-33 displays the Web Service Binding Proxy Generation options. Unless you want to store the files in a specific location, keep the location default values already in this window. Assure you have the **Show mappings** check box checked. This will be required for a later part of this wizard. Continue to the next window by pressing **Next>**.

Archived

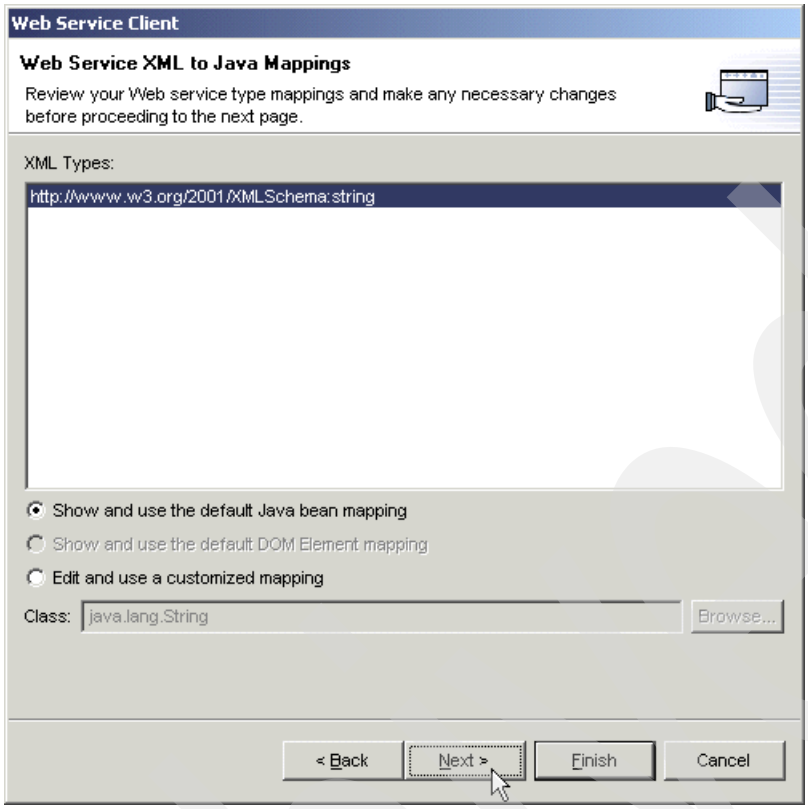


Figure 4-34 Web Service XML to Java Mapping window

The wizard allows us to specify a custom mapping of the XML to Java. In our case we are going to use the default Java bean mapping as shown in Figure 4-34. But if we had a specialized class for XML to Java mapping we would choose the radio button *Edit and use a customized mapping* and locate the class by use of the browse button.

Continue to the next section of the wizard by choosing **Next>**.

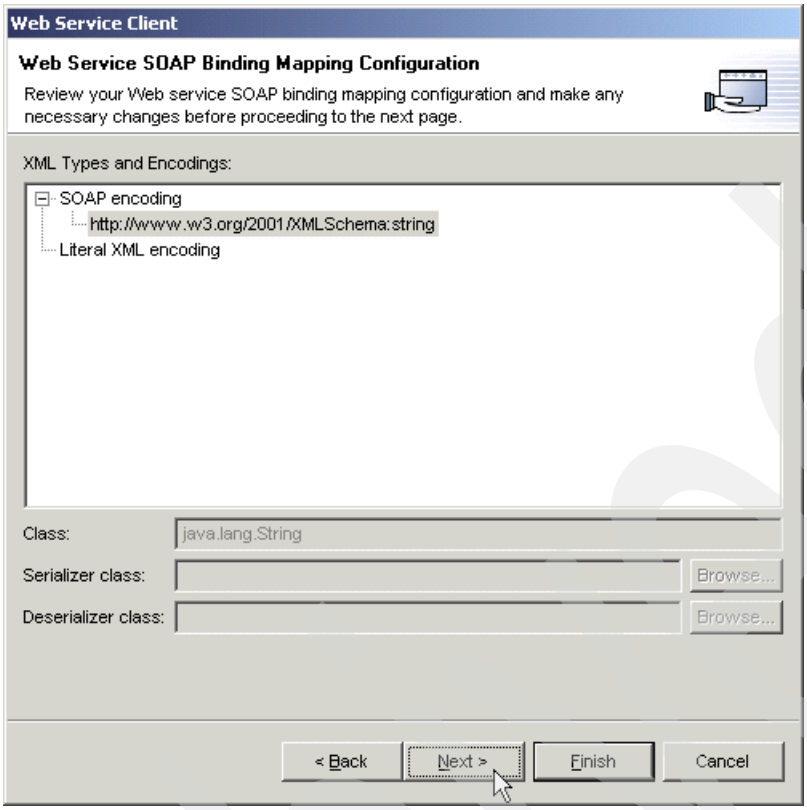


Figure 4-35 Web Service SOAP Binding Mapping Configuration window

The SOAP Binding Mapping Configuration window, Figure 4-35, would normally allow us to specify a Serialize/Deserializer class if we chose the Edit and use customized mappings radio button depicted in Figure 4-34. The Serializer/Deserializer classes are used when the Web Service requires a specialized, user defined, class as an input parameter or if the Web Service returns a class as a result. These classes are used for allowing the consumer to access the elements of this special class.

In our case we are only sending and receiving a String value, so we do not need to specify a special class. Continue to the next window by selecting **Next>**.

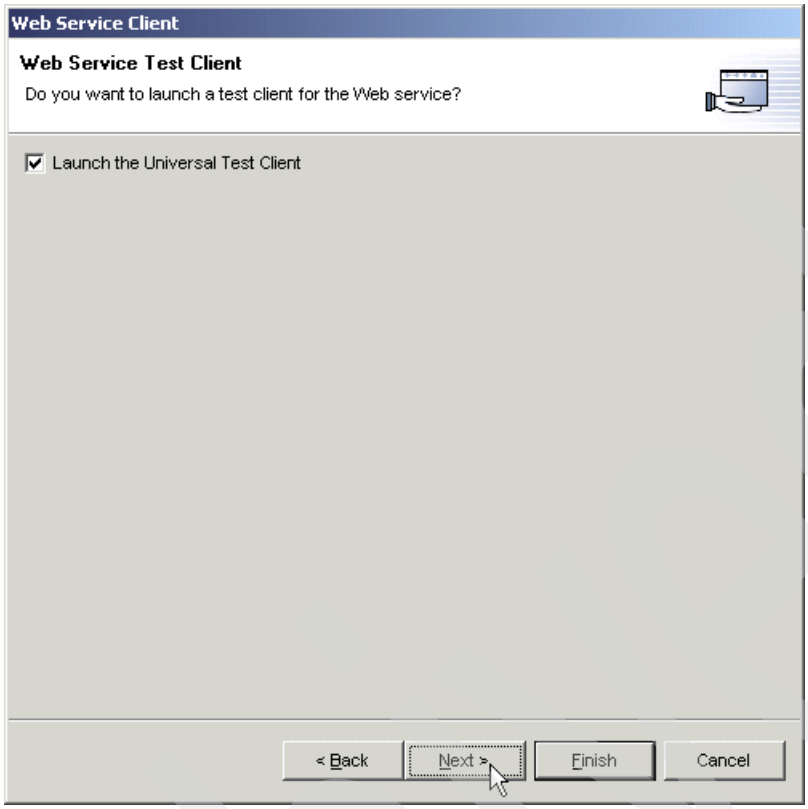


Figure 4-36 Web Service Test Client window

The window in Figure 4-36 is very simple. We only have to choose to Launch the Universal Test Client. If you select this button, when the wizard finishes and returns to WDSc, a browser within WDSc will be displaying a default WDSc test client, not the TestClient.jsp as mentioned earlier. This option is purely a preference. If you do not want the take the time to test your new client, you can uncheck this option to return to your Web Project perspective.

Select **Next>** to continue to the last section of this wizard.

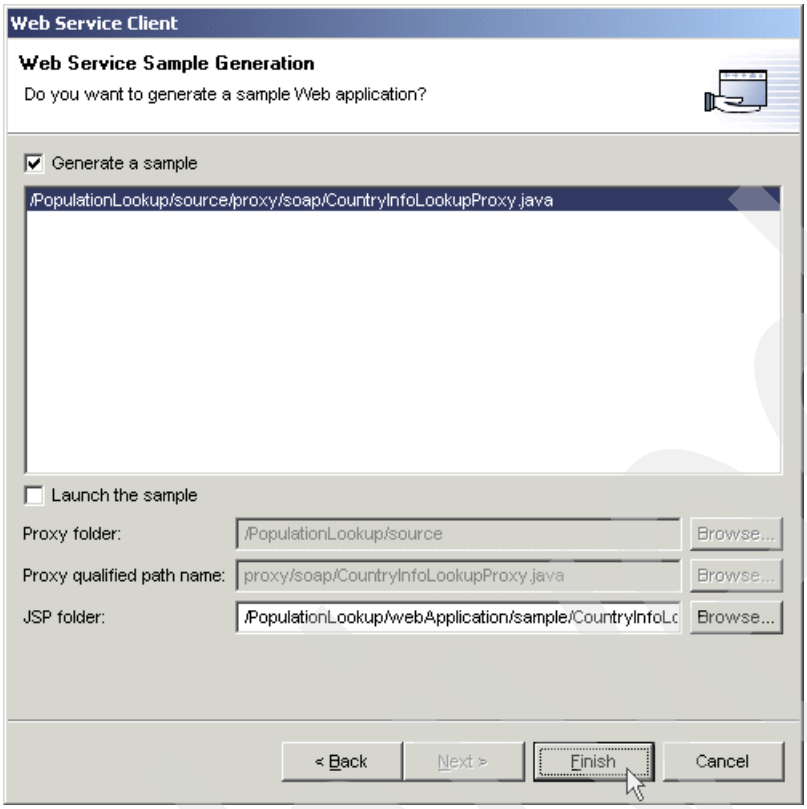


Figure 4-37 Web Service Sample Generation window

We have finally made it to the last section of the Web Service client wizard in WDSc. This final window is displayed in Figure 4-37. All of the options we needed to create a proxy client class have already been selected. This window gives you, the developer, the option to create a Web based Test Client by using the proxy created, and JSP. This is purely an option, because the proxy class has already been created.

To create this TestClient.jsp, check the box *Generate a sample*. If you remember all the way back to Figure 4-30 on page 133, you had the choice to Generate a Sample, and Launch the Universal Test Client. If we left these options unchecked, the wizard would have finished a few steps prior.

You also have the option to Launch the sample this wizard created. To do this, check the button **Launch the sample** and make sure the proxy and JSP folders are correctly specified. By selecting this check box, when you selected Finish, WDSc will start its Web browser and display the created TestClient.jsp. To finish this wizard, press **Finish**.

Congratulations!!! You have successfully navigated through the Web Service client wizard in WDSc. This wizard created a minimum of two files for you. First of all it created the proxy client .java file, located in:

```
PopulationLookup/source/proxy/soap/CountryInfoLookupProxy.java
```

The second file was created in the .class file compiled from the CountryInfoLookupProxy.java listed above. This file is located in:

```
PopulationLookup/webApplication/WEB-INF/classes/proxy/soap/CountryInfoLookupProxy.class
The directory and file structure after the wizard is illustrated in
```

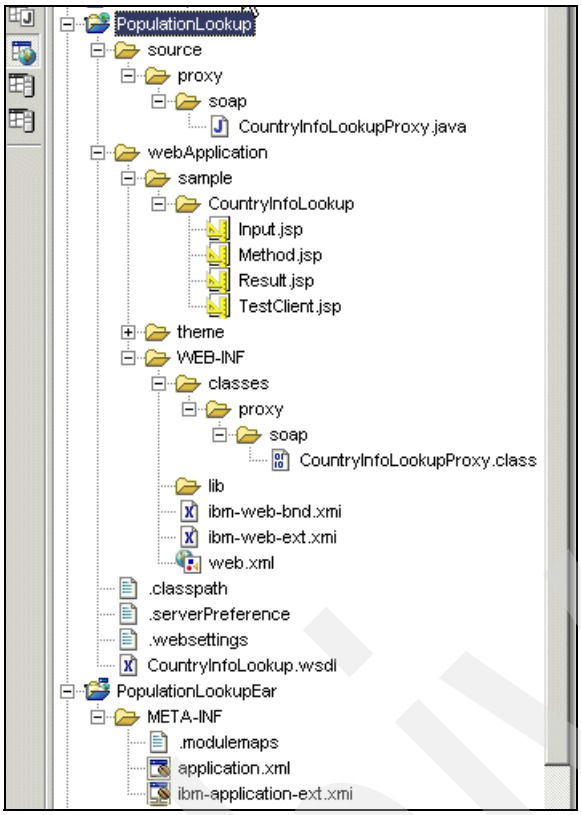


Figure 4-38 Directory and file structure after WDSc Web Service client wizard

Exporting the proxy client class from WDSC

In the previous section we created a proxy client .java and .class file in our WDSc project, PopulationLookup. The first steps to using this proxy in your Domino application is to export this class to the local file system of your PC. This section walks you through the exporting process for use in your Domino application.

The first step is to select the file you want to export. In our case we will technically only need the CountryInfoLookupProxy.class file. We also may want to export the .java file for quick reference if we will not have access to WDSc to view this file.

Click the CountryInfoLookupProxy.class file and select **File->Export**, shown in Figure 4-39.

Archived

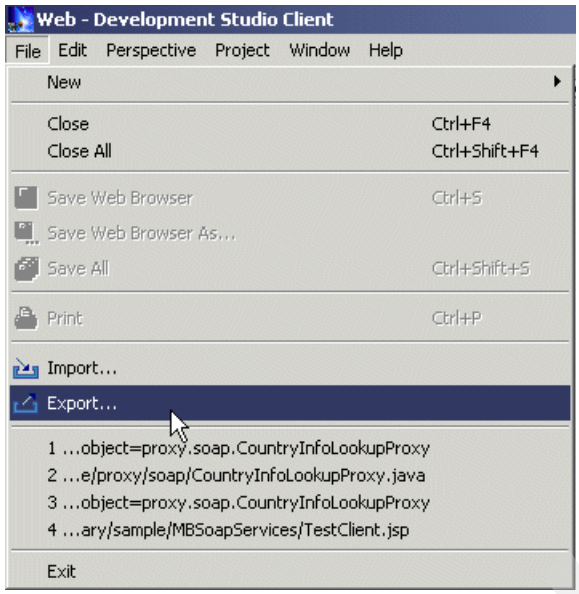


Figure 4-39 Exporting a file from WDSc

After the Export menu item is selected you should see the window shown in Figure 4-40.

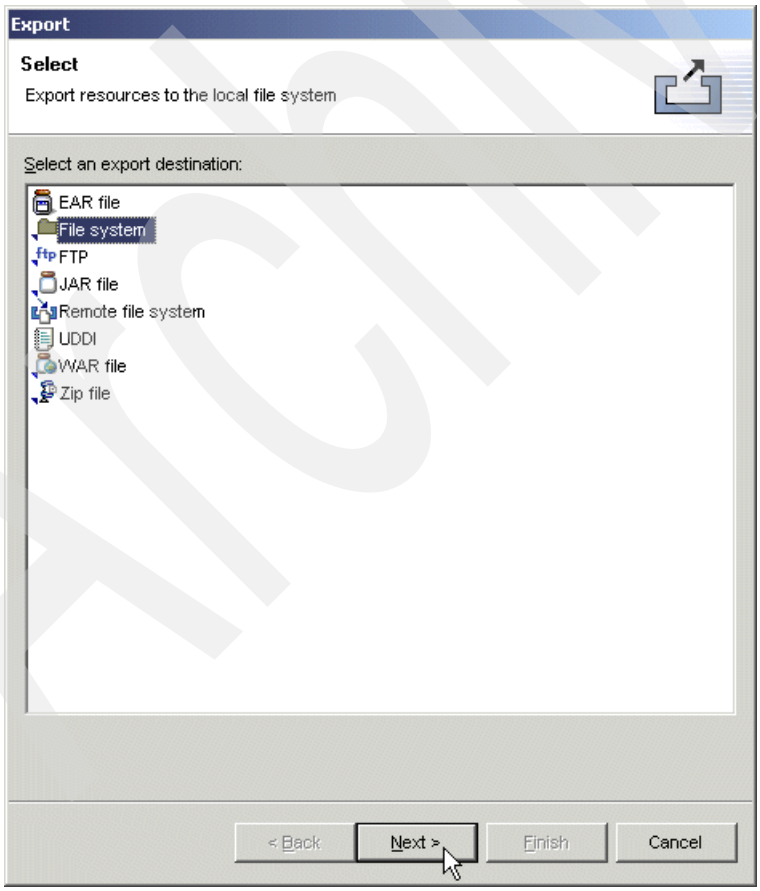


Figure 4-40 Select the destination of the exported file

Archived

By selecting the File System option, we have elected to save the file(s) to a local or network drive.

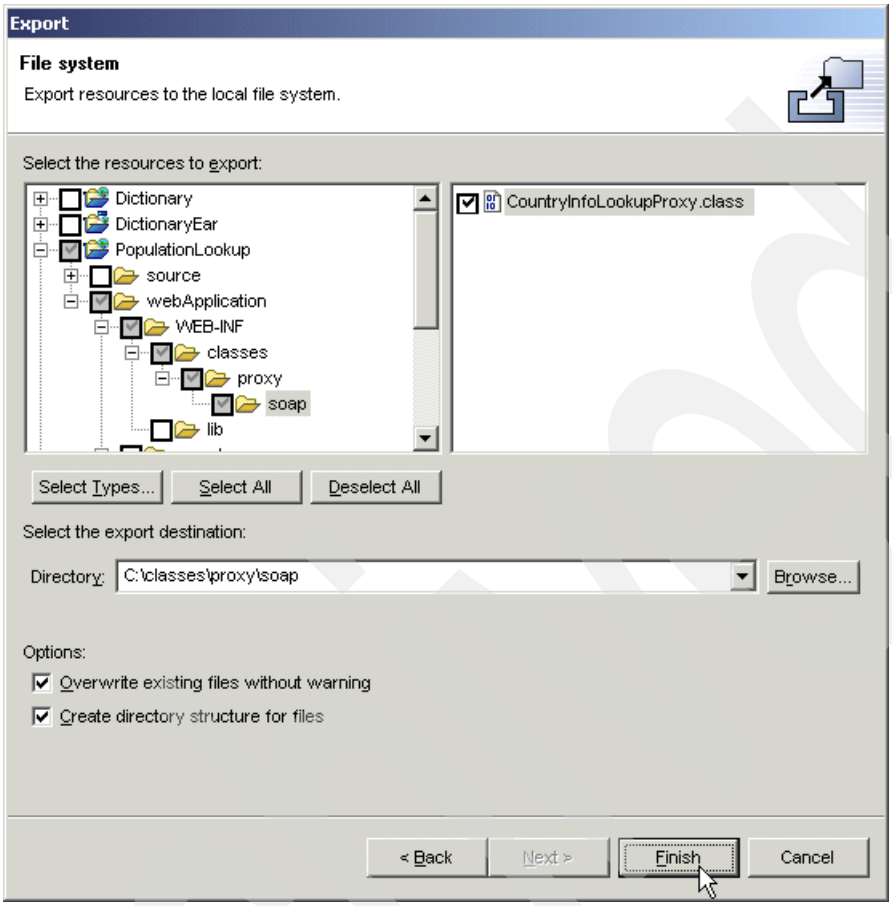


Figure 4-41 Select the source and destination of the file(s) exported

As you can see in Figure 4-41, the CountryInfoLookupProxy.class should have already been selected for us. If not, we want to select the directory structure shown above, and place a check in the box next to the file. Next you will need to select a destination for this .class file. We chose to place this file in c:\classes, because this .class is a package, under proxy\soap, we must specify this in the destination directory field:

```
c:\classes\proxy\soap
```

Also, you will want to assure the options **Overwrite existing files without warning** and **Create directory structure for files** are both checked because we cannot assume the directory structure is already there.

In the next section, we will discuss how and where you can use the CountryInfoLookupProxy.class file to simplify your consumer Domino application.

4.5.4 Using your new proxy client

In the previous section you learned how to create a proxy client for your Web service consumer. This client is merely a Java class that creates a SOAP message from the parameters we pass to it, and sends and receives this SOAP message to and from the server. As you have probably figured out, we are going to utilize this new class in a Java agent within our Domino application.

Creating your first Java consumer agent

In this section you need to understand the basics of Java programming in Domino Designer. We walk through, step-by-step, what you will need to create your first Java Web Service consumer agent. As with the other examples in this chapter, in this example we used the **LotusScript -> Java Agent -> Web Service** approach.

1. First we need to open Domino Designer. Once you have Designer open, you need to open the database you want to create this agent in. In our case, we will create our new agent in SoapRunner.nsf.
2. Once you have this database open, select the agents section of the design elements and click **New Agent** as shown in Figure 4-42.

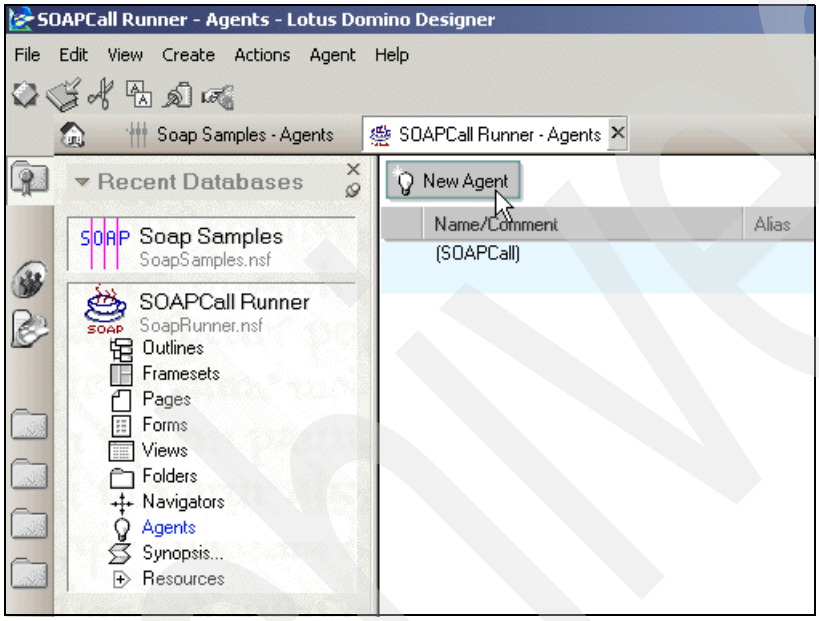


Figure 4-42 Create New Agent in Domino Designer

3. After you have clicked on New Agent, your Domino Designer client should look like Figure 4-43. You will notice all of the options we have selected at the top of the pane on the right-hand side.
   a. Name: (SOAPCountryPop)
   b. Shared agent: Checked
   c. When should this agent run? — Run manually from actions menu
   d. Which document(s) should it act on? — Run Once (@Command may be used)
   e. Run: Java

   By selecting these options, we have chosen the name of the agent to be (SOAPCountryPop) for Country Population consumer.

Tip: If you ever want to *hide* a design element from the actual end user, you can create your element, and place (parenthesis) around its name when saving. This will hide the element so it is not viewable from within the Notes Client.

We want to *share* this agent with all users that have access to this database. If you did not check this box, only your Notes ID would be able to run this agent. We also chose to run this agent manually from the Actions menu, instead of scheduled. We want to force this

agent to run, so we select Run Once. The other options in this drop-down box are for triggering your agent to run after such events as *When new mail arrives* and so on. Finally we want to select that this agent will be written in Java.

