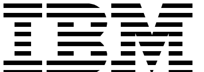

Enabling Web Services for the IBM eserver iSeries Server

Build Web services on iSeries by using this how-to guide

Gain an in-depth understanding of all aspects of Web services

Learn about the new tools from IBM to develop Web services

Aleksandr Nartovich
Scott Gerard
Elena Lowery
Bjarne Matzen

ibm.com/redbooks

Redpaper

International Technical Support Organization

Enabling Web Services for the IBM @server iSeries Server

January 2003

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

First Edition (January 2003)

This edition applies to Version 4 of WebSphere Development Studio Client for iSeries.

© Copyright International Business Machines Corporation 2003. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices v
Trademarks vi

Preface vii
The team that wrote this Redpaper vii
Become a published author viii
Comments welcome viii

Chapter 1. Overview of Web services 1
1.1 The essence of Web services 2
1.2 Enabling Web services 2
1.2.1 XML 3
1.2.2 SOAP 3
1.2.3 WSDL 4
1.2.4 UDDI 4
1.2.5 Alternative approaches 4
1.2.6 Other technologies 5
1.3 Pulling it all together 5
1.4 Substance or hype 6
1.5 Business value 7
1.5.1 Web service provider types 7
1.5.2 Categories of Web services 7

Chapter 2. Understanding the sample RPG application 9
2.1 The legacy RPG applications 10
2.1.1 The Loan Service legacy application 10
2.1.2 The Order Entry legacy application 11
2.2 The Web services implementations 12
2.2.1 Thread scoped versus job scoped 12
2.2.2 Calling an RPG program from Java 13
2.2.3 The socket server 13
2.2.4 The Loan Application Web service implementation 13
2.2.5 The Order Entry Application Web service implementation 14
2.2.6 Possible improvements in the code 14

Chapter 3. Web services application design and architecture 15
3.1 General design guidelines 16
3.1.1 Getting started 16
3.1.2 Architectural issues 16
3.1.3 Implementation issues 16
3.2 Service requestor architecture 17
3.2.1 An application is a requestor 17
3.2.2 An application server is a requestor 18
3.2.3 Integrating multiple Web services 18
3.3 General service provider architecture 19
3.4 Service provider architecture for RPG applications 21
3.5 Our architecture 22
3.5.1 System configuration 22
3.5.2 User interface 22

© Copyright IBM Corp. 2003. All rights reserved.

3.5.3 Stateful versus stateless Web services. 23
3.5.4 Workflow management 24
3.5.5 The Store Web service application architecture 24

Chapter 4. Implementing the Store Web service 27
4.1 Creating RPG application access Java Beans 28
4.1.1 Defining public interface for the access Java Beans. 28
4.1.2 Connecting to an iSeries server 28
4.1.3 Calling an RPG program 29
4.2 Creating Web service wrapper Java Beans 30
4.2.1 Defining a public interface for Web service wrapper JavaBeans 30
4.2.2 Generating Web service deployment files for wrapper Java Beans 34
4.2.3 Files generated by the Web services wizard. 37
4.3 Implementing the Store Web service Java Bean 42
4.3.1 Defining a public interface for the Store Web service Java Bean 42
4.3.2 Generating Web service deployment files for the Store Web service Java Bean . 43

Chapter 5. Web service client 45
5.1 The Store Web application architecture 46
5.2 Testing the Store Web application in Development Studio Client. 47

Chapter 6. Web services deployment 51
6.1 Exporting an EAR file from Development Studio Client 52
6.2 Deploying Web services in the SOAP server 52
6.3 Deploying Web services in WebSphere Application Server 54

Chapter 7. Other topics 57
7.1 UDDI 58
7.2 Security 58
7.3 Transactions 59
7.4 Collecting revenue 59
7.5 Trust 60

Appendix A. Additional material 61
Locating the Web material 61
Using the Web material 61
System requirements for downloading the Web material 61
How to use the Web material 62

Related publications 63
IBM Redbooks 63
Referenced Web sites 63
How to get IBM Redbooks 63
IBM Redbooks collections. 64

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. 2003. All rights reserved.

Trademarks

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

AS/400®	WebSphere MQ®	SP™
IBM®	OS/400®	WebSphere®
IBM eServer™	Redbooks(logo)™	
iSeries™	Redbooks™	

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

Lotus®	Word Pro®

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.