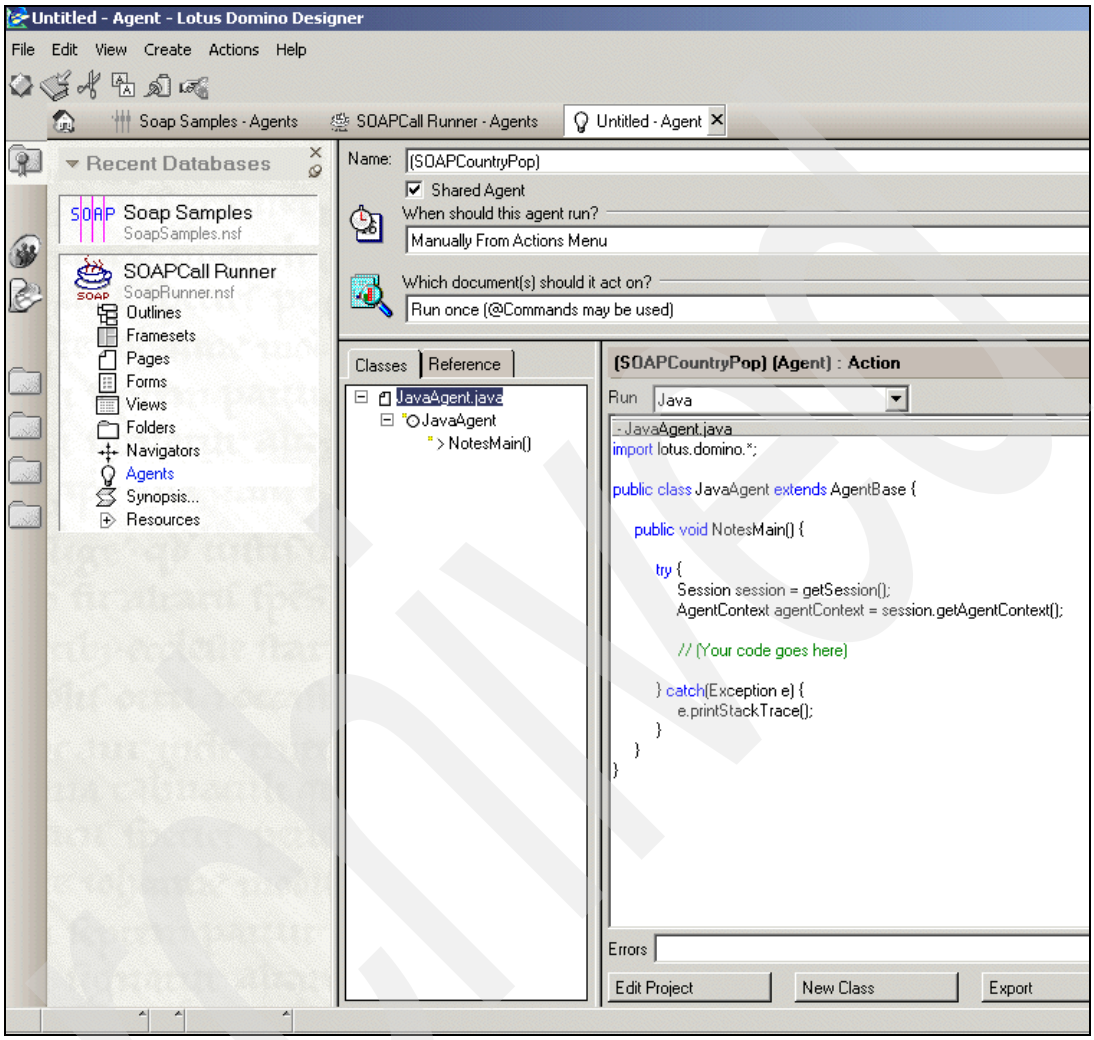


Figure 4-43 Creating a Java agent in Domino Designer

4. Now that we have the agent template ready for us to start developing, we need to figure out what non-Domino classes or JARs we will be using in this agent. The reason we need to know these files is because we need to specify to the Java agent what these files are and where they are located. This process is known as importing, in Java.

   The first and easiest way is by directly importing the file (.class or .jar) into your Java project within Domino Designer. If you have read through this redbook from cover to cover, you will notice we have covered this in “Customizing GetStockQuote Java agent” on page 119.

   This can be done by clicking on the **Edit Project** button, located near the bottom-center of your Domino Designer client. This will actually *import* the file into your Java project, so the agent will be able to use all of its objects. After you have checked the Edit Project button a window will pop up for you to select what files you will be importing. You can see this in Figure 4-44.

   In the window you will see the files: activation.jar, mail.jar, soap.jar, xerces.jar and xml4.jar. All of the files are required by soap.jar to create a SOAP message. Along with these files

you will see a **proxy\soap\CountryInfoLookupProxy.class**. This is the class that we have created using WDSc. Take note that the file has the directory structure listed in the right-hand pane. **THIS IS REQUIRED.** In this example we have created a directory **c:\lotus\notes\proxy\soap** to hold this file.

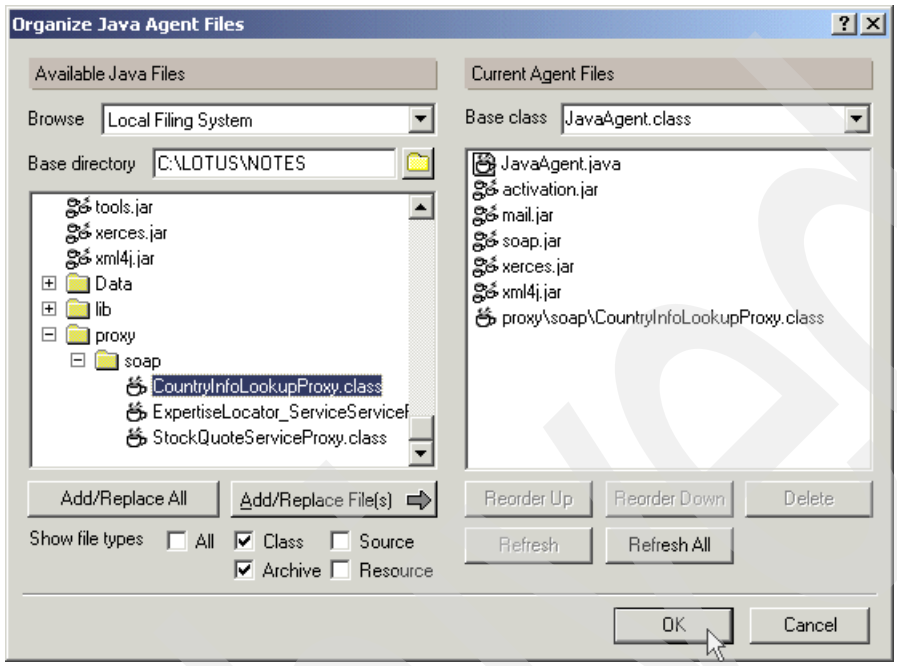


Figure 4-44 Importing Java class files into project

Importing files this way makes your Java agent *very* inefficient. Now you are asking, how can I make it more efficient? There is a better way to do this, but for most developers this method can be nearly impossible. We can specify the required classes and JARs directly in the notes.ini. So you're asking, why is that impossible? It is not, if you only want to run this agent on your Notes client, since you do have access to your own notes.ini. But, if you want to run this agent on the server, or as a scheduled agent, these files will need to be specified to the notes.ini of the Domino server. In most cases, this will be hard to get your Domino Administrators to do this for you.

To do this method we have created a directory in our local PC or in IFS on the iSeries called **\classes**. It is a good idea to keep your directory structure short because the ini parameter has a limitation to its length. The ini parameter should follow the form as the example shown below.

```
JavaUserClasses=c:\classes;c:\classes\soap.jar; ........ (and so on)
```

When you want to specify class files, you only need to specify the root of the package directory. As an example, our class file has the following path:

c:\classes\proxy\soap\CountryInfoLookupProxy.class

In the JavaUserClasses notes.ini parameter, we only need to specify c:\classes. It will be smart enough to look through the proxy\soap directories to find the class we require. For this setting to take effect, you will need to restart the Notes Client or Domino Server, wherever you added the new ini parameter.

5. Now that we have all of the Java files imported into our project, we are ready to start developing our agent. The structure our agent is going to follow is:

   a. Find the NotesDocument that contains its parameters.

   b. Read off all the parameters.

c. Create an instance of the proxy client, and access the Web Service.

d. Process the return value of the Web Service.

We will not detail, line by line, the straight Notes programming portion of this agent. If you have any problems, the Domino Designer has excellent help files for your use. You can simply access the help by pressing your **F1** key.

Example 4-11 (SOAPCountryPop) Java agent

```
import lotus.domino.*;
import java.util.*;
import java.net.*;
import proxy.soap.*;

public class JavaAgent extends AgentBase {
   public void NotesMain() {
          Document doc = null;
      try {
         Session session = getSession();
         AgentContext ac = session.getAgentContext();
         Database db = ac.getCurrentDatabase();
         Agent agent = ac.getCurrentAgent();

         // Retreive Arg from Agent Data document
         doc = db.getDocumentByID(agent.getParameterDocID());
         Vector params = doc.getItemValue("Country");
         String country= (String) params.elementAt(0);

         // Create Proxy Instance from CountryInfoPopulationProxy Class
         //    class generated and downloaded from WDSc
         proxy.soap.CountryInfoPopulationProxy proxy =
            new proxy.soap.CountryInfoPopulationProxy();
         java.lang.String result = proxy.getPopulation(country);

         doc.replaceItemValue("population", result);
         doc.save();
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

You may think this agent looks very similar to the other examples in this chapter, and you are right. The only real difference is what proxy your agent utilizes. That is all that is required. If you make your Java SOAP agent smart enough to utilize several different proxy classes, you can reuse the same forms, LotusScript agent, and Java agents for consuming multiple Web Services.

From this point on, all that you are required to do is create some sort of user-interface to your consumer. You can integrate this into an existing form you may have in your Domino application, you may add this as a new function to an existing Agent within your Domino application, or maybe add it to the “Welcome Page” of your NotesClient. The choice is up to you.

Archived

A

Creating your own Web Service application

This appendix describes how to create your own Web Service application. Here we choose to describe the InventoryAvailabilityService Web Service application. You will develop a Java class that provides the core functionality of accessing data from DB2/400 and then create a Web Service out of this Java class.

This appendix describes:

- Business benefits:

  How to provide commonly required portal type services without writing a lot of Java code.

  - You are running Manufacturing company and you want your vendors to know availability of Inventory Items.
- Technology and product components:
  - Web Service standards
    - SOAP
    - WSDL
    - XML
  - Tools for creation of a Web Service
    - WDSc
    - WAS

Archived

© Copyright IBM Corp. 2002. All rights reserved.

Scenario description

VCK Inc. is a fictitious company we created for this redbook project. VCK Inc. is a Manufacturing company which wants to extend its Web presence by creating a Web Service for an online check of Inventory Availability.

Business view

Let's say, for example, a local vendor company called Rochester Food and Manufacturing, another fictitious company, needs to know the inventory status from its supplier VCK Inc. Rochester Food and Manufacturing does not want to build and maintain their own application for providing this service. So they asked VCK Inc. to create a Web Service for them. The Java Developer at VCK Inc. wrote a Java class to access the inventory database from DB2/400. He then used WDSc to create a Web Service out of the Java class.

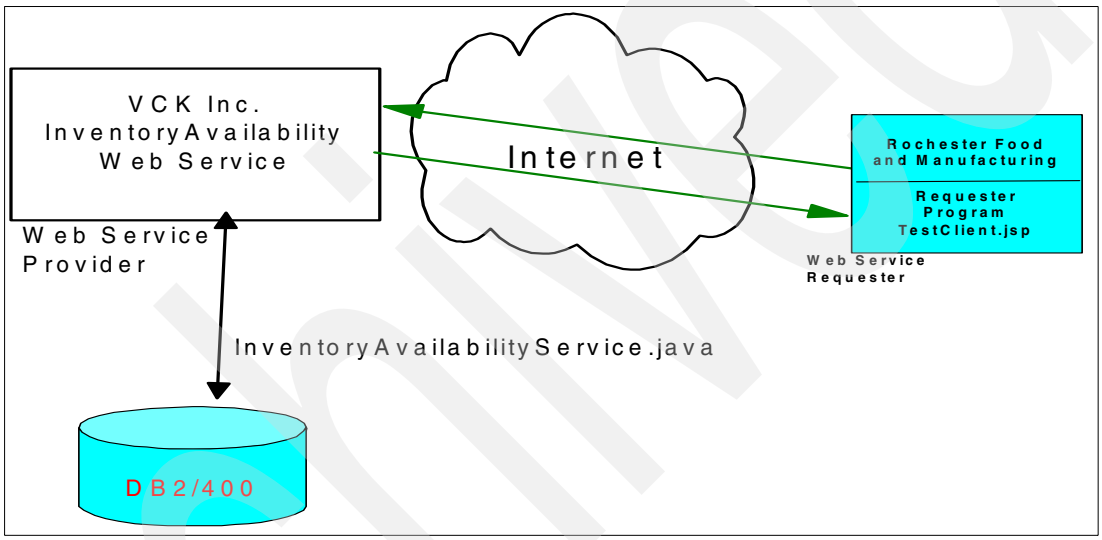


Figure A-1 Business view of Inventory Availability Web Service

Figure A-1 illustrates the business view of the StockQuoteApp Web Service application.

1. The Web Service requester, Rochester Food and Manufacturing, clicks on the 'InventoryAvailability' portlet to check on the current inventory status of any particular part number. This will be formatted as a Web Service request and sent to VCK Inc's Web Service server.
2. VCK Inc's Web Service provider program will send the request to its Java class for database access. The return data from Java class is packaged as a SOAP response to send to the requesting portlet.

Technology view

Let's map this business view with the technology view.

Archive

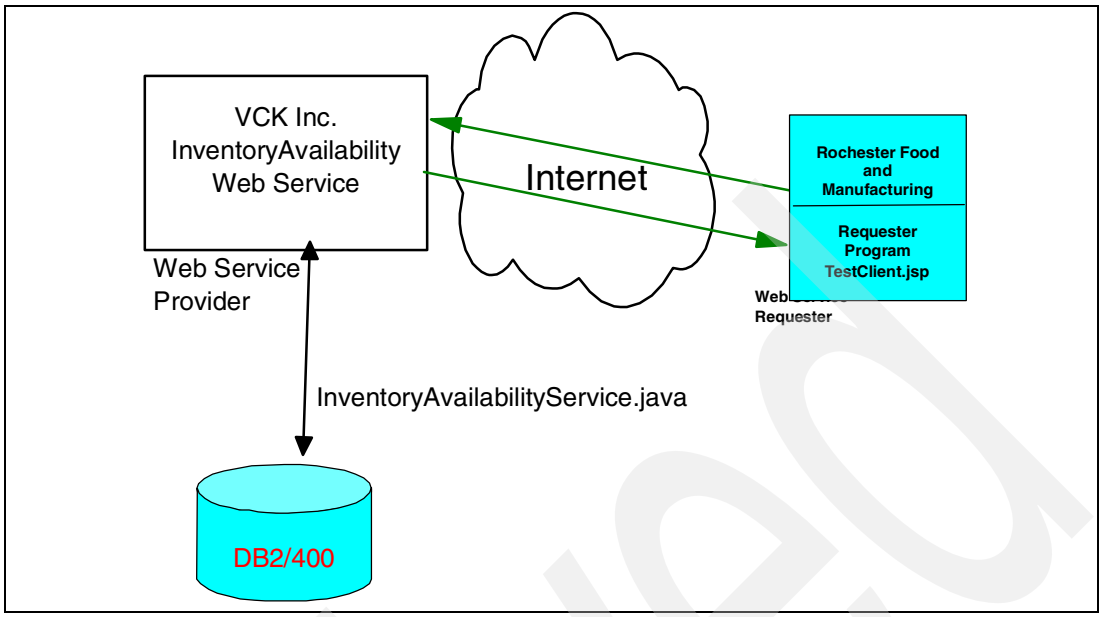


Figure A-2 Technology View of InvecntoryAvailability Web Service

To create database access to DB2/400 for an inventory check we need to have a Java class as shown in Figure A-3 and Figure A-4. This Java class performs a jdbc access using the AS/400 toolbox for Java JDBC driver. You specify your iSeries system name, username and password to obtain a JDBC connection. Here we already created a physical file on an iSeries machine as INVAVLPF, which is in the source library IWEBSVCLAB. We use a prepared statement to execute a query and then iterate through the result set to read the availability of a certain part number.

Tip: For faster local database accesses which involve a JDBC connection from within the iSeries machine to the DB2/400 database, you can use a native driver class: `com.ibm.db2.jdbc.app.DB2Driver`.

Archive

```
package com.vck.rst;


import java.util.*;
import javax.sql.*;
import java.sql.*;

public class InventoryAvailabilityService {

public String getInventoryAvailability(String partNo){

       Connection connection   = null;
       String system = "as06";
       String userName = "vinit";
       String password = "vinitpwd";
       String collectionName = "IWEBSVCLAB";

       ResultSet resultSet =null;
       String quantityAvailable = null;
       String byDate = null;
       String availability = null;


        try {

        //


           // This will register the driver with DriverManager.


         Class.forName("com.ibm.as400.access.AS400JDBCDriver");

            // Get a connection to the database.
            System.err.println("\nabout to get connection ");
            connection = DriverManager.getConnection ("jdbc:as400://"
                + system+"/"+collectionName,userName,password);

      System.err.println("\nconnection obtained "+connection);

         // Prepare a statement for inserting rows.  Since we
            // execute this multiple times, it is best to use a
            // PreparedStatement and parameter markers.
            PreparedStatement invAvlpStmt = connection.prepareStatement ("SELECT
INVQNTY,INVBKDTE FROM INVAVLPF WHERE INVPRTNO=? ");

              //set the prepared statement for part No
              invAvlpStmt.setString(1,partNo);

            //execute the query string
             resultSet = invAvlpStmt.executeQuery();
```

Figure A-3 InventoryAvailabilityService.java (part 1 of 2)

Archived

```
            //this will also advance
            if(resultSet!=null && resultSet.next() ){

            quantityAvailable = resultSet.getString(1);
            byDate = resultSet.getString(2);

            availability = "This item has available quantity "+new
String(quantityAvailable).trim()+" by Date "+byDate;

            }
            else {

            availability = "Item not found in inventory, Please try again";
            }

         if(connection!=null){
         connection.close();
         }


        }//end of try

        catch (Exception e) {

            System.out.println ("Exception in getInventoryAvailability, ERROR: " +
e.getMessage());
          e.printStackTrace();
        }

        finally {

            // Clean up.
            try {
                if (connection != null)
                    connection.close ();
            }
            catch (SQLException e) {
                // Ignore.
            }
        }//end finally

   return availability;

     }//end of method


}//end of class
```

Figure A-4 InventoryAvailabilityService.java (part 1of 2)

Archived

InventoryAvailabilityService creation: step-by-step guide

In this section we take you through the procedure of creating an InventoryAvailability Web Service, step-by-step.

Create a new project

The first thing to create a new Web Service project is to create a Web Project in WDSc.

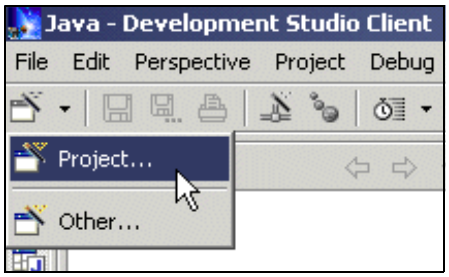


Figure A-5 Create new project in WDSc

1. In WDSc, as illustrated in Figure A-5, click **File -> New -> Project -> Web Project**.

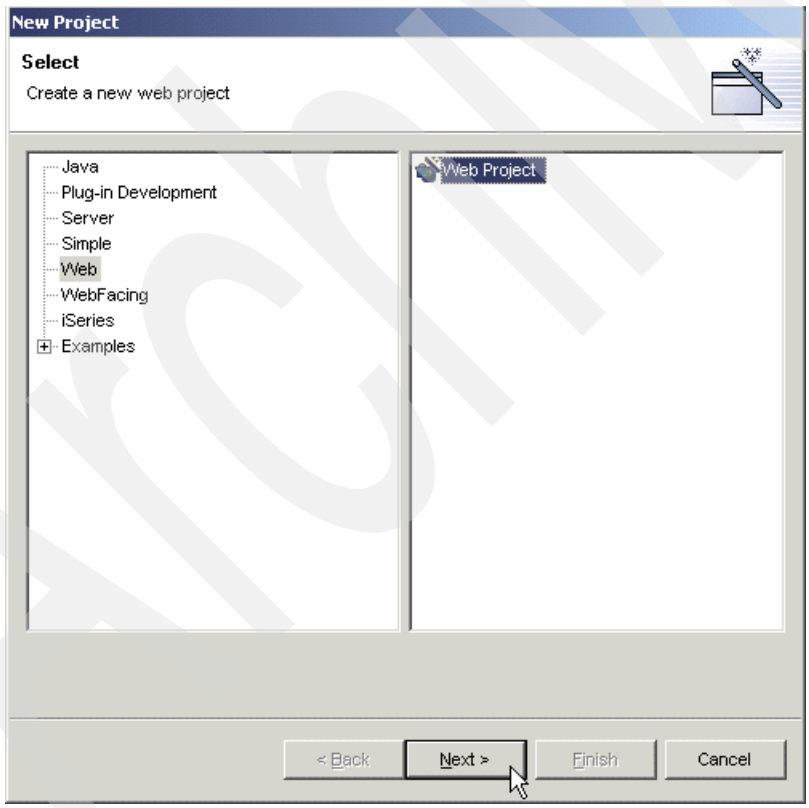


Figure A-6 Select New Project type "Web Project" in WDSc

2. From the selection menu for the new project, select Web from the left pane, then select Web Project from the right pane as shown in Figure A-6.

Archived

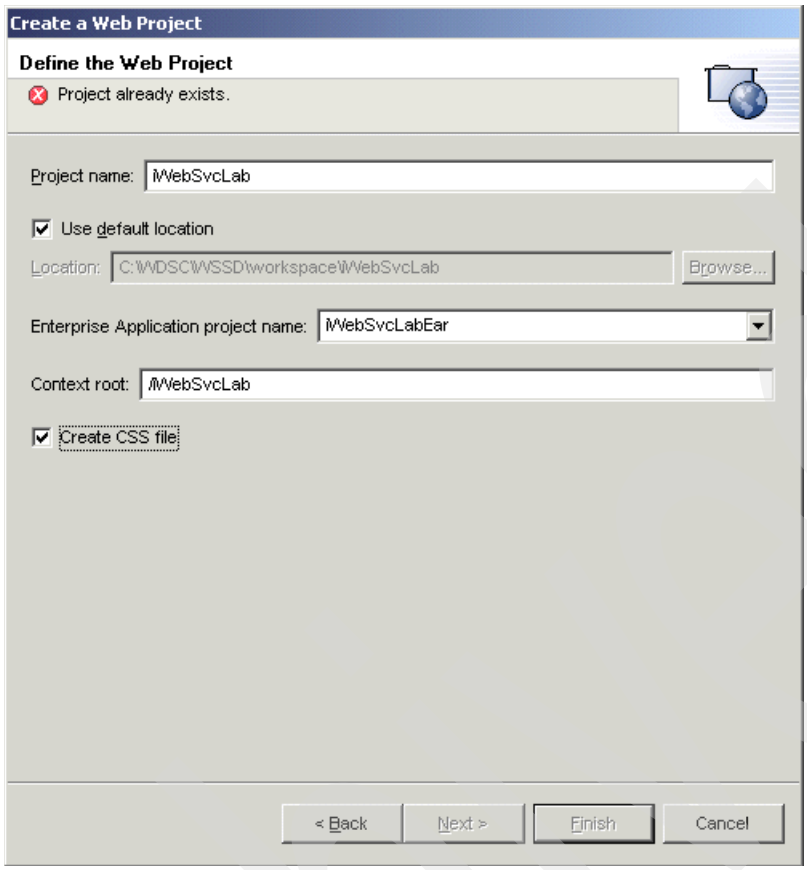


Figure A-7 Specifying project name, context root and EAR project file in WDSc

3. Refer to Figure A-7, for the Web project enter the name of the project, context root, and enterprise application project file.

   Enter information **Project Name > iWebSvcLab**

   Enterprise Application Project Name > iWebSvcLabEar

   Context Root > /iWebSvcLab

   Check the boxes for **use default location** and **create CSS file**

   The context root has to be entered with **"/"** because this will map to the directory path in your Web service when you deploy it on the WebSphere Application Server. The context root will make up the URL of the Web Service.

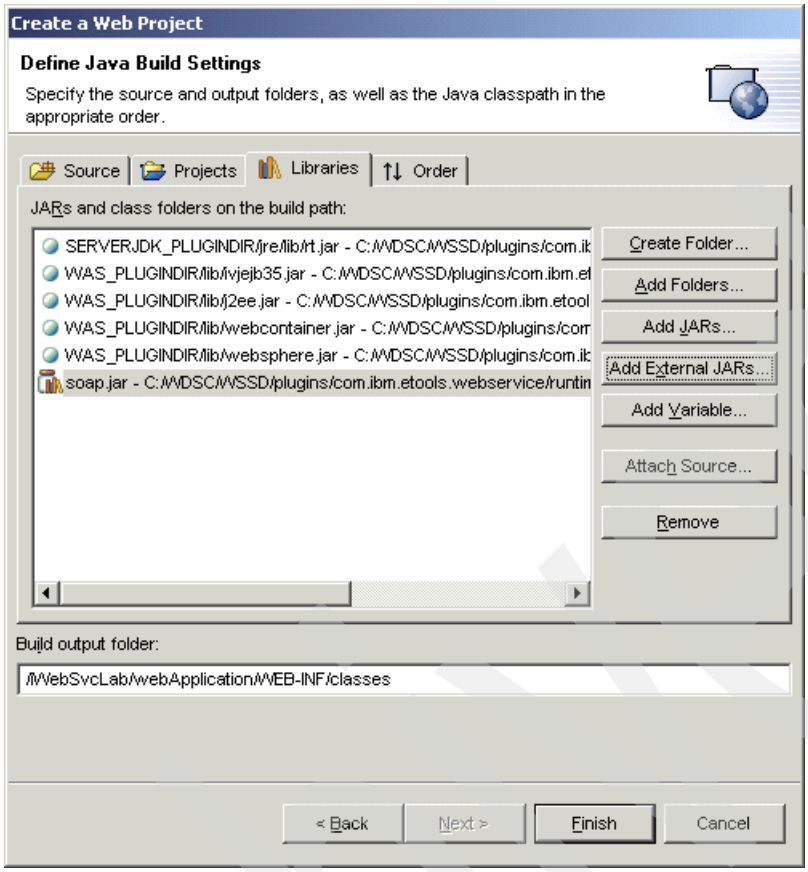


Figure A-8 Adding External Jars to Web Project classpath in WDSc

4. To add dependant jar files for classes, click **Libraries -> Add External JARs** as shown in Figure A-8.

Archive

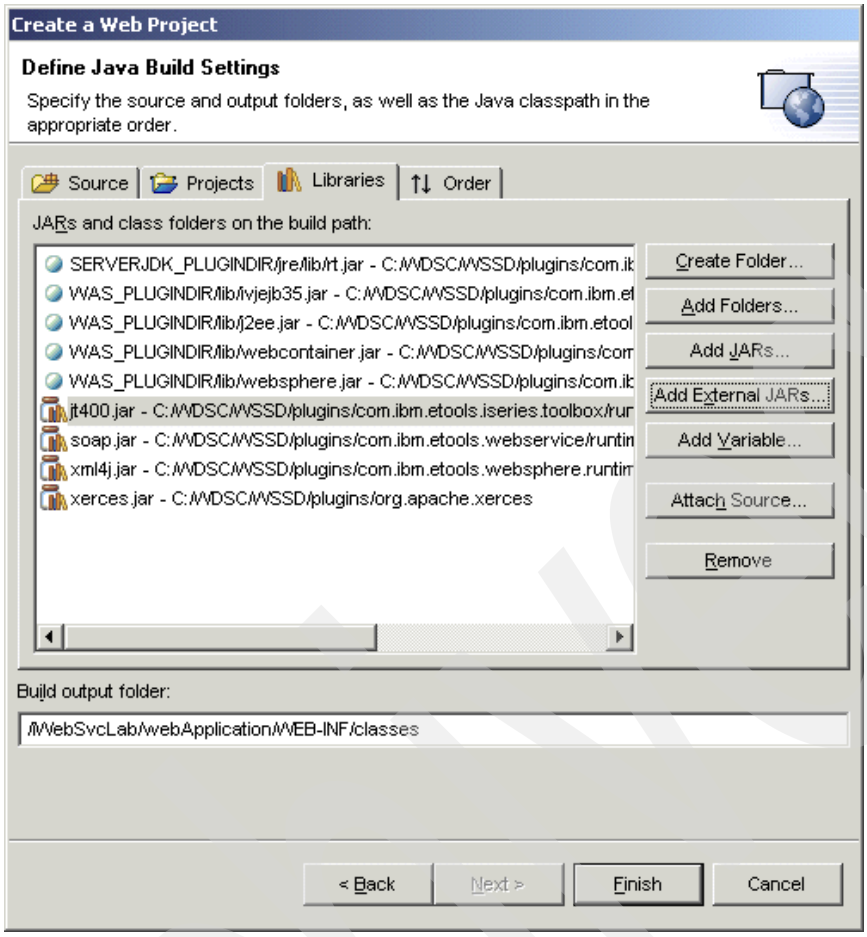


Figure A-9 Add jt400.jar file from WDSc plug-ins directory

5. Refer to Figure A-9 and select:
   C:\WDSc\WSSD\Plugins\com.ibm.etools.iseries.toolbox\runtime\jt400.jar
   to be added as external jar in the iWebSvcLab Project. You should likewise import other jar files such as **xml4j.jar** and **soap.jar** files. These jar files are to be found on your **WDSc\WSSD\Plugin** directory. You can browse the directory as shown in Figure A-10. Click **Finish** to create the Web project.

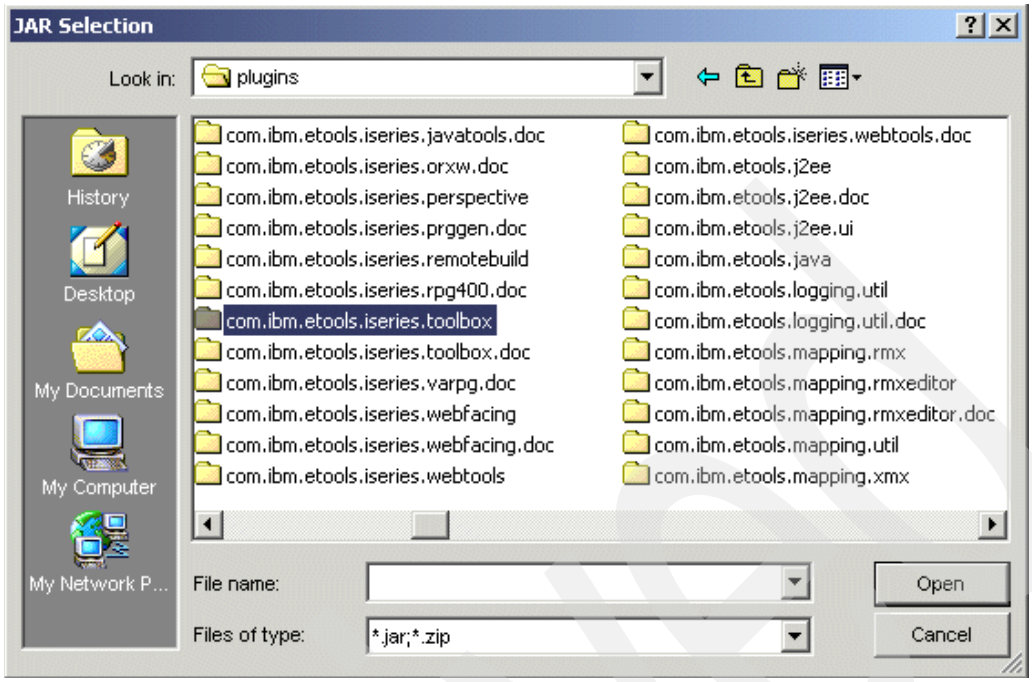


Figure A-10 Plug-in Directory in WDSc to add external jars in WDSc

Create InventoryAvailabilityService class in iWebLabSvc Project

The next task is creating a Web Service provider Java class in the Web Project that we just created in the previous task.

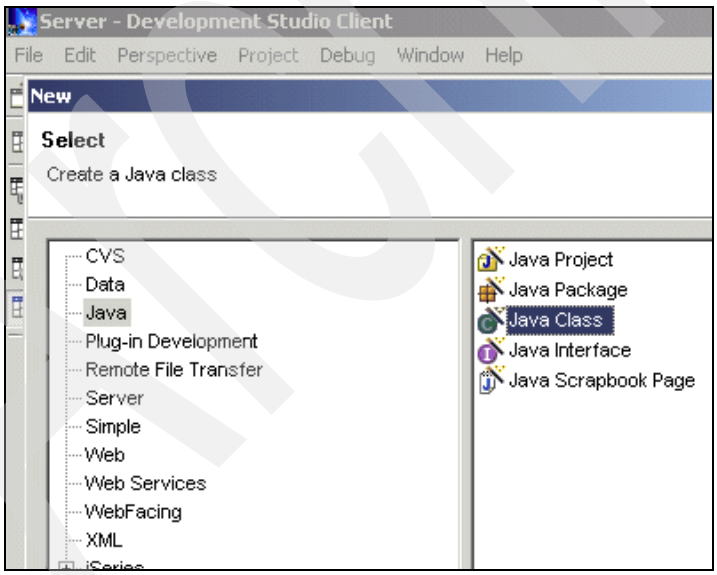


Figure A-11 Selecting Java Class to create InventoryAvailabilityService class

1. On your WDSc main window, click **File -> New -> Other -> Java -> Java Class** as shown in Figure A-11.

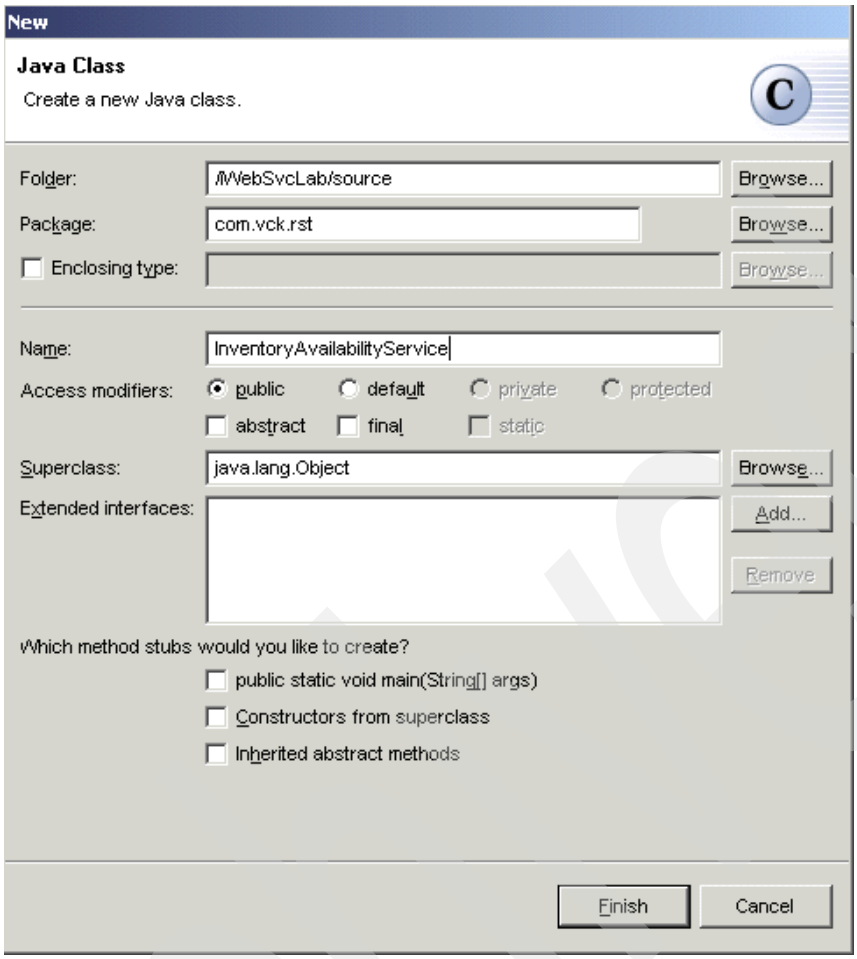


Figure A-12 Creating InventoryAvailabilitySerivce class in WDSc

2. Create a class in WDSc as shown in Figure A-12. Enter **InventoryAvailabilityService** as the class name, and **com.vck.rst** as the package name, and click Finish. Copy the code from Figure A-3 on page 150 and Figure A-4 on page 151 in WDSc com.vck.rst.InventoryAvailabilityService.java file and compile the code by pressing **ctrl -> save**.

Create a new Web Service

The next task is creating a new Web Service in WDSc.

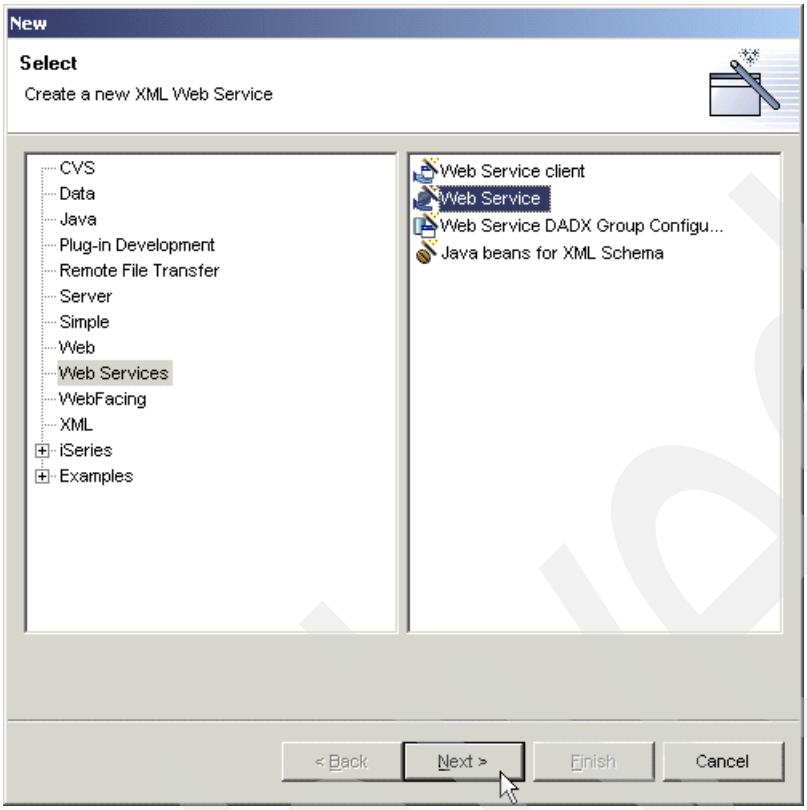


Figure A-13 Create Web Service from StockQuoteService.java in WDSc

1. Refer to Figure A-13. Click **File > New > Other > Web Services > Web Service** to create a Web service. Click **Next** to go to the Web Service options page.

Archive

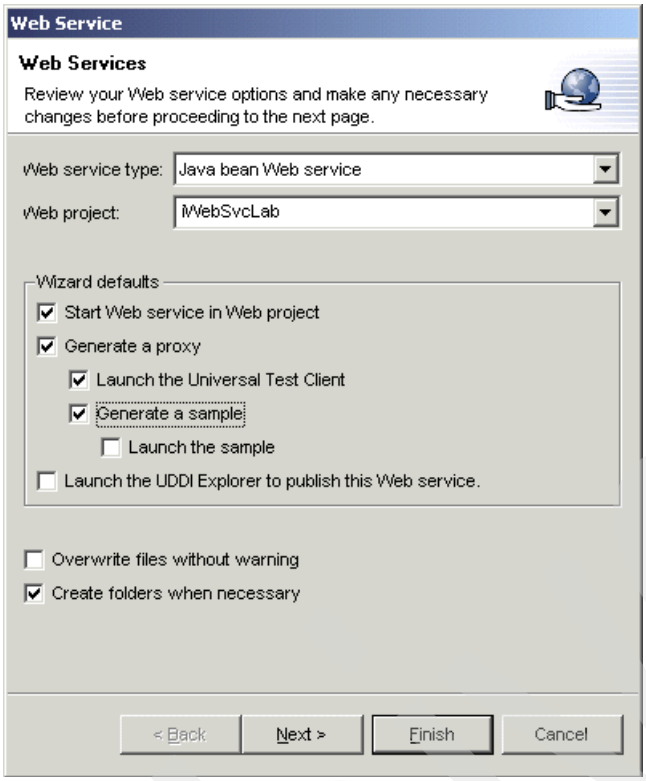


Figure A-14 Web Service Create options in WdSc

2. As given in Figure A-14, for the Web Service create options page, make sure **Web service type** is **Java Bean Web Service**. Also the Web project name should be correct. Check on options **Wizard defaults** - **Start Web service in Web project**, **Generate Proxy**, **Launch Universal Test Client**, **Generate the sample**. Also check on **Create folders when necessary**. Leave the other options unchecked and click **Next**.

Archived

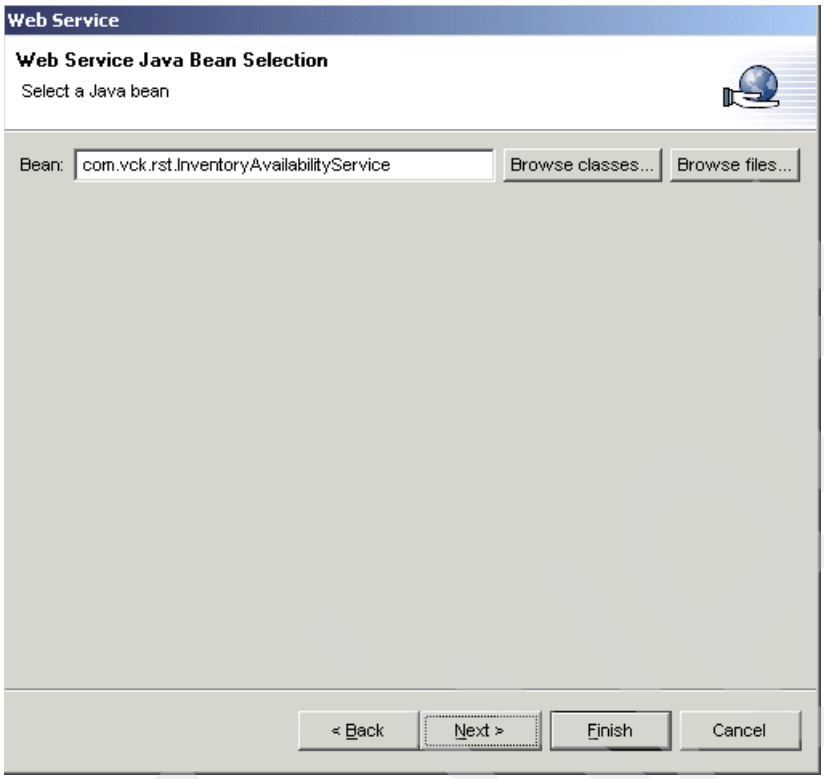


Figure A-15 Select Java bean on which Web Service will be built in WDSc

3. In our InventoryAvailability Web Service example, the Java bean com.vck.rst.InventoryAvailabilityService will encapsulate the core functionality of the Web Service. Refer to Figure A-15, click **Browser Classes** to make a selection.

Archive

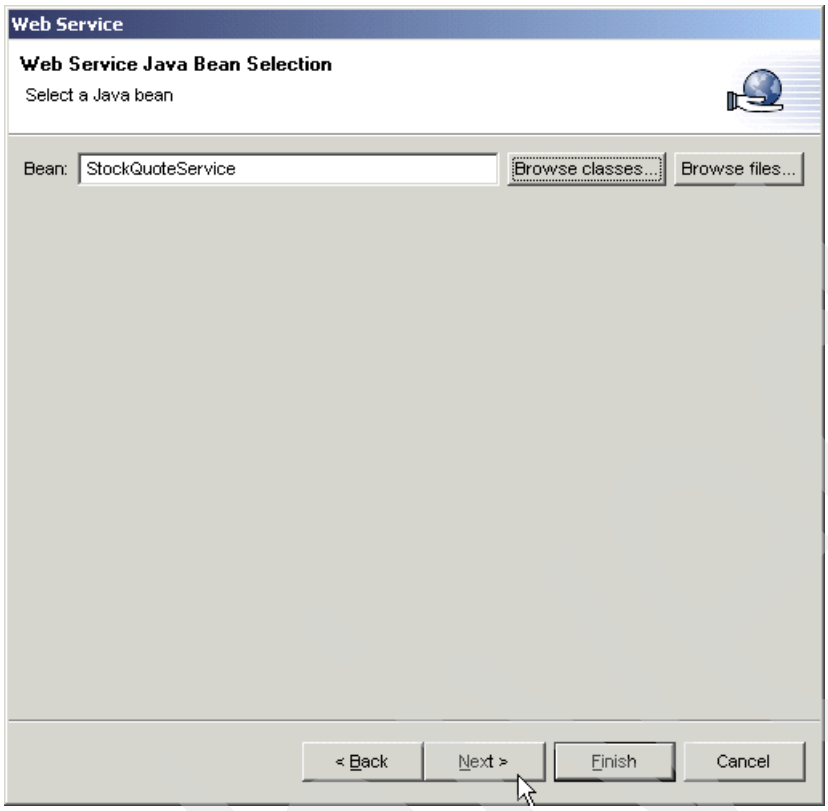


Figure A-16 Select StockQuoteService after selection from Java Bean browser in WDSc

4. As shown in Figure A-16, after the selection from the class browser you will see it appear as a Bean name on the option. Click **Next**.

Archive

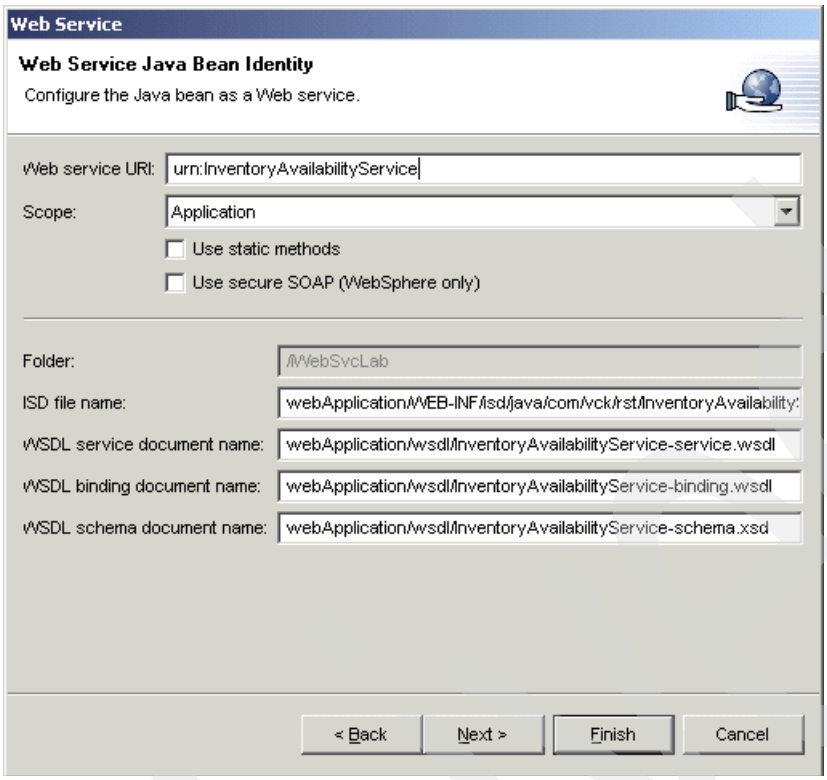


Figure A-17 Specify Web Service Java Bean identity on WDSc

5. As depicted in Figure A-17, on the option window of the Web Service Java Bean Identity, select a name for the Web Service URI. Enter the **urn:InventoryAvailabilityService** as the Web Service URI. The URI, or Uniform Resource Identifier identifies the Web Service to the configuration files. Leave these options as defaults and click **Next**.

Archive

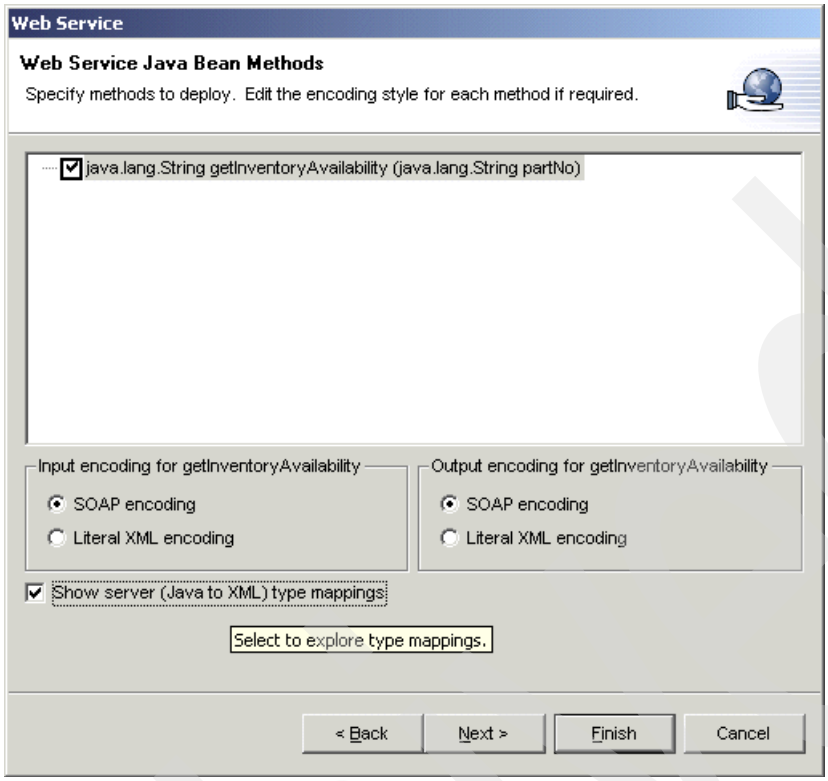


Figure A-18 Specify encoding for Web Service Java Bean Methods in WDSc

6. Refer to Figure A-18. You will see methods of your Web Service Java Bean appear in the selection. Make sure that **Input encoding for getInventoryAvailability** method and **Output encoding for getInventoryAvailability** method are both checked for SOAP encoding. This is because we are building the Web Service on the SOAP encoding envelope. Check **Show Server (Java to XML) type mappings** and click **Next**.

Archive

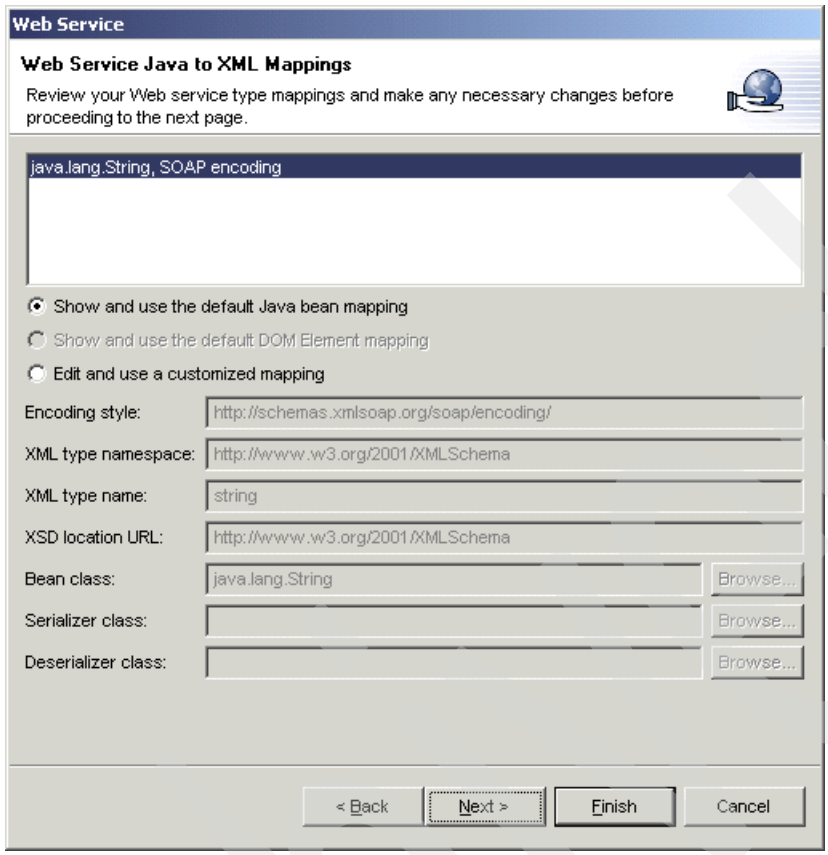


Figure A-19 Verify Java to XML Mappings for Web Service on WDSc

7. Figure A-19 shows the details of InventoryAvailabilityService Java bean method argument and return type with its Web Service encoding style. In this case **java.lang.String,SOAP encoding** depicts the return type and also the call argument type for the Web Service. In this example we have the **Show and use the default Java bean mapping** radio button checked. So leave everything on this option as defaults and click **Next**.

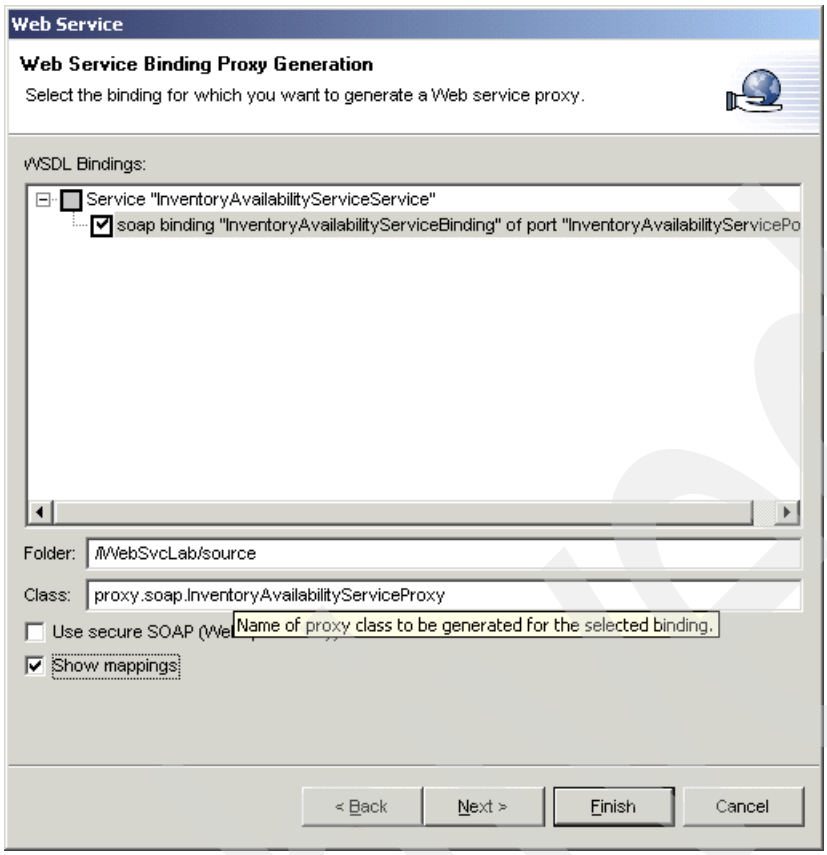


Figure A-20 Verify Web Service Binding Proxy Generation option in WDSc

8. In Figure A-20, you will see the binding type that will be generated for the InventoryAvailability Web Service. This binding will be stored in WSDL files and make the framework of the Web Service. Leave these options as defaults and check the **Show mappings** box, click **Next** to continue.

Archived

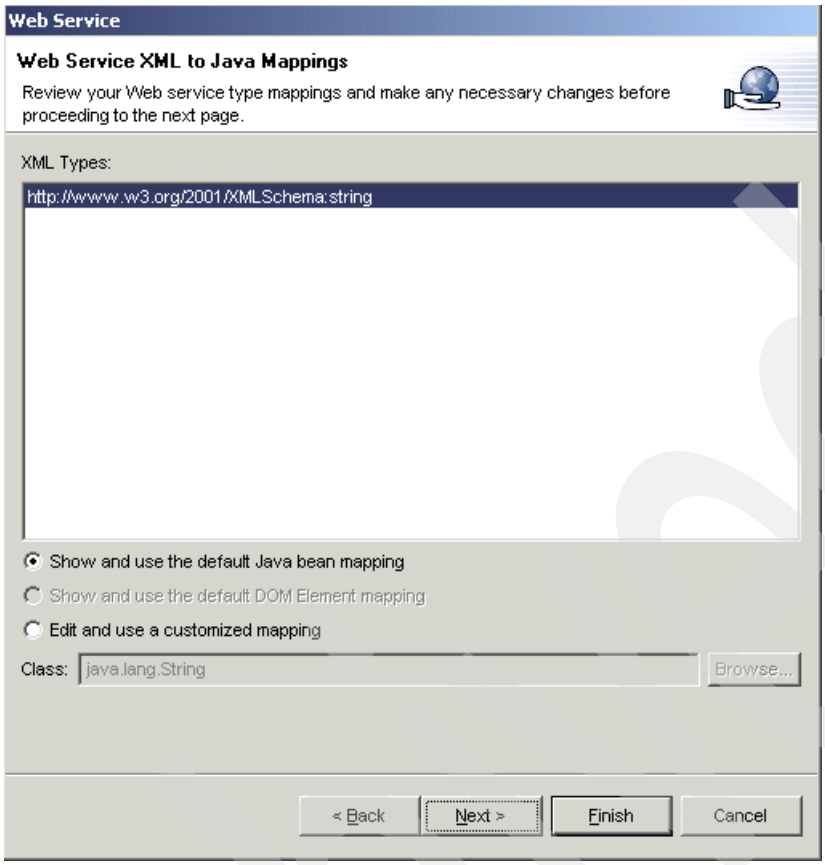


Figure A-21 Verify the input and output parameter Schema mappings for Web Service in WDSc

9. As shown in Figure A-21, WDSc lets you verify whether the XMLSchema mapping and Java Class mapping for a particular input or output parameter is appropriate. Here you get a chance to verify the mappings. In our case, we will leave default options and click **Next**.

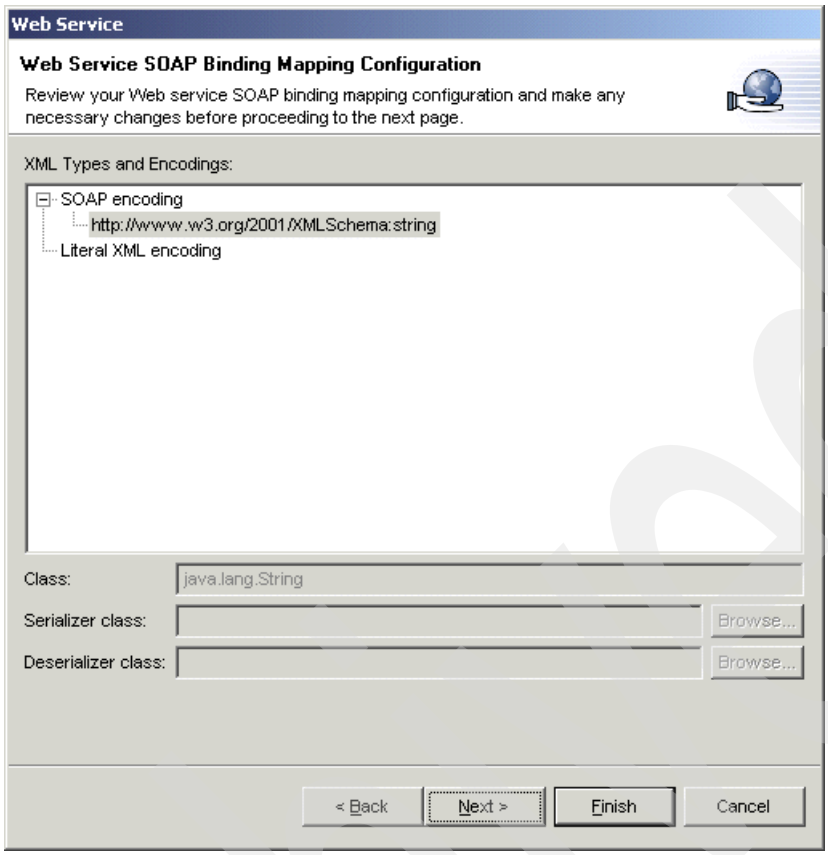


Figure A-22 Web Service SOAP Binding configuration in WDSc

10. Figure A-22 is another figure for cross verifying the configuration with SOAP encoding types for parameters and return types in the Web Service method. In some cases when a request parameter or return parameter are specialized classes, we populate Serializer and Deserializer class fields. Leave everything as it is and click **Next**.

Archived

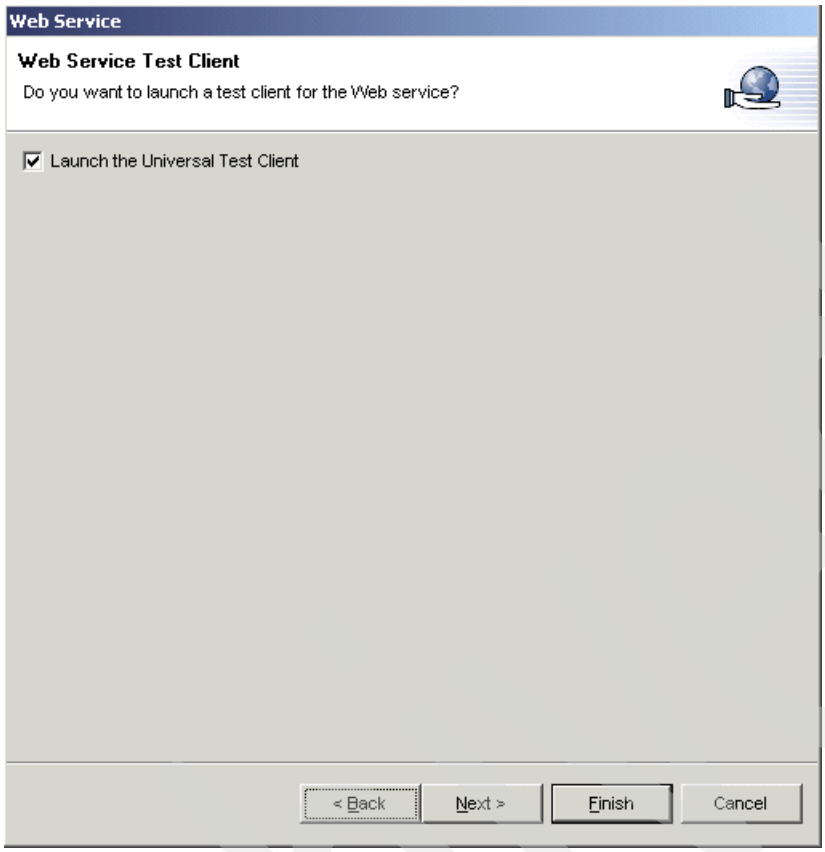


Figure A-23 Test Client create selection window for Web Service on WDSc

11. In Figure A-23, you will have option to create a Web Service test client. This test client is created as TestClient.jsp in WDSc and packaged under the **sample** folder. Select **Launch the Universal Test Client** and click **Next**.

Archive

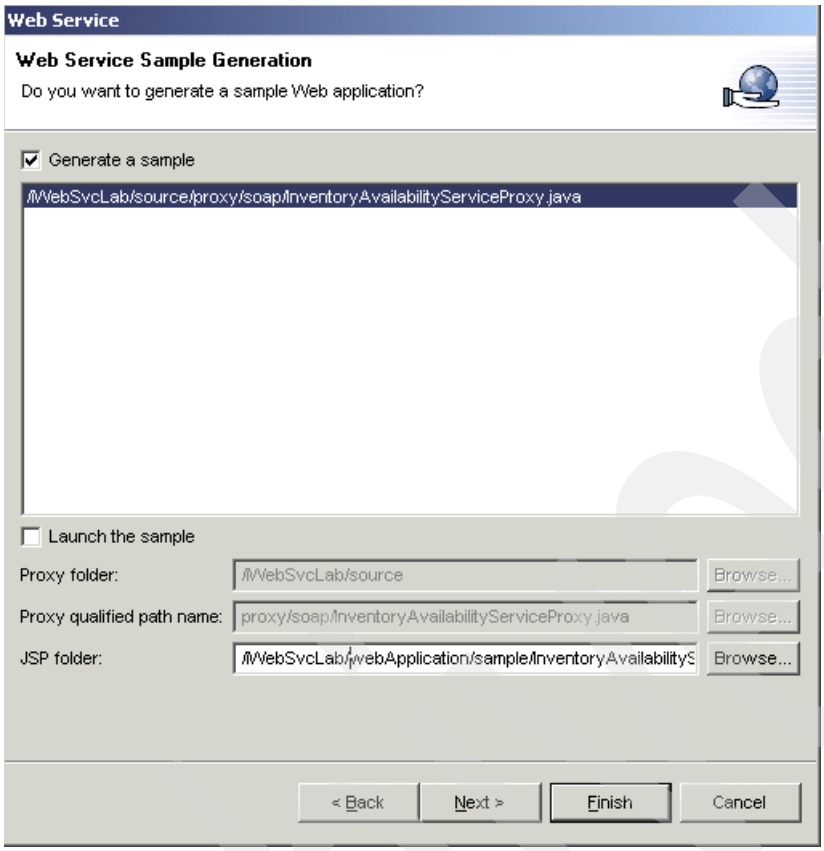


Figure A-24 Web Service Sample Generation option for Web Service in WDSc

12. In Figure A-24, you specify the sample generation option by checking **Generate sample**, the sample TestClient.jsp will be generated in the **JSP folder**, as specified in Figure A-24. Click **Next**.

Archive

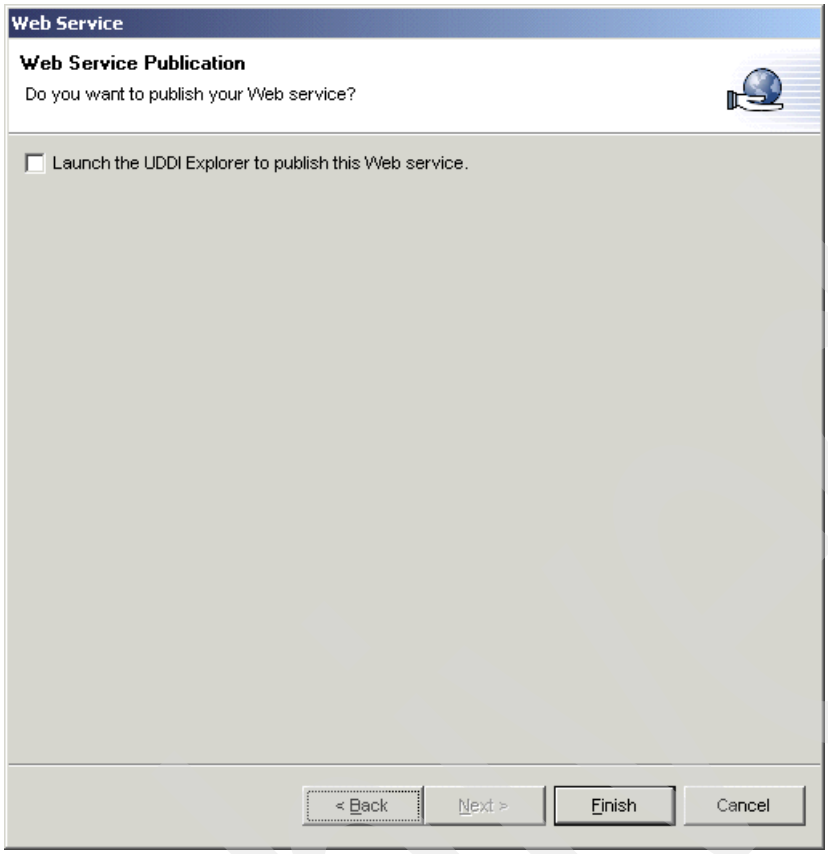


Figure A-25 Web Service Publication option in WDSc

13. In Figure A-25, leave the UDDI publication option unchecked (we will describe UDDI in "Publication of Web Service using UDDI" on page 179). Click **Finish**.

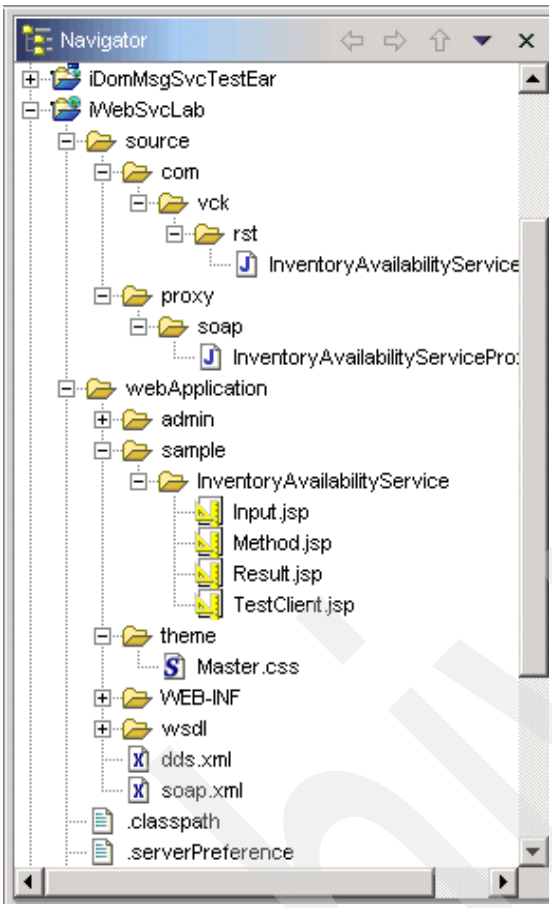


Figure A-26 Directory structure after creation of InventoryAvailability Web Service in WDSc

14. In Figure A-26, you will see the directory structure after the Web Service InventoryAvailability is created. This is the view from the **Web perspective** of WDSc. You will notice that it has a Web Application module structure:
    - The **source** folder has all the Java files. One file which is newly created in the Web Service creation is InventoryAvailabilityProxy.java.
    - The next folder **webApplication** contains all the executable class files and other resource files.
    - The **Admin** folder within **webApplication** contains **index.html**, which gives details on all the running Web Services.
    - The **sample** folder contains the TestClient.jsp file and other jsp resources.
    - **WEB-INF** > **Classes** folder contains all the classes required by the Web Service.
    - **WSDL** folder contains the WSDL XML files which define the framework of the Web Service in terms of Web serving methods, the parameters in these methods, their return types, XML encoding, and so on.

Test the Web Service under WDSc's WebSphere test environment

Before we actually export what we have created to WebSphere Application Server, we can test the application under test environment of WDSc.

Approved

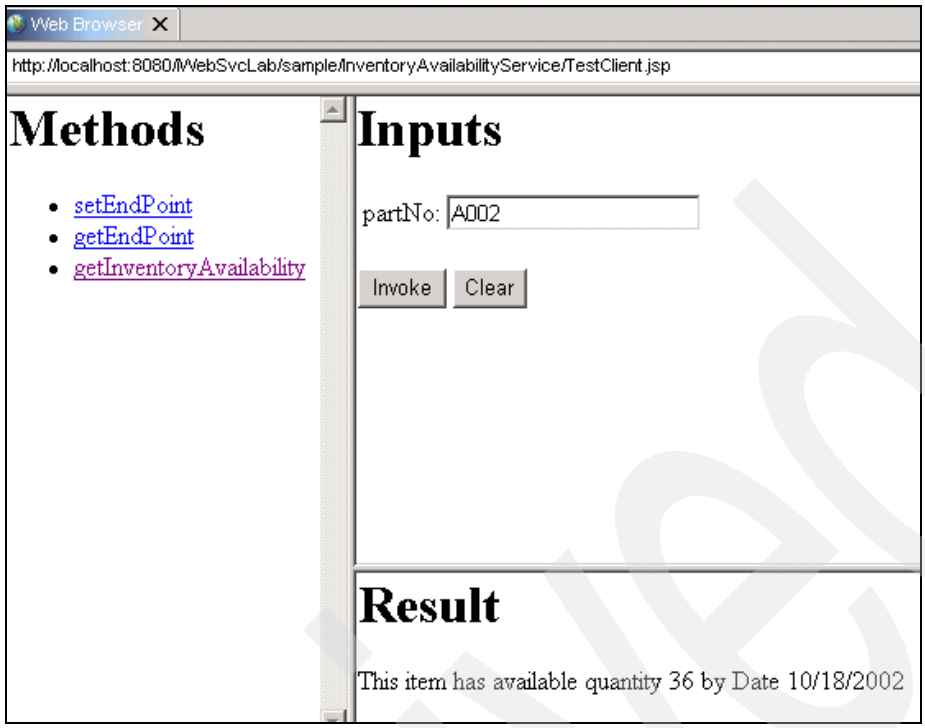


Figure A-27 Launch the TestClietn.jsp WebSphere test environment on WDSc

In Figure A-27, you will see how to make use of WebSphere Test Environment in WDSc to test your first Web Service Application. After creation of the Web Service, WDSc will automatically start the WebSphere Test Environment for you and you will see a window like in Figure A-27. Since this Web Service by default is deployed to **localhost**, you can type in `http://localhost/iWebSvcLab/sample/InventoryAvailabilityService/TestClient.jsp` in the Web Browser pane of WDSc. On accessing the URL you will be provided with all the object references to the proxy client in the Web Service. In our case of the InventoryAvailablity Web Service we want to test the getInventoryAvailablity method using InventoryAvailabilityServiceProxy. Click **getInventoryAvailability** in the references pane and enter the symbol **A002** in the value field of the Parameters pane as a java.lang.String input Parameter. Click **Invoke** to test the Web Service. On successful run of the Web Service you will see a result like `This item has available quantity NN by Date MM/DD/CCYY` in the Parameters pane.

Congratulations! You have successfully created and tested your first Web Service on WDSc and WebSphere Test Environment under WDSc. If you had any trouble on the creation of the Web Service, refer to “Tips and techniques” on page 175 for troubleshooting hints.

The rest of the procedure of creating the Web Service application is identical as described in our earlier example of your first Web Service application, in Chapter 2, “Writing my first Web Service application” on page 9. You can move to 2.2.5, “Setting up iSeries environment for deployment” on page 40. That task is right after you run your application under WDSc’s test environment. Simply change the names of the modules, objects, classes, and so on, appropriately.

References

This section collects reference materials related to the application.

Figure A-28 and Figure A-29 illustrate the proxy code which could be automatically generated by WDSc.

```
package proxy.soap;

import java.net.*;
import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.messaging.*;

public class InventoryAvailabilityServiceProxy
{
  private Call call = new Call();
  private URL url = null;
  private String stringURL = "http://idomws1:5500/iWebSvcLab/servlet/rpcrouter";
  private SOAPMappingRegistry smr = call.getSOAPMappingRegistry();

  public InventoryAvailabilityServiceProxy()
  {
  }

  public synchronized void setEndPoint(URL url)
  {
    this.url = url;
  }

  public synchronized URL getEndPoint() throws MalformedURLException
  {
    return getURL();
  }
```

Figure A-28 InventoryAvailabilityServiceProxy.java (1 of 2)

Archived

```
 private URL getURL() throws MalformedURLException
  {
    if (url == null && stringURL != null && stringURL.length() > 0)
    {
      url = new URL(stringURL);
    }
    return url;
  }

  public synchronized java.lang.String getInventoryAvailability(java.lang.String partNo)
throws Exception
  {
    String targetObjectURI = "urn:InventoryAvailabilityService";
    String SOAPActionURI = "";

    if(getURL() == null)
    {
      throw new SOAPException(Constants.FAULT_CODE_CLIENT,
      "A URL must be specified via
InventoryAvailabilityServiceProxy.setEndPoint(URL).");
    }

    call.setMethodName("getInventoryAvailability");
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setTargetObjectURI(targetObjectURI);
    Vector params = new Vector();
    Parameter partNoParam = new Parameter("partNo", java.lang.String.class, partNo,
Constants.NS_URI_SOAP_ENC);
    params.addElement(partNoParam);
    call.setParams(params);
    Response resp = call.invoke(getURL(), SOAPActionURI);

    //Check the response.
    if (resp.generatedFault())
    {
      Fault fault = resp.getFault();
      call.setFullTargetObjectURI(targetObjectURI);
      throw new SOAPException(fault.getFaultCode(), fault.getFaultString());
    }
    else
    {
      Parameter refValue = resp.getReturnValue();
      return ((java.lang.String)refValue.getValue());
    }
  }

}
```

Figure A-29 InventoryAvailabiliytServieProxy.java (2 of 2)

Figure A-30 illustrates the record definition of a physical file on iSeries, *INVAVLPF*.

Archived

```
Columns . . . :   1  80                              Edit
IWEBSVCLAB/QDDSSRC
 SEU==>
INVAVLPF
FMTPF.....A.........T.Name++++++RLen++TDpB......Functions++++++++++++++++++++++++++++
        *************** Beginning of data
*****************************************************************************************
0001.00 A*
020605
0002.00 A              R INVAVLREC
020605
0003.00 A                INVPRTNO      10A         COLHDG('PART NO.')
020605
0004.00 A                INVQNTY       10A         COLHDG('QUANTITY.')
020605
0005.00 A                INVBKDTE      10A         COLHDG('BACK ORDER DATE')
020605
0006.00 A*
020605
0007.00 A              K INVPRTNO
020605
        ****************** End of data
*****************************************************************************************
**
F3=Exit  F4=Prompt  F5=Refresh  F9=Retrieve  F10=Cursor  F11=Toggle
 F16=Repeat find    F17=Repeat change     F24=More keys
```

Figure A-30 Create INVAVLPF in library IWEBSVCLAB on iSeries

Tips and techniques

This section lists some tips and techniques related to the scenario:

- For troubleshooting and checking purposes, we recommend that the Web Service be tested first in the WebSphere Test Environment of WDSc.
- After deployment of the enterprise application on WebSphere Application Server, make sure that the installation directory path is generated correctly by looking at the /QIBM/UserData./WebAsADv4 directory of WebSphere Application Server.
- On WebSphere Application Server make sure that the appropriate node has a proper Web server plug-in generated against it.
- If you get a page not found error on servlet, jsp or html pages, make sure the context root of the application is properly set. Also check the URL entered in the browser for such an error.
- Always make sure that the Web Service is running by pointing your browser to the SOAP admin page:

  `http://<yourserver>:<yourport>/<yourcontextroot>/admin/index.html`

Archived

Archived

B

Using UDDI for both ends — publishing and requesting

UDDI is the publishing mechanism of Web Services. For Web Service providers, it is a vehicle to advertise their services. For Web Service requesters, it is a vehicle to find the services and archive the information required to create Web Service clients.

This chapter covers:

- How UDDI (Universal Description, Discovery and Integration) helps Web Services in publication and discovery
- How businesses benefit each other with UDDI
- Technical overview regarding UDDI publication
- Step-by-step guide to publish a Web Service using WDSc
- Step-by-step guide to create a Web Service client using WDSc

Note: This example can only be run when you have access to IBM Test Registry UDDI Web site or any other commercially available UDDI Web site.

© Copyright IBM Corp. 2002. All rights reserved.

What is UDDI's role in Web Service?

The Universal Description, Discovery and Integration (UDDI) specifications define a way to publish and discover information about Web Services. A Web Service is the specific business functionality exposed by a company, through the use of an Internet connection, for the purpose to use the service.

As shown in the Figure B-1, a Web Service services the marketplace and provides functionality to business users. The technical users use UDDI registry information to discover a Web Service. Once discovered UDDI provides the technical users interface to create a Web Services consumer client.

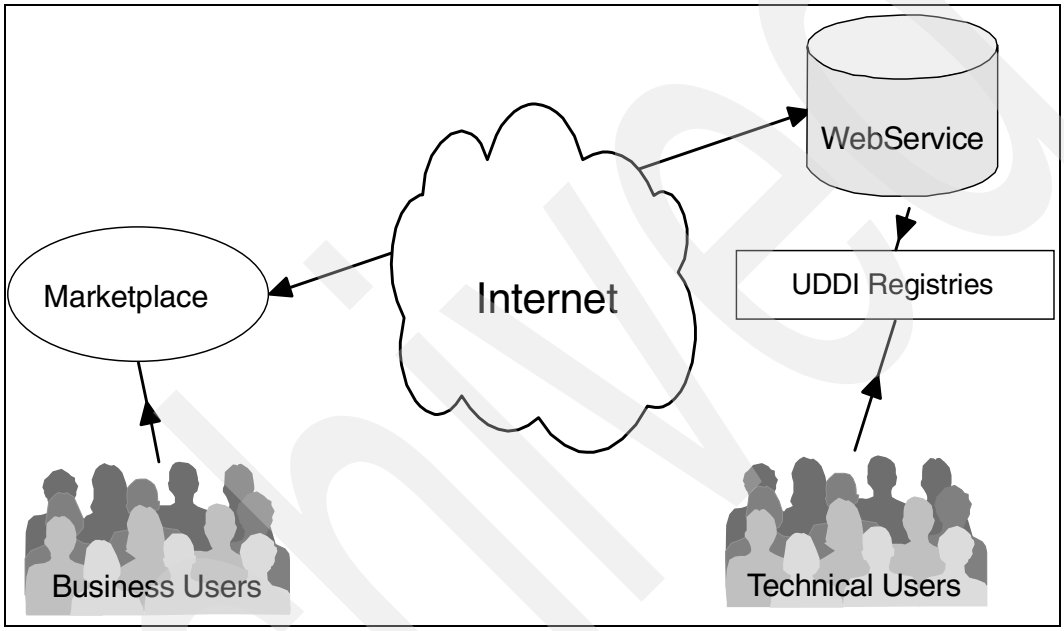


Figure B-1 Web Service registration and discovery using UDDI

Business scenario

Lets say there is company called VCK Inc., which manufactures certain food products and markets in the mid-western United States. The company is headquartered in Rochester. VCK Inc., wants to publish its Inventory Availability Web Service, so that its vendors can see online inventory. VCK Inc. decides to publish the Web Service with the help of IBM iSeries tools.

Technical scenario

For UDDI business registration, VCK Inc. needs a tool to create an XML file used to describe a business entity and its Web Services. The information that will be provided in a UDDI business registration will consist of three components: "white pages" which include address and contact information, "yellow pages" which include business categorizations and "green pages", the technical information about services that are provided by the business.

VCK Inc. decided to have the WDSc for iSeries tool from IBM to publish their Web Service. They also decided that they will publish their Web Service under the Food Manufacturing category under the IBM Test Registry.

In the following section we describe how VCK Inc. published and tested their Inventory Availability Web Service using IBM tools. For more technical information on the UDDI topic refer to the Web site:

http://www.uddi.org

Publication of Web Service using UDDI

This section explains how to publish your Web Service using UDDI. You are a Web Service *provider*. Again, we are using WDSc as development and deployment tool. We will use WDSc to publish our Web Service to UDDI.

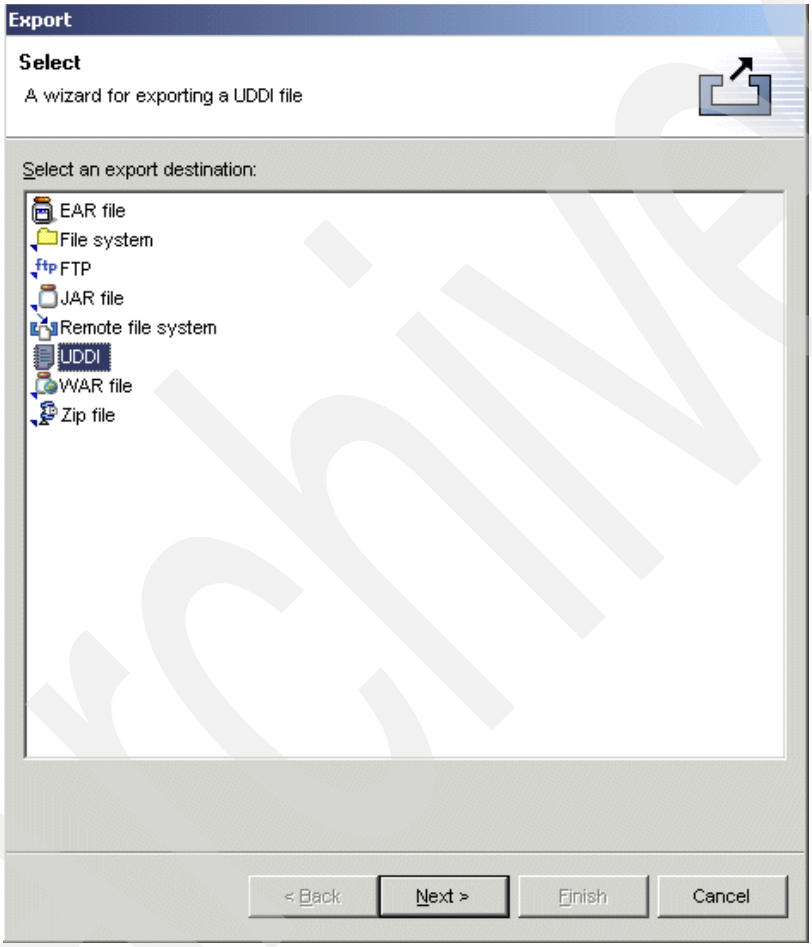


Figure B-2 Export UDDI for publication in WDSc

1. As shown in Figure B-2, select the **File -> Export -> UDDI** steps to export the InventoryAvailability Web Service project for publication.

Archived

Figure B-3 UDDI Export window on WDSc

2. Click **Finish** on the UDDI export window as shown in Figure B-3, to start the UDDI explorer.

Archive

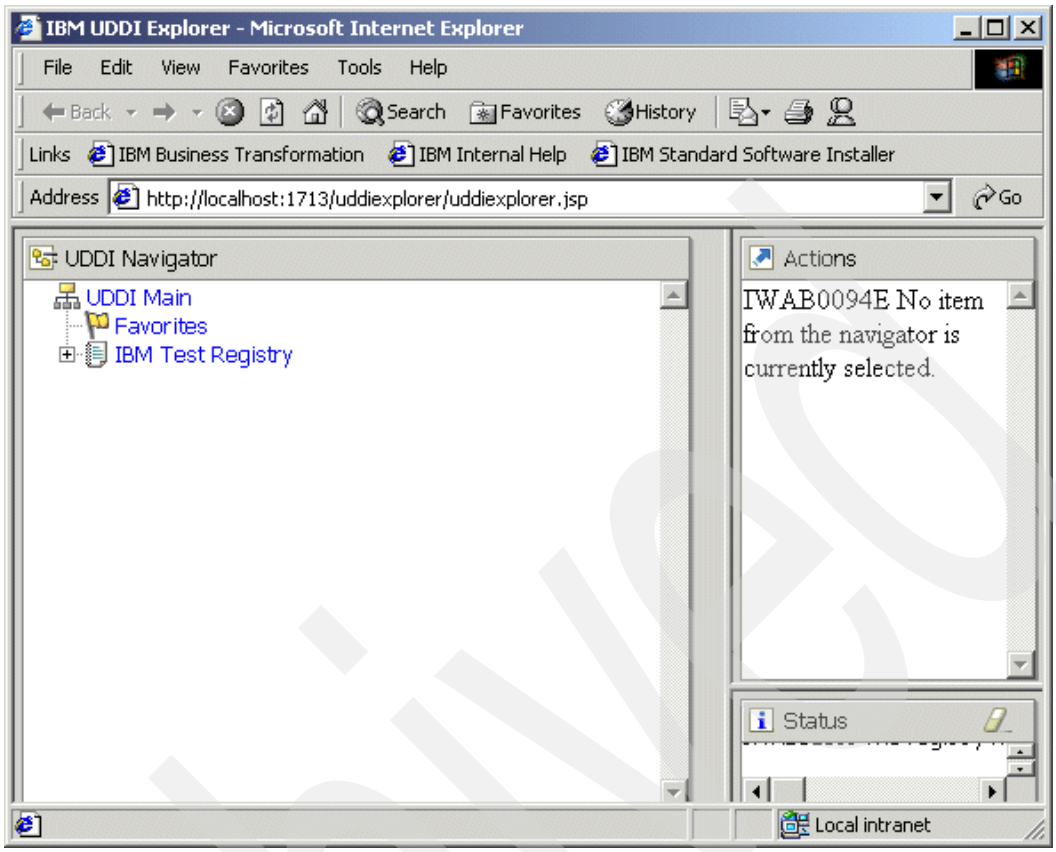


Figure B-4 UDDI Explorer on WDSc

3. Figure B-4 shows the UDDI explorer in WDSc. This explorer, although serving jsp pages from WDSc using a localhost machine, can access IBM Test Registry.

Archive

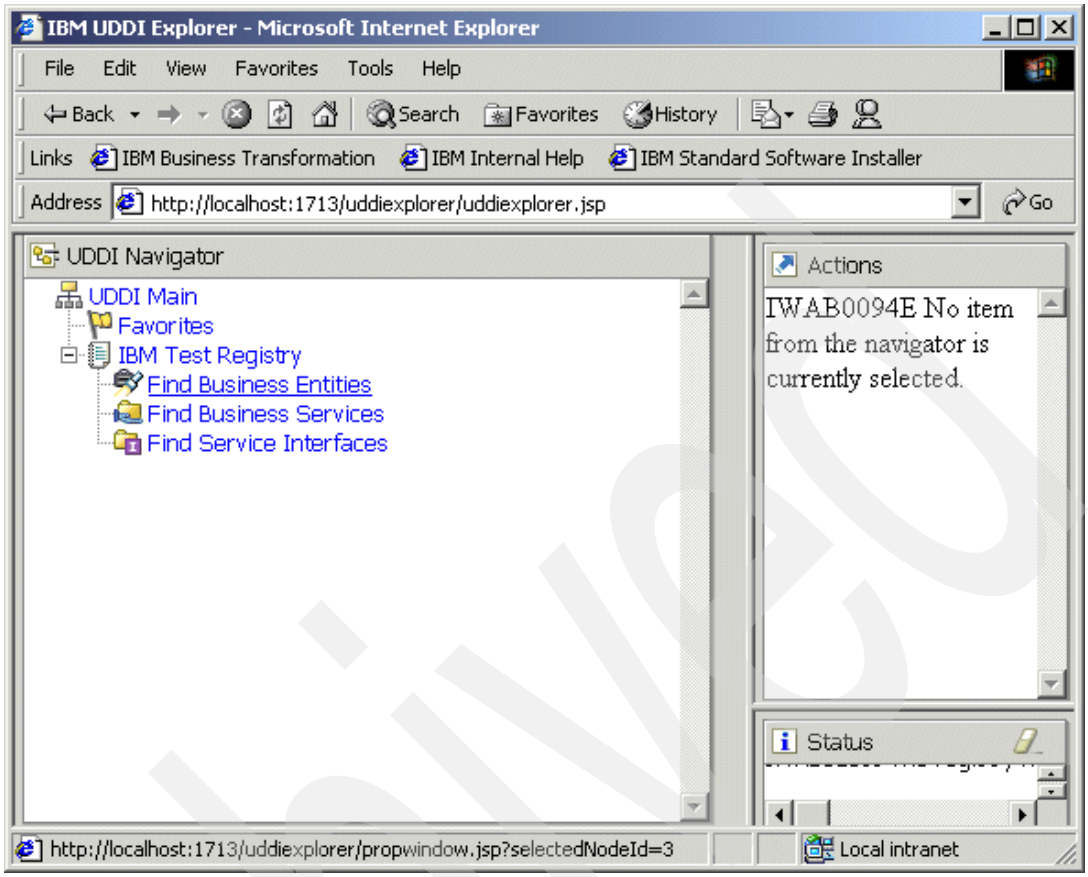


Figure B-5 Expanded View of IBM Test Registry in WDSc

4. As shown in Figure B-5, when you expand IBM Test Registry you get to see the *Find* tools for finding Business Entities, Business Services and Service Interfaces.

   But for our Web Service to become searchable, we need to publish it first.

Archive

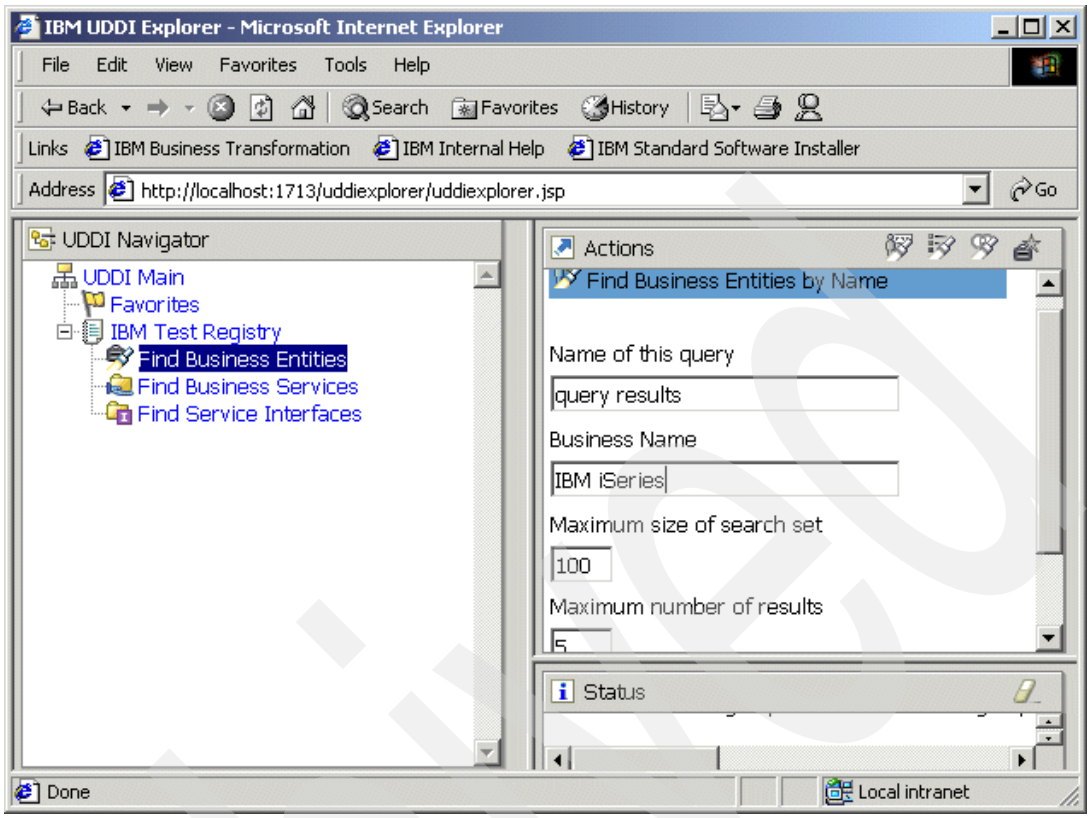


Figure B-6 Search for IBM iSeries Business Entities in WDSc

5. Refer to Figure B-6, click **Find Business Entities** and enter **IBM iSeries** on the right hand Actions pane of UDDI Explorer. We need to search for categories in IBM iSeries under which we can register our Web Service.

Archive

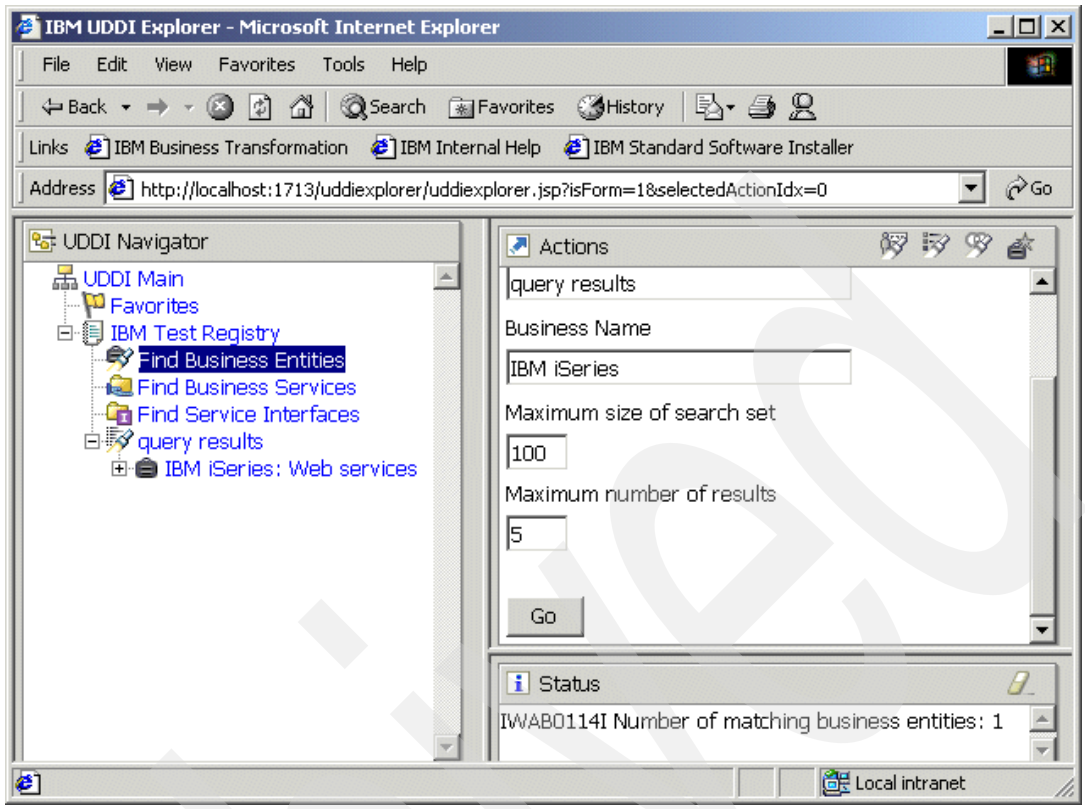


Figure B-7 Query results for IBM iSeries Business Entity on WDSc

6. As shown in Figure B-7, you can see that search on an IBM iSeries business name returned the Web Services in IBM iSeries.

Archive

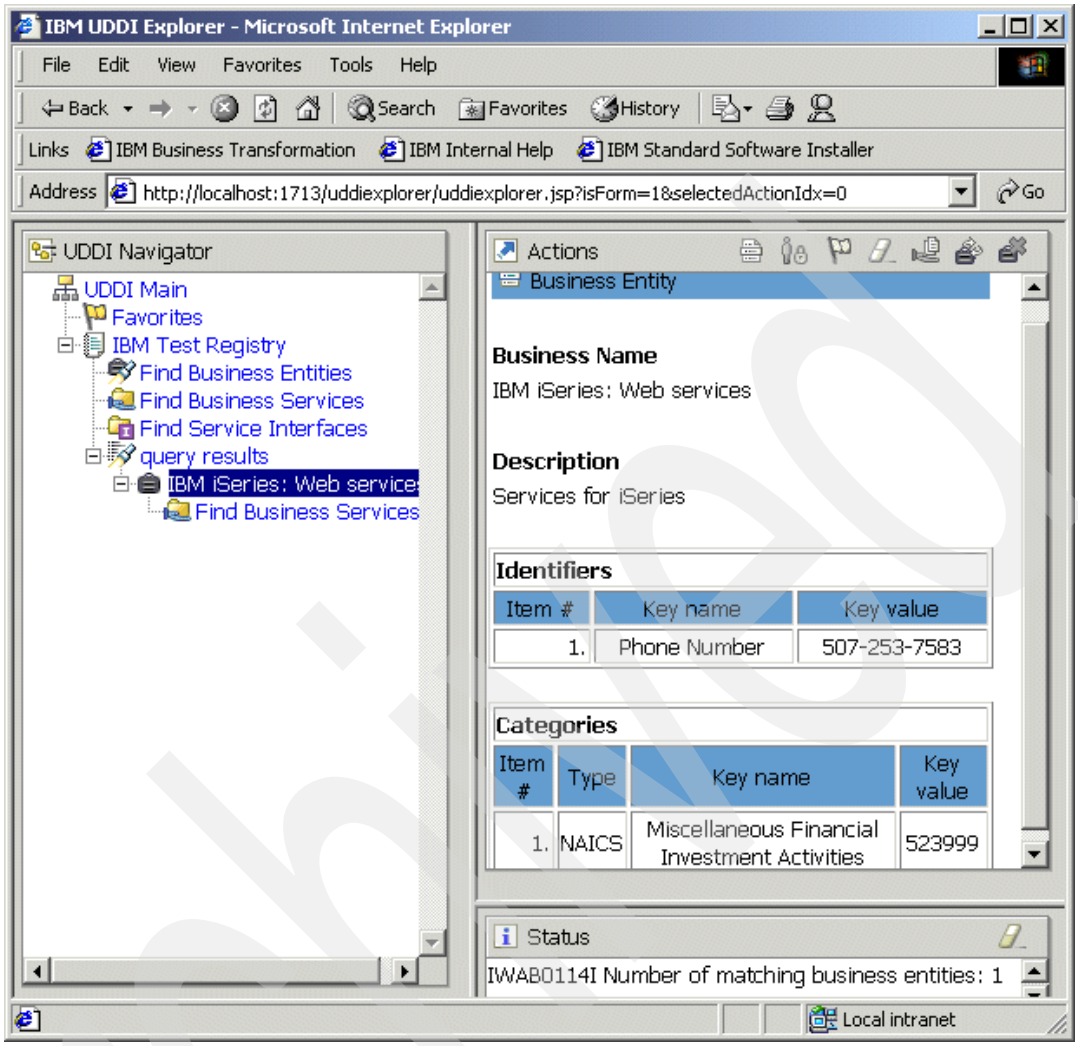


Figure B-8 Details for IBM iSeries registry in WDSc

7. Figure B-8 tells us about the “White Pages” section (refer to “Technical scenario” on page 178) of a UDDI registry. As you can see it gives address and contact details of IBM iSeries UDDI registry. You can also see the categories under which Web Services can be registered. Figure B-9 shows the login window.

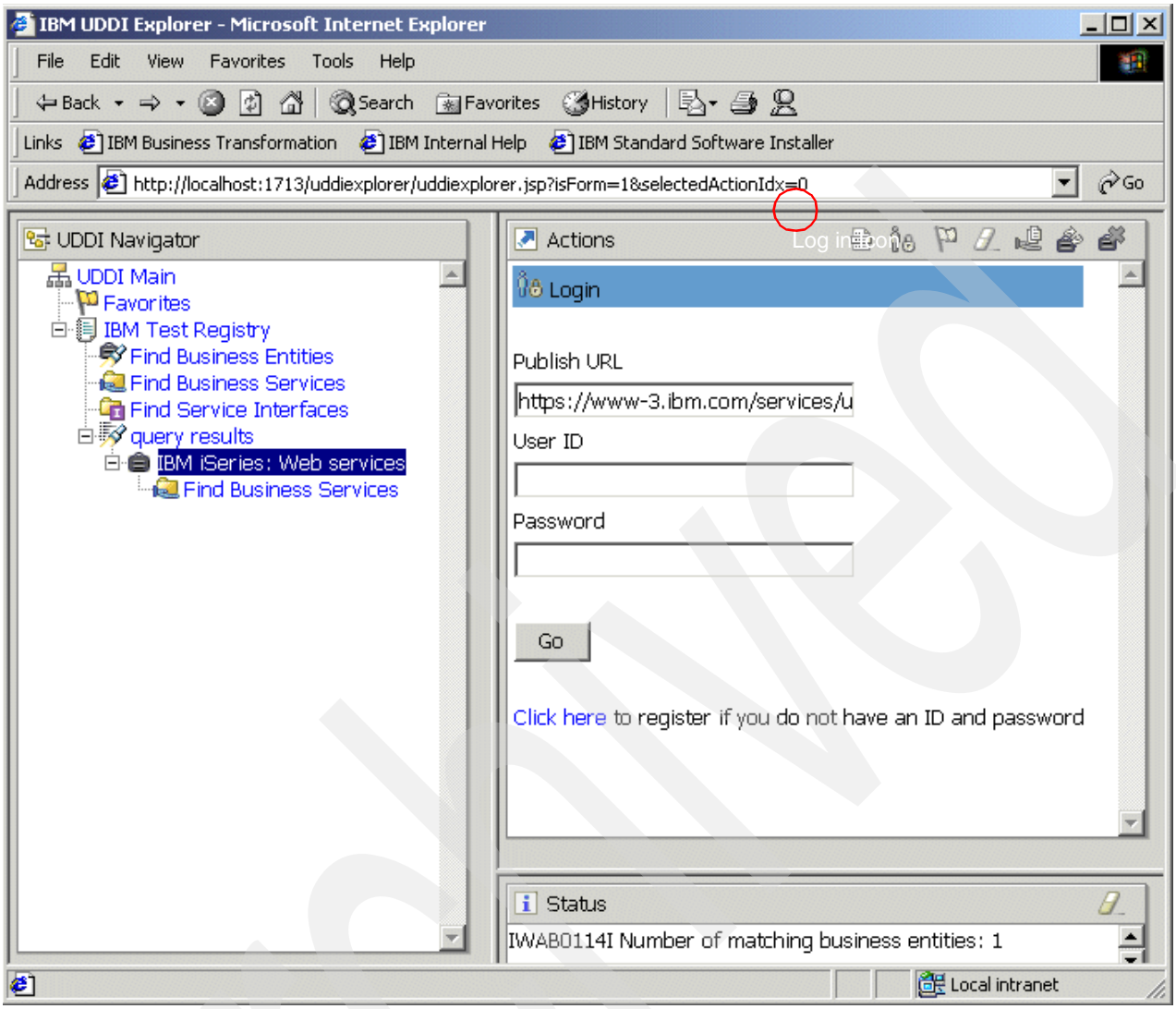


Figure B-9 Login to publish the Web Services in WDsc

8. Click the **Login** icon as shown in Figure B-9. Enter User ID and Password for IBM iSeries Test Registry and press **Go** on the Actions pane. This login information can be obtained through PartnerWorld in Development from the IBM Web site.

   After logging in you will see a window as shown in Figure B-10, where you enter information about the Web Service WSDL files that you want to publish.

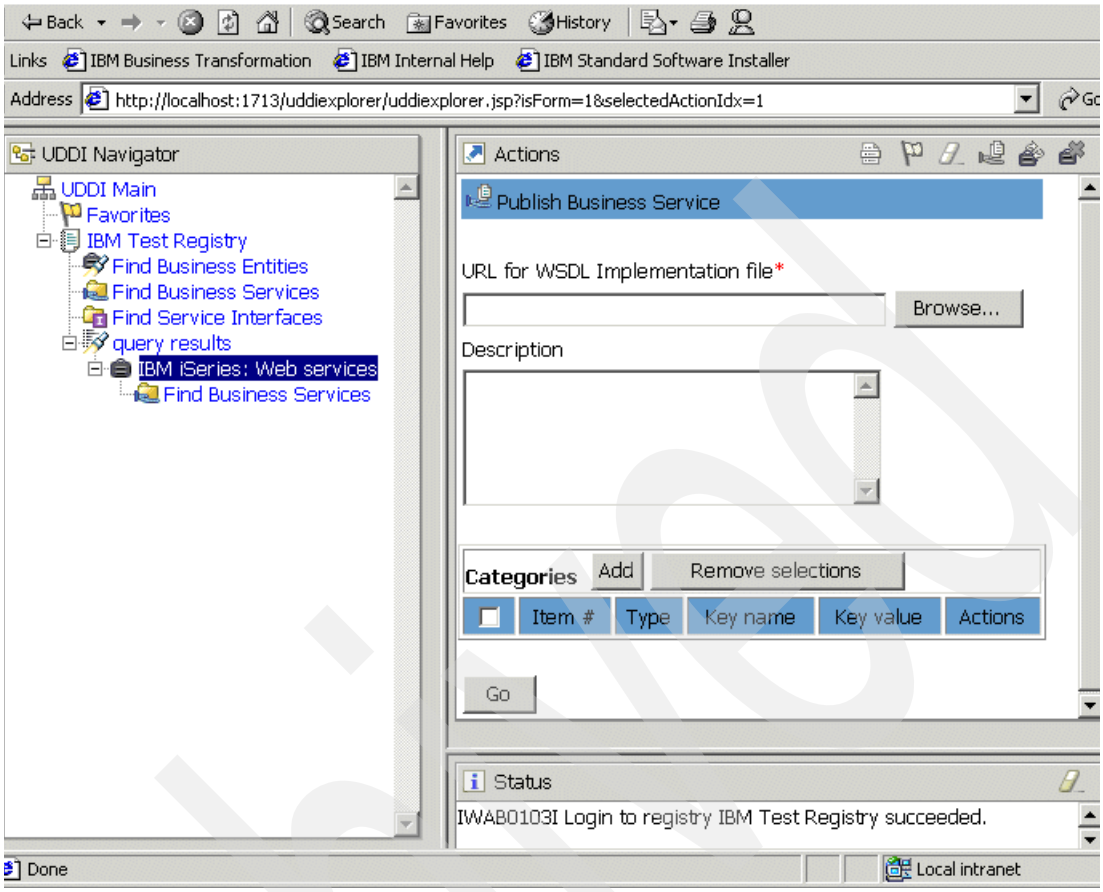


Figure B-10 Specify WSDL implementation file and description on in WDSc

9. Enter the path as the URL for the WSDL implementation file:
   `http://idomws1:5500/iWebSvcLab/wsdl/InventoryAvailabilityService-service.wsdl`

   This is the URL path for the WSDL file for Inventory Availability Web Service installed on WebSphere Application Server on iSeries. This WSDL implementation serves as “Blue pages” of technical information about the services this Web Service exposes. Also enter a description as shown in Figure B-11 for the Web Service and press **Add** on the Actions pane.

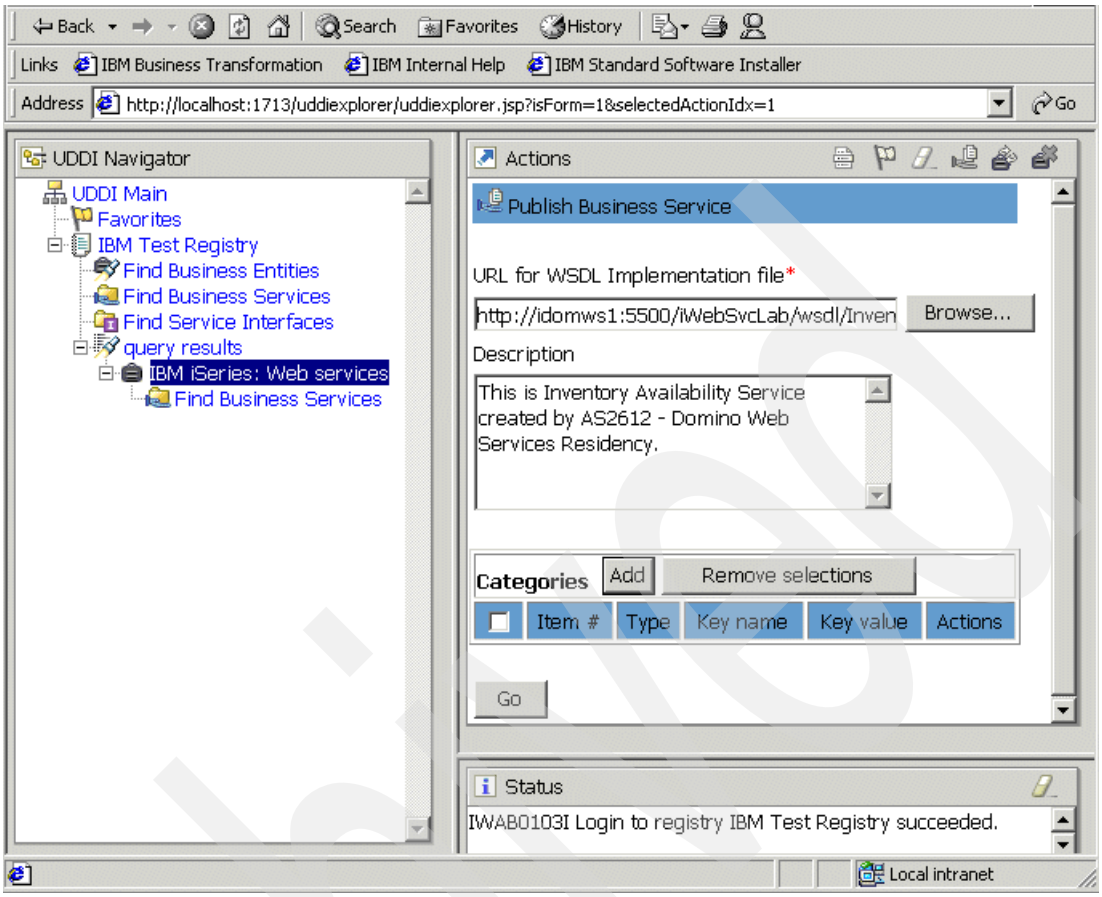


Figure B-11 Add Web Service WSDL implementation and description in WDSc

10. After pressing **Add** on the Actions pane, you will have an opportunity to select Taxonomy type under which you want to publish your Web Service, as shown in Figure B-12.

Archived

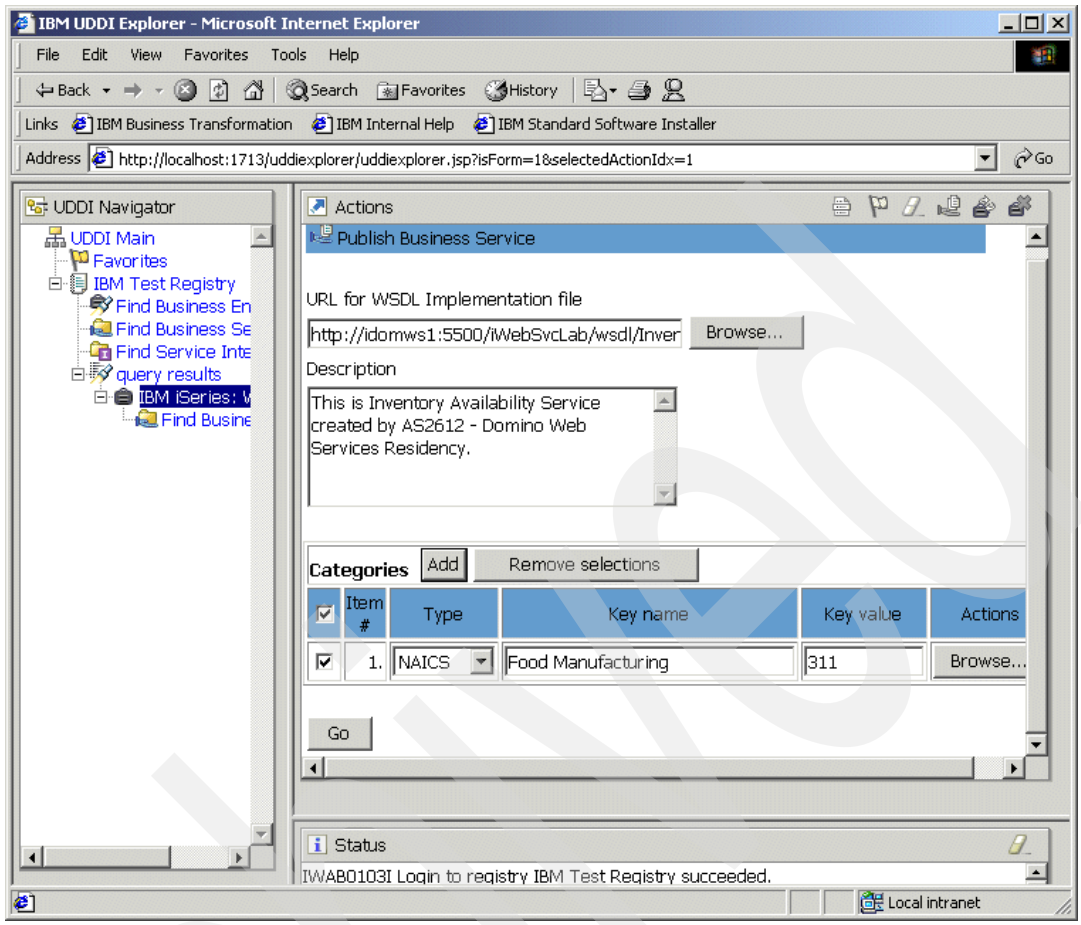


Figure B-12 Choose category for Web Service publication in WDSc

11. Before choosing the Web Services category, click **Browse** to navigate the available hierarchical taxonomy and select **Food Manufacturing** as shown in Figure B-13.

Archive

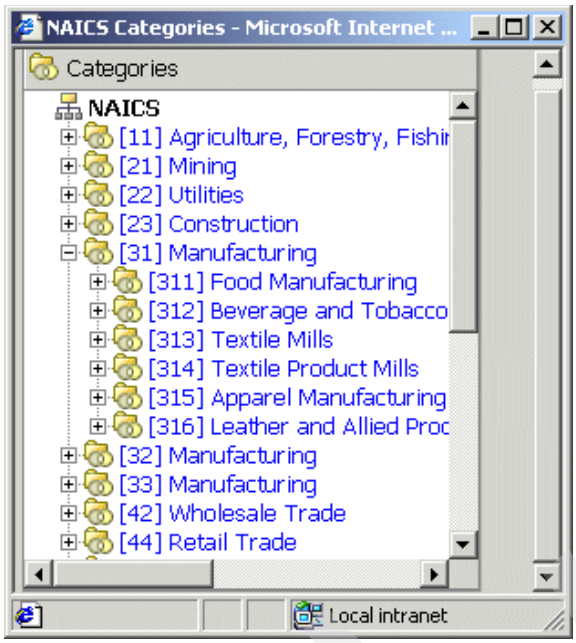


Figure B-13 Browse the categories under NAICS in WDSc

12. Click **Go** on the Actions pane after selecting the appropriate Taxonomy type. If you want to add your Web Service under other Taxonomy terms also, add those additional categories.

Archived

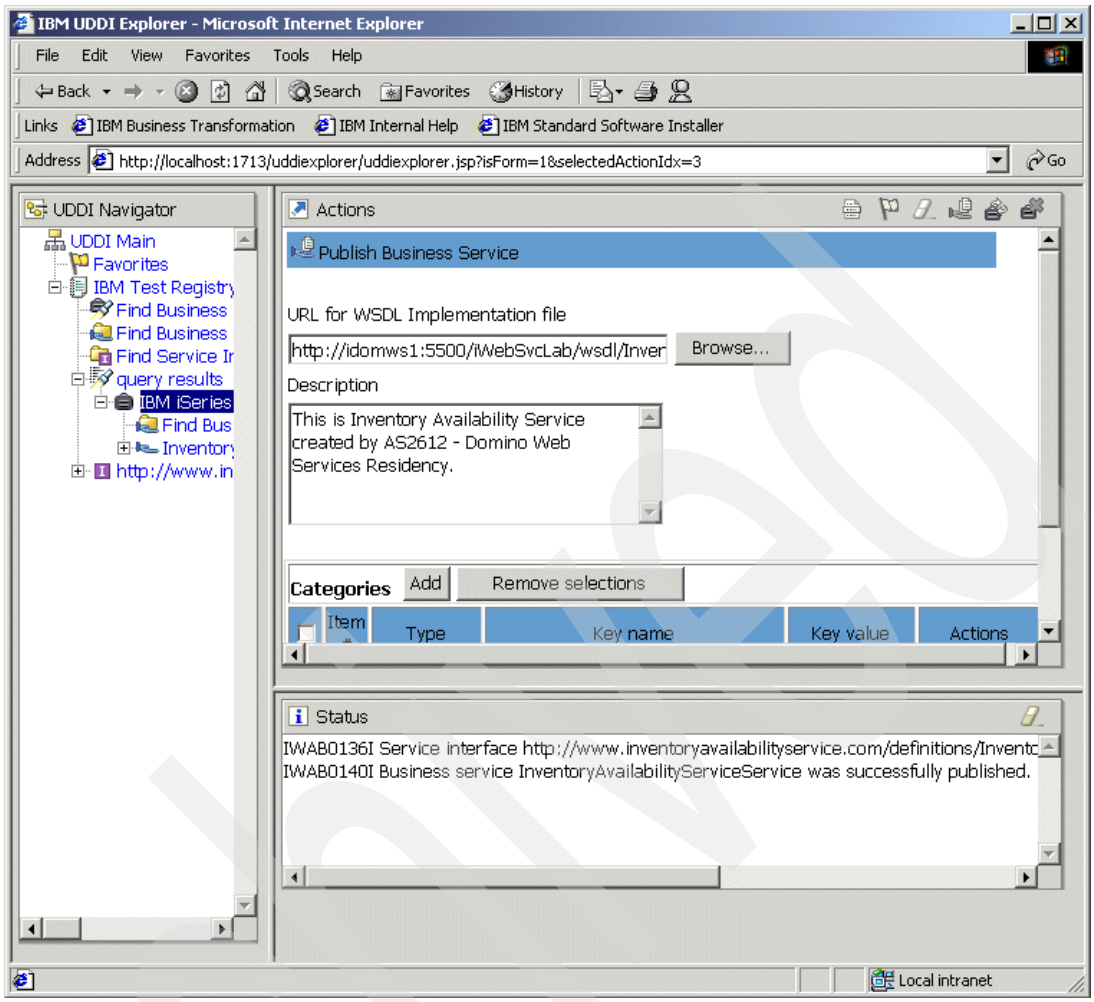


Figure B-14 Publish the Web Service under selected category in WDSc

13. After pressing **Go**, you will see a message on the Status pane as shown in Figure B-14 and you will find your Web Service appear on the UDDI Navigator pane. Congratulations! you have successfully published your Web Service under the IBM Test Registry. Now your Web Service is available over the Internet for other businesses as “Yellow Pages”. If you see errors in Status pane, please refer to “Tips and techniques” on page 204.

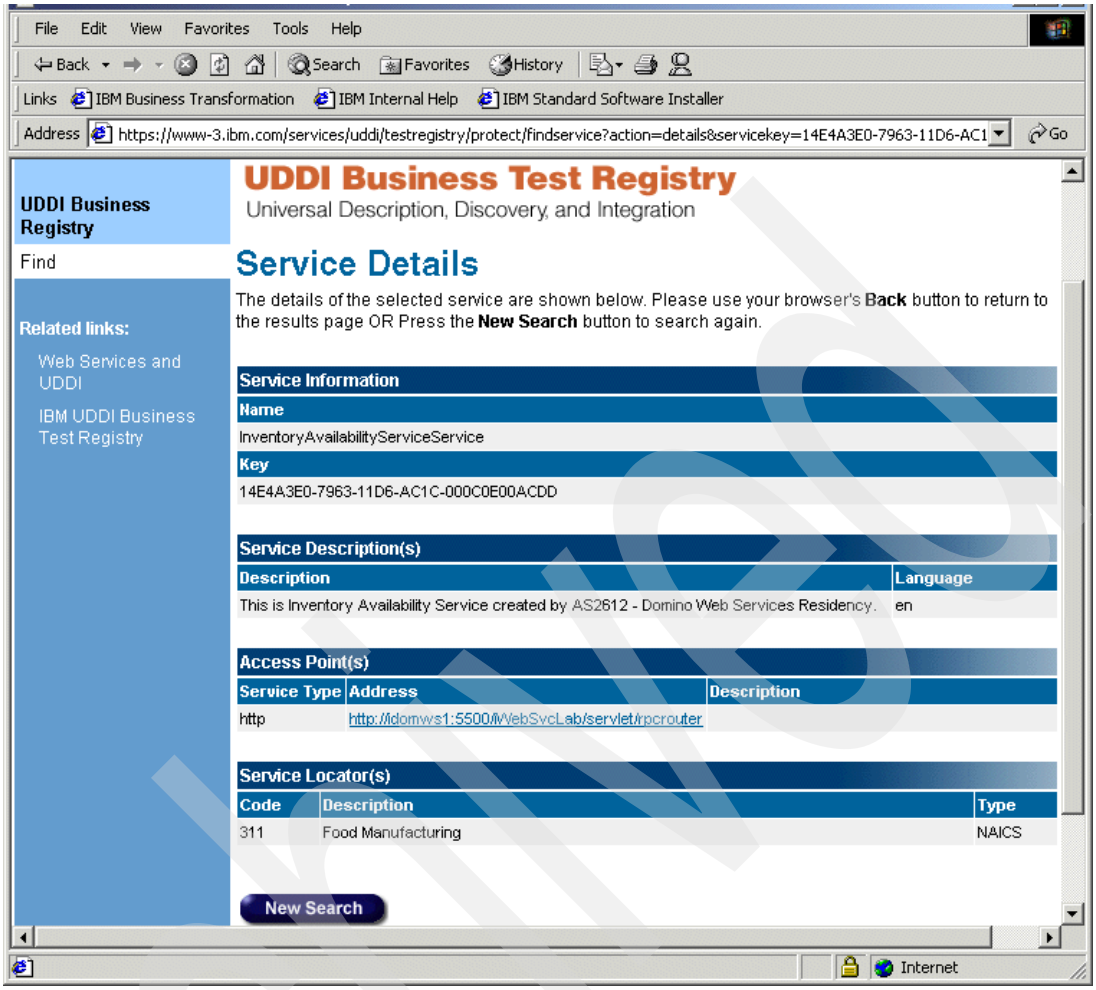


Figure B-15 Inventory Availability Service published on IBM UDDI Registry

Figure B-15 shows that the registry information is now available on UDDI IBM Test Registry.

Creation of Web Services client

Now you are a Web Service requester. In other words, you are about to consume the Web Service from a Web Service provider. We also use WDSc for our Web Service related development and deployment works. For creation of a Web Services client on WDSc, you need to the discover the Web Service using the IBM Test Registry and then import it into WDSc.

Create Web Services client Web Project

First we will create a Web Project on WDSc to import the WSDL files from the Web Service.

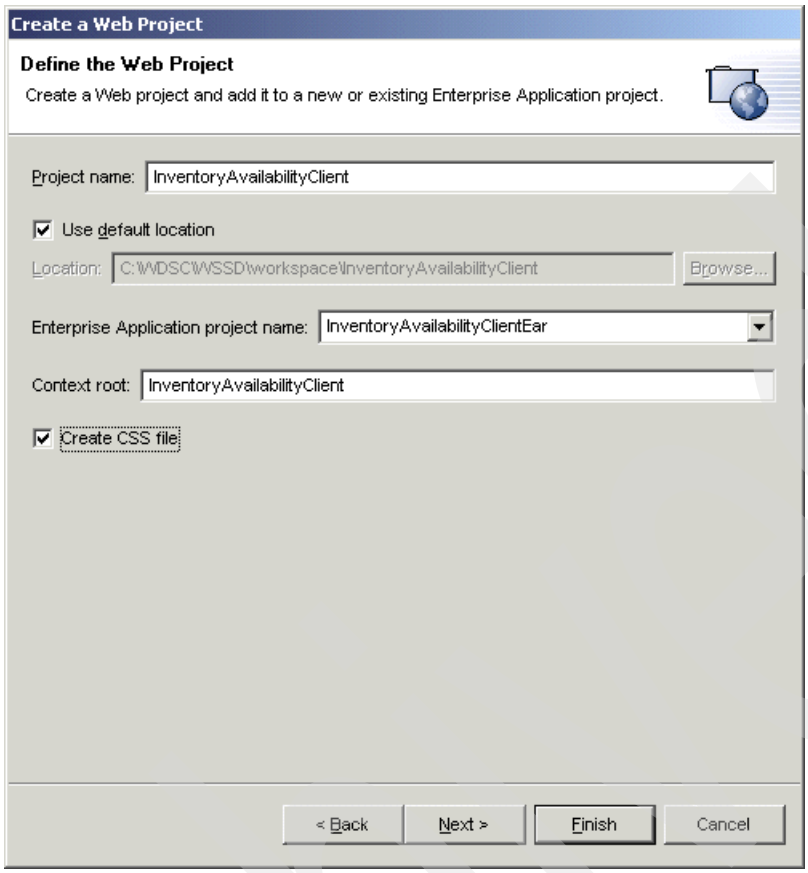


Figure B-16 Create new Web Project for Web Services Client in WDSc

1. Click **File -> New -> Web Project** in WDSc and enter **InventoryAvailabilityClient** as the Web Project name and **InventoryAvailabilityClientEar** as the EAR file name, as shown in Figure B-16.

r

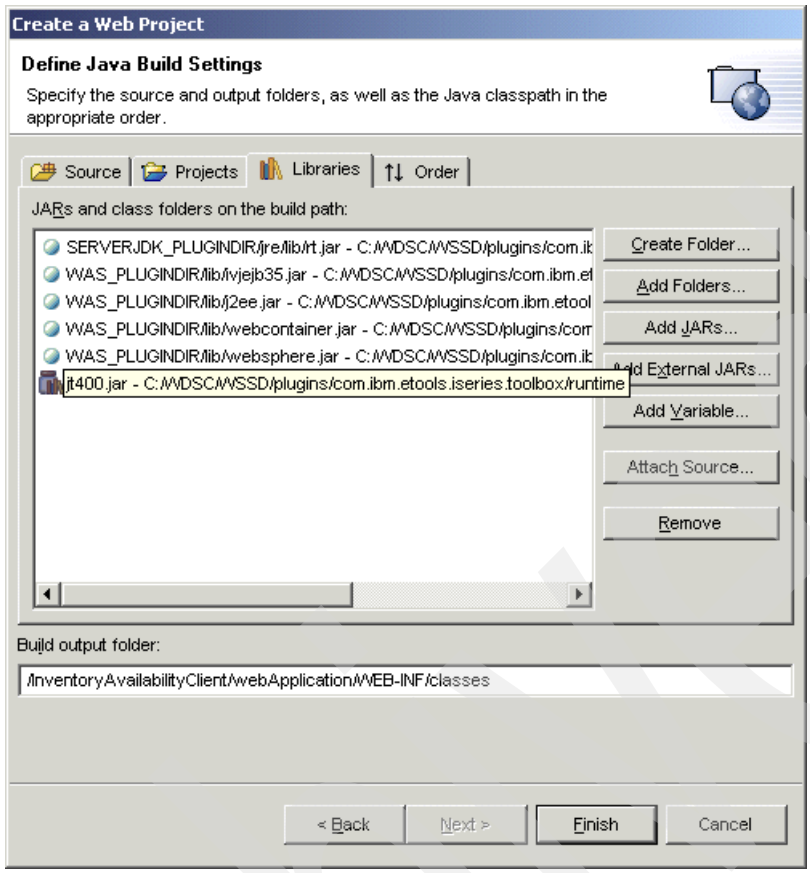


Figure B-17 Add external jar files on WDSc

2. See Figure B-17 and add any external jar files required. Then click **Finish** to create a Web Project.

Import the WSDL implementation files from UDDI Explorer

The next task is importing the WSDL implementation files from UDDI Explorer.

Archive

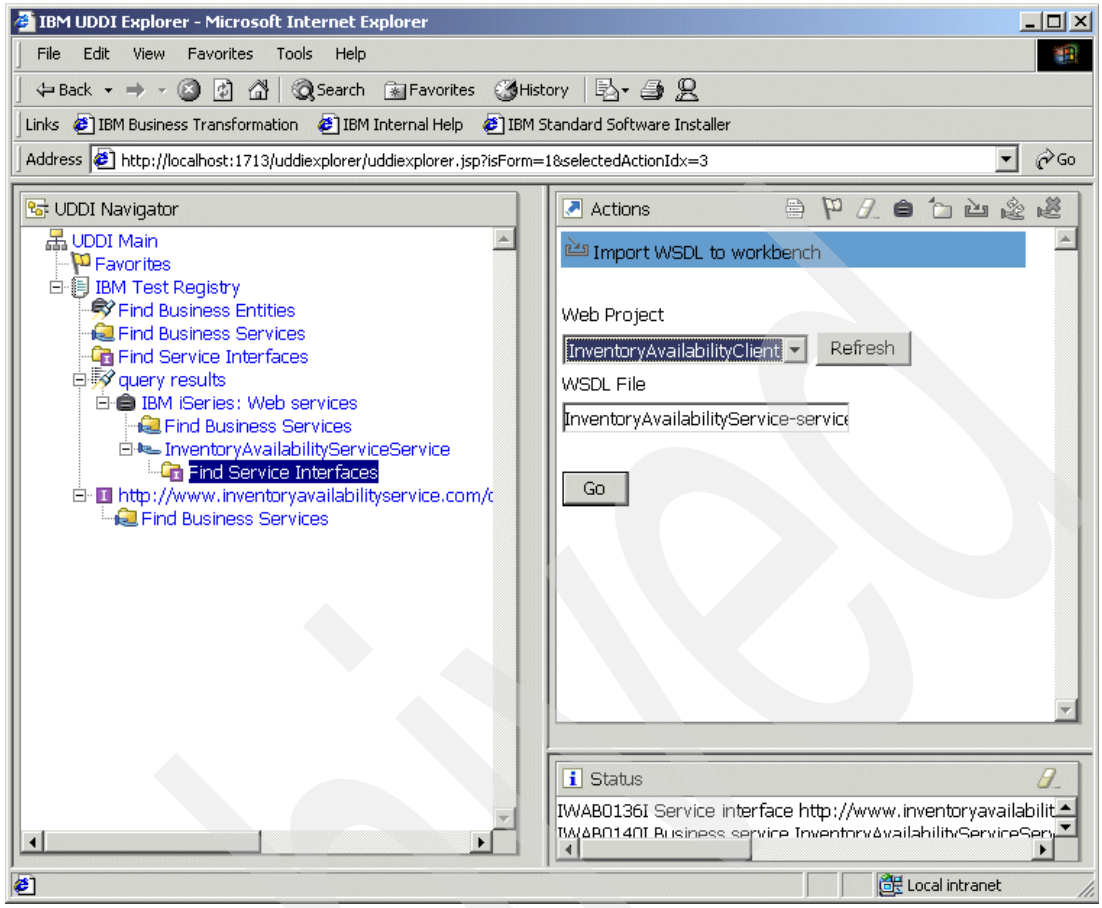


Figure B-18 Specify the Web Project and WSDL file to import in WDSc

1. After creation of InventoryAvailabilityClient Web Project, go to UDDI explorer as shown in Figure B-18 and specify the WebProject name and WSDL file on the Actions pane to import into WDSc.

Archive

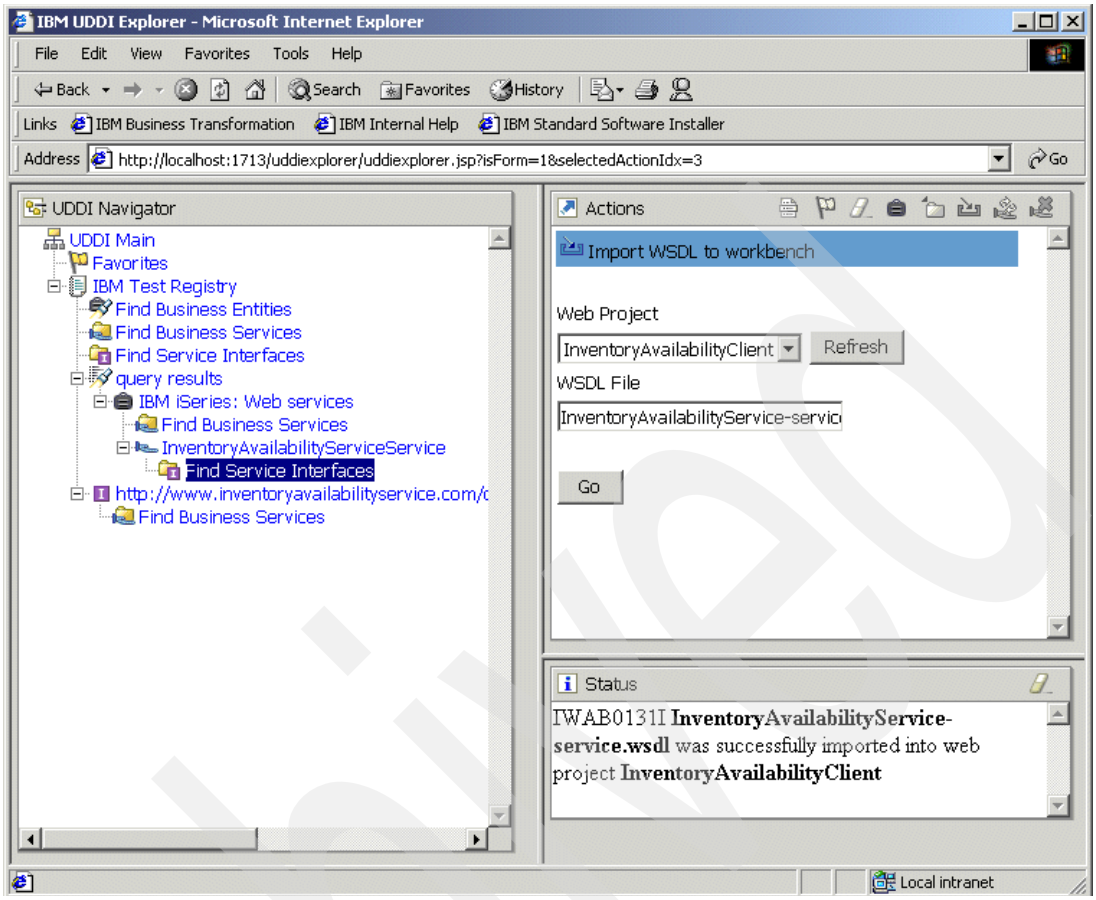


Figure B-19 Successful import messages on UDDI Explorer in WDsc

2. On successful import you will see messages on the Status pane as shown in Figure B-19.

Create and test Web Services client

With imported WSDL files, now we are ready to create a Web Service client application.

Archive

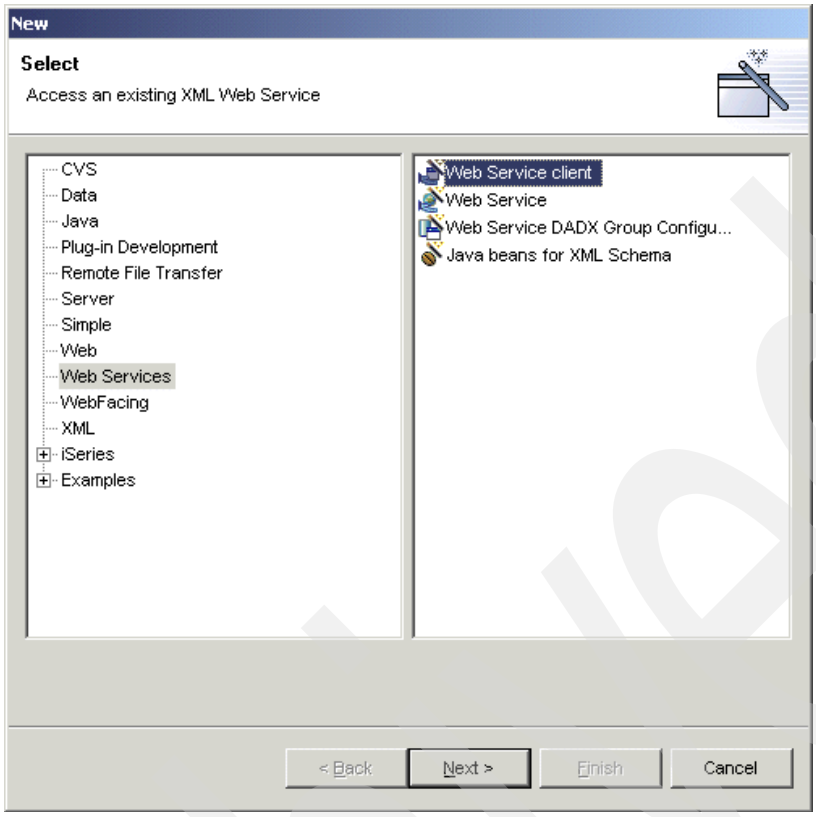


Figure B-20 Create Web Service Client on WDSc

1. To create a Web Services client, click **File -> New -> Web Services -> Web Services Client** as shown in Figure B-20.

Archive

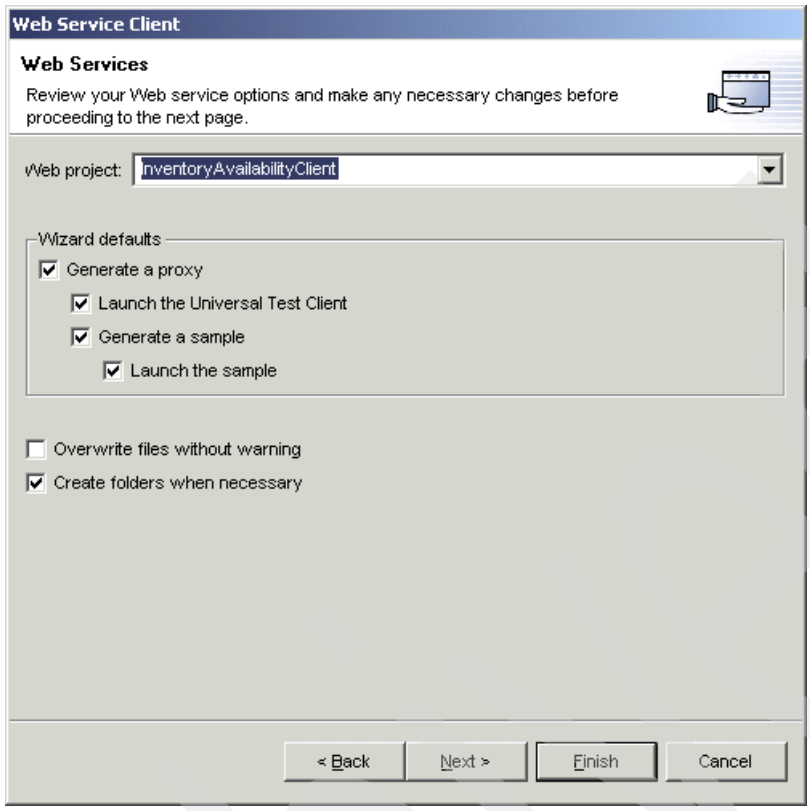


Figure B-21 Name the Web Services client project in WDSc

2. Enter **InventoryAvailabilityClient** as the name of the Web Services client project as shown in Figure B-21. Also check:
   - Generate proxy
     - Launch the Universal Test Client
     - Generate Sample
     - Launch the sample
   - Create folders when necessary

   You need to create sample application for InventoryAvailablityClient so check the sample creation and Test Client options. Click **Next**.

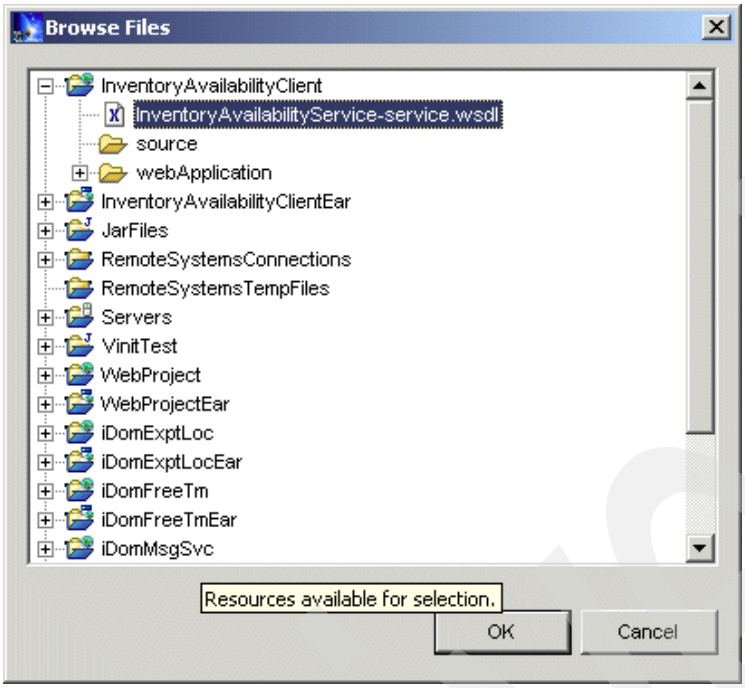


Figure B-22 Browse to select WSDL for Web Service Client on WDSc

3. On the next window, Figure B-22, you specify the WSDL file which you imported in “Import the WSDL implementation files from UDDI Explorer” on page 194.

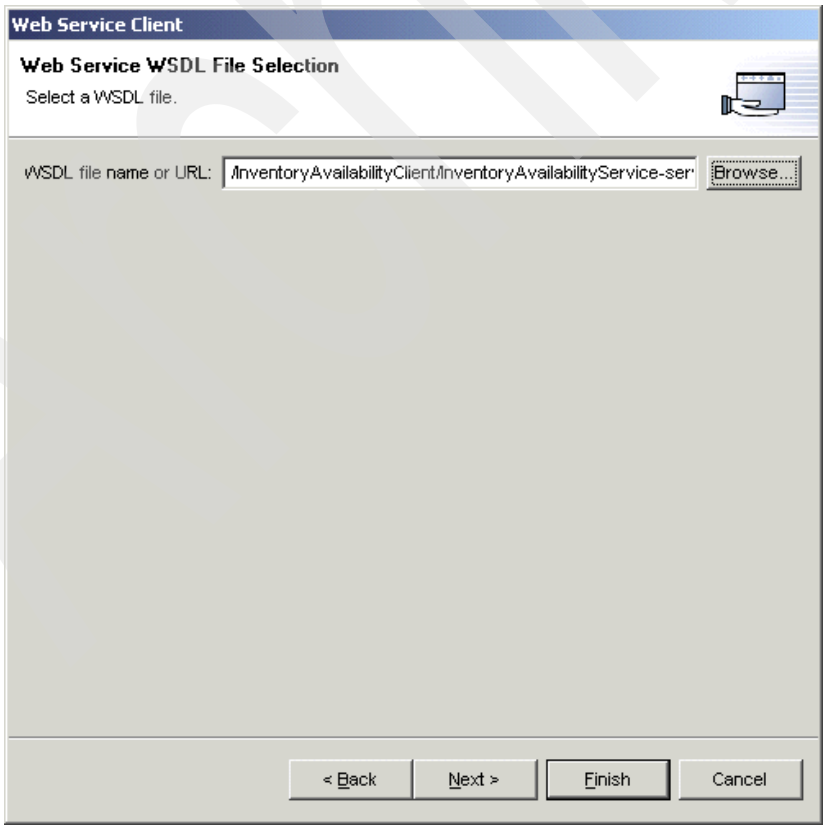


Figure B-23 Web Service WSDL file selection on WDSc

4. You can browse files as shown in Figure B-23. After specifying the WSDL file, click **Next**.

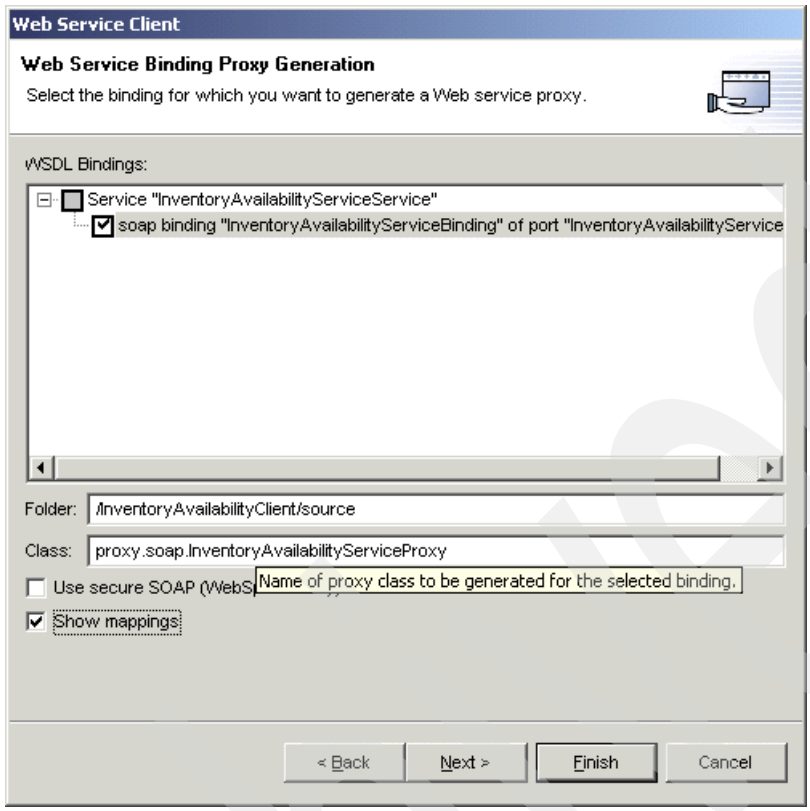


Figure B-24 Web Service Client Proxy Generation in WDSc

5. As depicted in Figure B-24, verify the proxy generation setting, check on **Show mappings** check box and click **Next**.

Archived

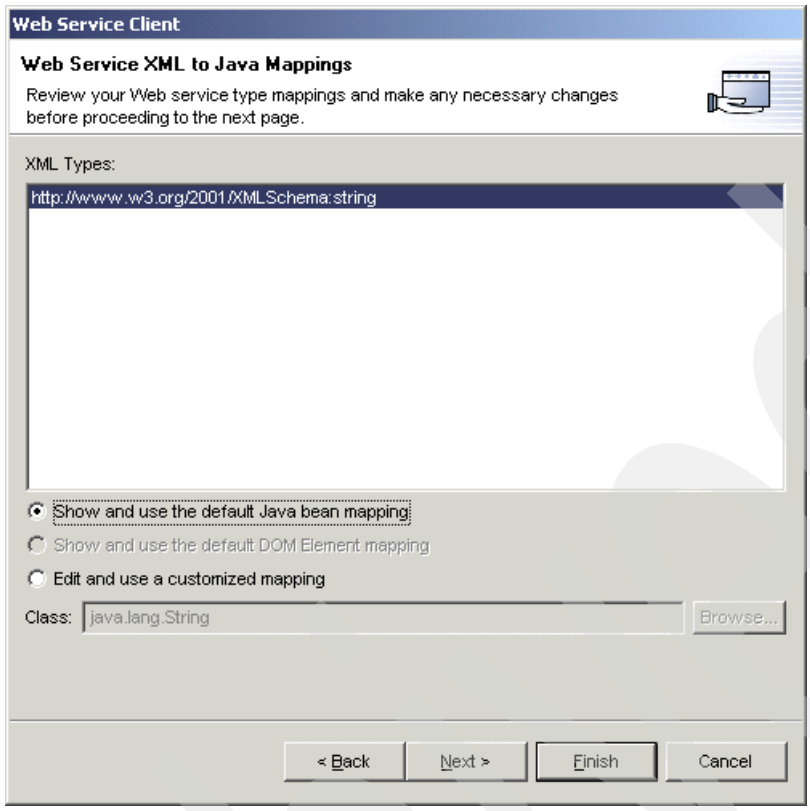


Figure B-25 Web Service XML to Java Mappings

6. Figure B-25 tells about the XML schema mappings for parameters and return types for methods, which the InventoryAvailability Web Service has exposed. Click **Next** on this window.

Archive

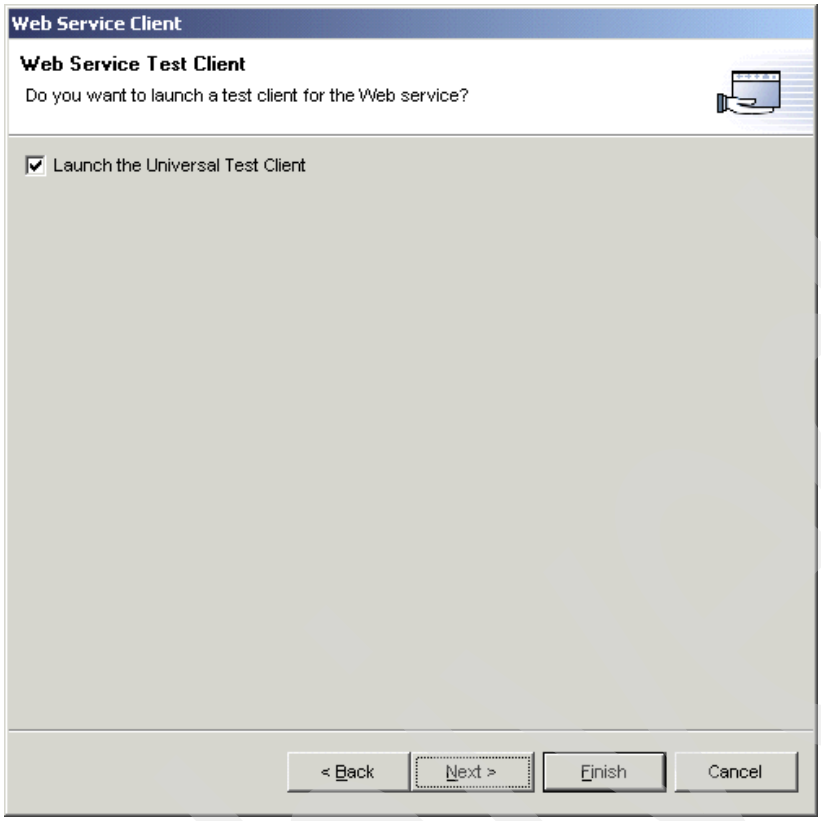


Figure B-26 Selection to launch the Universal Test Client in WDSc

7. As shown in Figure B-26, check **Launch Universal Test Client** and click **Next**. This will automatically invoke the Test Client for the Web Service upon creation. The Test Client is run under WebSphere Test Environment of WDSc. Click **Next**.

Archive

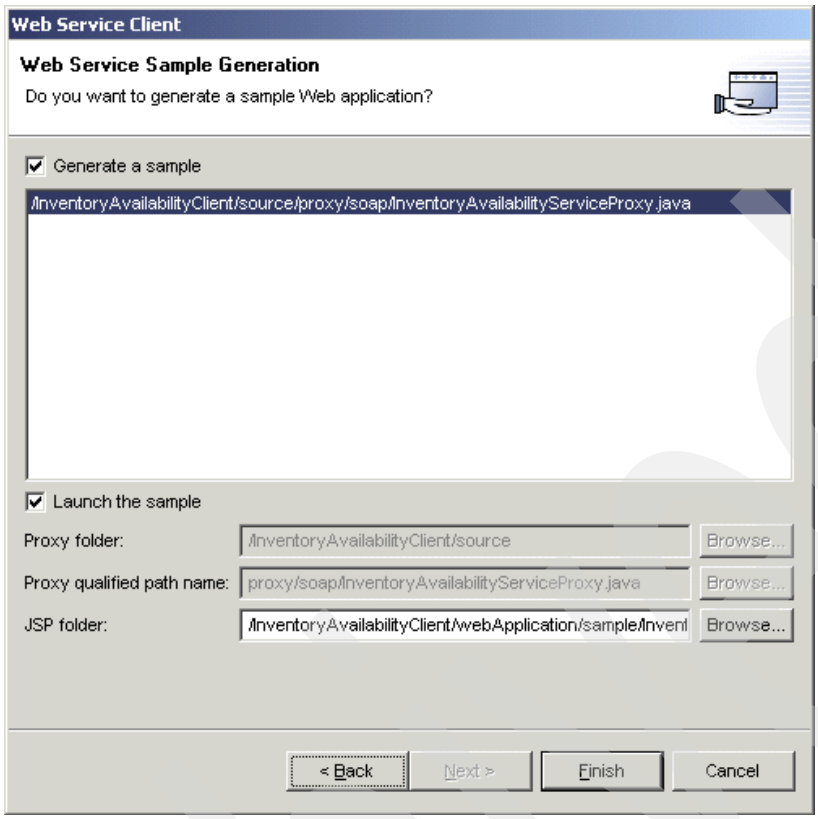


Figure B-27 Generate and Launch sample for Web Service in WDSc

8. On the next window as shown in Figure B-27, check **Launch the sample** and click **Finish**.

   This will create a Web Service client in WDSc and launch the TestClient.jsp file under the WebSphere Test Environment.

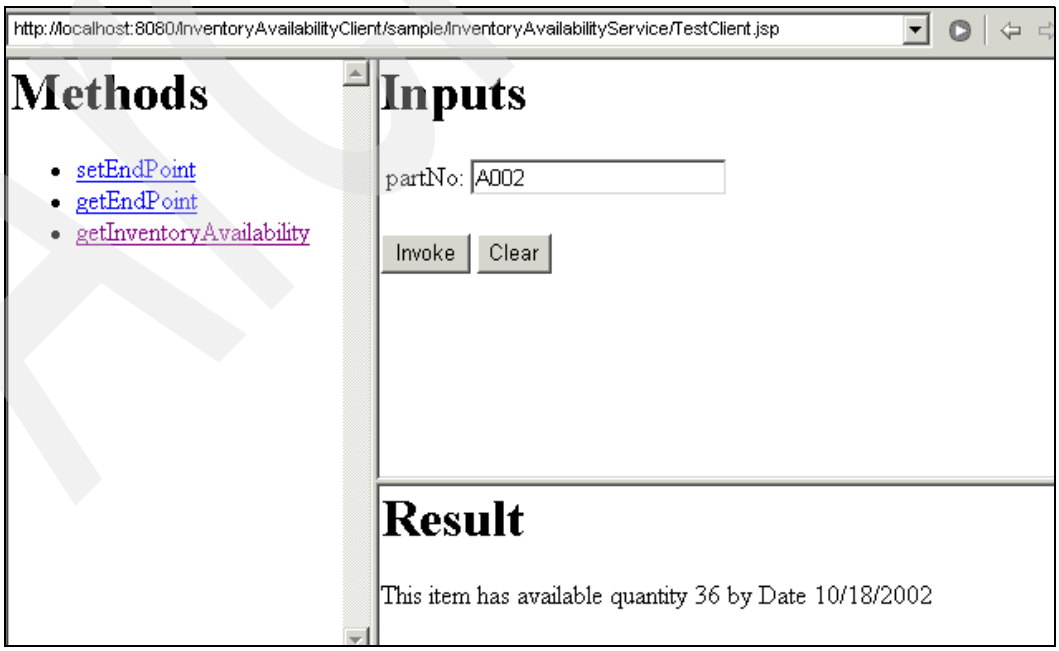


Figure B-28 Verify the TestClient.jsp for InventoryAvailability Web Service in WDSc

9. As illustrated in Figure B-28, you can see the methods exposed by the Web Service listed in left pane. Click getInventoryAvailablity and enter `A002` as partNo on the Inputs pane. You are supposed to have the response as illustrated on the lower right pane of *Result*. Congratulations! You have successfully developed a Web Services client from UDDI.

Tips and techniques

Here are some tips related to the topic of UDDI.

- Make sure you have proper access to IBM Test Registry site or any other commercially available UDDI registry.
- Successful registry of your Web Service is solely dependent on the URL of WSDL configuration file. Make sure that URL path is defined correctly.
- After UDDI registry of the Web Service, try to locate it with the help of UDDI search.

Archived

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 206.

- *Self-Study Guide: WebSphere Studio Application Developer and Web Services,* SG24-6407
- *Lotus Domino for AS/400 R5: Implementation,* SG24-5592
- *WebSphere 4.0 Installation and Configuration on IBM eServer iSeries Server,* SG24-6815

Other resources

These publications are also relevant as further information sources:

- *HTTP Server for iSeries Webmaster’s Guide,* GC41-5434

Referenced Web sites

These Web sites are also relevant as further information sources:

- SOAP technical information, samples, articles:

  http://xml.apache.org/soap
  http://www.PerfectXML.com/SOAP.asp
  http://www.soaprpc.com
  http://www.soap-wrc.com
  http://soap.weblogs.com
  http://msdn.microsoft.com/soap
  http://www.soaplite.com
  http://www.zaks.demon.co.uk/com/soap.html

- Web Services directories:

  http://www.uddi.org/register.html
  http://www.xmethods.com
  http://www.soapwebservices.com
  http://www-106.ibm.com/developerworks/webservices

- Other pertinent Web sites:

  http://www.ibm.com/websphere
  http://tempuri.org
  http://www.xmltoday.com
  http://schemas.xmlsoap.org/wsdl
  http://www-10.lotus.com/ldd
  http://www.w3.org
  http://www.alphaworks.ibm.com

Archived

© Copyright IBM Corp. 2002. All rights reserved.

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Archived

Index

Symbols
(SOAPCall) agent 104

A
AAT (Application Assembly Tool) 8
activation.jar 100
add Xerces.jar file 19
adding External JARs in WDSc 18
adminclient 46
alphaWorks 61
Apache SOAP
  serialization concept 92
Application Assembly Tool (AAT) 8

C
C 7
C++ 7
COBOL 7
com.vck.rst 157
contextual collaboration ix, 214
CORBA/IIOP 4
create new agent
  in Domino Designer 143
create new project in WDSc 14
create Web Service
  StockQuoteService.java in WDSc 25
CSS file 16
customized mapping 32
customized serializer 68
cXML 2

D
DCOM 4
DDS.XML
  deployment descriptor 52
  example 53
default Java bean mapping 32
default location 16
deployment descriptor 52
deserializer 35
directory structure
  after StockQuote creation 39
  WDSc Web Service client wizard 140
directory stucture
  WebProject 20
directory view
  LWSEK contents 94
Discovery Server 5
Document Type Definition (DTD) 4
DOM (document object model) 51
Domino 5.0.10 8
Domino Web Service
  consumer runtime flow 98
Domino Workflow-based applications 5
DTD (Document Type Definition) 4

E
EDI 2
encoding
  input 31
  output 31
encoding envelope 31
EndPoint 110
ERM 6
ExpertiseLocator.html 72
ExpertiseLocator_ServiceService 72
export UDDI for publication
  WDSc 179
eXtensible Markup Language (XML)
  definition 2

G
get database access 66
get result document 67
getQuoteRequest 53
getQuoteResponse 53

H
Help_SoapConnect.nsf 100
http
  //tempuri.org 30, 76
  //www.ibm.com/websphere 7

I
IBM iSeries Business Entity
  UDDI 183
  UDDI query results 184
IBM iSeries registry 185
IBM soap.jar 107
IBM test registry
  WDSc 182
import source files
  WebProject 21
InventoryAvailabilityService 157
InventoryAvailabilityServiceProxy.java 173
iSeries command
  crtnewinst 41
  strqsh 41
  strwasinst 41
  WRKACTJOB 41
iseries.toolbox 155
iWebSvcLab project 155

© Copyright IBM Corp. 2002. All rights reserved.

J
J2EE 6
Java Bean Web Service 26, 159
Java Class mapping 34
jt400.jar 155

K
K-station 5

L
LDAP 6
LearningSpace 5
Lighweight Directory access protocol (LDAP) 7
localhost 40
Lotus Discovery Server 5
Lotus Knowledge Discovery System 5
Lotus K-station 5
Lotus LearningSpace 5
Lotus Web Service Enablement Kit (LWSEK)
  analysis 93
lotus.kds.ws 73
LWSEK
  classes 94
  consumer clients 94
  database 94
  WSDL files 94
LWSEK (Lotus Web Service Enablement Kit)
  analysis 93

M
mail.jar 100
mapping
  Java Class 34
  XMLSchema 34
MessageRequest service 95

N
NAICS 190
NCSOW.jar 92
node name
  adminclient 46
NotesFactory class 66

P
package result document 67
people.nsf 94
plug-ins
  xerces.jar file 19
port number
  12000 41
  5500 41, 43
  8000 41
  8800 41
  adminclient 46
proxy 26
proxy client
  concept 117
  creating from WSDL 124
  example 117
  how to use 142
proxy/soap 43
publish Web Service
  WDSc 191

Q
QSHELL interpreter 41
QShell utility 41
QuickPlace 5

R
Redbooks Web site 206
  Contact us xi
result document 67
Result.jsp
  example 57
RMI 4
Rosetta Net 2
rpcrouter 43
RPCRouterServlet 53
RPG 7

S
Sametime 5
search database 67
search parameters 66
Secure Sockets Layer (SSL) 7
select java bean 27
select library path 46
select New Project type in WDSc 15
select StockQuoteService java bean 28
serialization concept
  Apache SOAP 92
serializer 35
ServiceURI 111
servlet/rprouter 43
session creation description 66
Simplified Object Access Protocol (SOAP)
  definition 2
SOAP
  concept 4
SOAP (Simplified Object Access Protocol)
  definition 2
soap address 53
SOAP binding configuration 35
SOAP encoding 31
SOAP messages 98
soap.jar 19
soap/servlet 110
soapcfg.jar 53
SOAPClient 100
SOAPConnect
  built-in examples 101
  components 100
  introduction 100
  usage 100

Archived

SoapConnect.lss 100
SOAPFault 100
SOAPHTTPConnection 100
SOAPParameter 100
SoapRunner.nsf 100
SoapSamples.nsf 100
source folder
  WebProject directory 20
specify Web Service Java Bean identity 30
specifying context root in WDSc 16
specifying EAR project file in WDSc 16
specifying module dependencies on WDSc 17
specifying project name in WDSc 16
StockPortfolio 101
StockQuote
  sample source 23
StockQuoteApp
  analysis 51
  components view 51
  creation flow 12
  introduction 10
StockQuoteService
  program source code 52
StockQuoteService.java 23
  provider java program 51
StockQuoteService-binding.wsdl 55
StockQuoteServicePort 53
StockQuoteServiceProxy 43
StockQuoteServiceProxy.java 56
StockQuoteService-service.wsdl 54
SubmitExpertiseLocator 72

T

targetNamespace 126
TCP Tunnel/Monitor 114
TempfinderSample 101
test client 26
test client create selection 36
TestClient.jsp 37
  example 57
  results 50
  testing 50

U

UDDI
  concept 4
  publishing Web Service 191
  role 178
  scenario 178
UDDI (Universal Description, Discovery and Integration)
  usage 177
UDDI (Universal Description, Discovery, and Integration)
  definition 2
UDDI explorer 194
  WDSc 181
UDDI export screen
  WDSc 180
UDDI registry
  proper access 204
UDDI search
  locating Web Service 204
UNID (Universal Document ID) 104
Uniform Resource Identifier 8
Unique Resource Identifier (URI) 52
Universal Description, Discovery and Integration (UDDI)
  role 178
  usage 177
Universal Description, Discovery, and Integration (UDDI)
  definition 2
Universal Document ID (UNID) 104
URL path 204
urn
  StockQuoteService 30, 76

W

WAS
  EAR file path 47
  installing enterprise application 47
WDSc
  benefits 13
  creating proxy client 127
  exporting EAR file 45
  exporting Web project 44
  introduction 7
  proxy generation option 33
  Web perspective 39
  Web Service create options 26
  Web Service publication option 38
  webApplication 39
Web perspective 171
Web Service
  definition 2
Web Service consumer
  basic requirements 99
  introduction 98
Web Service consumption
  tutorial 98
Web Service port 53
Web Services model 3
WebProjectEar 16
WebSphere Development Studio Client for iSeries (WDSc)
  introduction 7
WebSphere Studio Application Developer (WSAD)
  introduction 7
WSAD 7
WSDL
  binding description example 55
  concept 4
  example 42, 124
  finding from UDDI 126
  service description example 54
WSDL (Web Services Description Language)
  definition 2
WSTK 95

X

xerces.jar 19

Archived

XML
description 3
example 3
XML (eXtensible Markup Language)
definition 2
XML DOM 51
XML mapping 32
xml4j.jar 19
XMLSchema 34
XMLSchema mapping 34
XML-SOAP admin client 50
xsd.bean.runtime.jar 53

Archived

IBM Redbooks

Domino Web Service Application Development for the IBM @server iSeries Server

(0.2”spine)
0.17”<->0.473”
90<->249 pages

REDBOOKS

Archived

Archived

Domino Web Service Application Development

for the IBM eserver iSeries Server

Redbooks

Step-by-step guide to create your first Web Service application

Guide to transform your Domino application for Web Service

Tips including how to use SOAP, WSDL, and UDDI

"Contextual collaboration" is a new term whirling in the industry today. This phenomenal new type of business asset can be realized by two major mechanisms: the components managing mechanism and the services delivery mechanism. The component managing mechanism deals with calendar, workflow, messaging, e-mail, and so on, while the services delivery mechanism deals with receiving and responding to a request. In terms of IBM product offerings, we have Domino for the component managing mechanism and WebSphere for the services delivery mechanism.

We need standards to connect the applications from different business entities: your customers, your suppliers, your business partners, and you. These business entities may run their business applications on different platforms, different application languages, different networks, and so on. To connect these un-identical entities together, we need standards. For that, we have Web Service standards such as SOAP, WSDL, XML, and UDDI.

This IBM Redbook provides real application scenarios which illustrate how to transform existing Domino applications into either a Web Service provider or a Web Service requester. This is achieved and served through the integration of Domino and WebSphere Application Server.

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

SG24-6862-00 ISBN 0738427314