Preface

Enterprises are interested in integrating multiple applications within or across companies. Integrating brings several major benefits:

- The ability to use existing host applications without modifications
- Extended business value for the customers
- Streamlined introduction of new services to the customers
- Reduction in I/T expenses on application development and maintenance
- Simplification of a user interface
- A single point of access for all customer applications

It is the newest integration technology of Web services that provides these great promises. Web services enable the dynamic, flexible, and potentially real-time selection of any kind of business service across the Internet. Business processes running in different enterprises can bind together and cooperate as if they belong to a seamlessly designed system. However, they may be implemented by different software vendors or using different software technologies.

This IBM Redpaper explains developing Web services for an IBM @server iSeries environment. It covers:

- An introduction to Web services
- An introduction to Web services technologies
- Developing a Web services sample application with WebSphere Development Studio Client for iSeries
- Enabling iSeries Web services with WebSphere Application Server

The material assumes that you have extensive programming experience in Java. As such, this Redpaper targets Java programmers who want to start developing Web services.

The team that wrote this Redpaper

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

Aleksandr V. Nartovich is a senior I/T specialist in the IBM ITSO Rochester Center. He joined the ITSO in January 2001 after working as a developer in the IBM WebSphere Business Components (WSBC) organization. During the first part of his career, Aleksandr was a developer in AS/400 communications. Later, he shifted his focus to business components development on WebSphere. Aleksandr holds two degrees, one in computer science from the University of Missouri-Kansas City and in electrical engineering from Minsk Radio Engineering Institute. You can reach Aleksandr at: mailto:alekn@us.ibm.com

Scott Gerard has 23 years with IBM in a variety of leadership and architectural roles. His current assignment is lead architect/programmer in the IBM @server Custom Technology Center software services organization. His responsibilities include working with customers, partners, and ISVs to architect and deploy e-business applications. He works with Java, XML, Web services, object-oriented, and Internet technologies. Scott's previous assignments include serving as the technical lead for IBM's OO-based business components project, the technical lead for an initial system software object request broker project, and the leader of the initial AS/400 openness API project. Scott has authored six external articles and a C++ book.

© Copyright IBM Corp. 2003. All rights reserved.

Elena Lowery is a software engineer at IBM Rochester. She currently works in the IBM @server Custom Technology Center where she designs and develops custom applications for IBM Customers and Business Partners. She is a Sun Certified Java Programmer with five years of experience in object-oriented programming and Java. Her areas of expertise include Java, XML, Web services, and Internet software development. Elena can be contacted at mailto: elowery@us.ibm.com

Bjarne Matzen is the director of product development for Mid-Comp International, an Australian-based IBM Business Partner. He has more than 18 years experience with application development and systems integration, spanning from industrial robotics to commercial logistics systems (ERP). His area of expertise includes the iSeries server with RPG, C, and Java, as well as object-oriented programming on various other platforms. He is a frequent speaker on Java performance and was the original author of the Snapshot/400 performance monitoring system, which is widely used by iSeries customers worldwide. His current responsibilities include the development of a Java-based ERP.

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

Barbara Morris
IBM Toronto, Canada

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 papers to be as helpful as possible. Send us your comments about this Redpaper 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

1

Overview of Web services

Web services are essentially software services that available for consumption over the Internet. However, they extend basic Internet person-to-program interactions, such as individuals accessing programs on Web servers via browsers, to support program-to-program interactions. Data from one Web site can be integrated with data from another site today, but it often involves tricky HTML parsing or manually cutting and pasting elements together. The new technologies that enable Web services allow programs to automate this integration.

© Copyright IBM Corp. 2003. All rights reserved.

1.1 The essence of Web services

Web services are concerned primarily with the program-to-program interactions. For example, you can use Web services to connect the backend of an enterprise system to markets and other industrial partners. Or you can use Web services internally for example, for order fulfillment and employee management.

They also provide service offerings between a business and its customers, just as in the browser-based Internet model. Web services let businesses deal with daily tasks or deliver services programmatically. The selection of which Web service to use can be dynamic.

Figure 1-1 shows the three stages of the Web services puzzle and their interactions:

1. A service provider develops a new Web service application and publishes information about the service (the interface) with a service broker.
2. A service requestor (a program) uses the broker to find services. The service requestor applies various criteria to choose which service or services to select, or the service requestor displays a list and lets the user decide.
3. The service requestor retrieves the stored information from the broker, including how to connect to the service provider. Then the requestor binds to the provider and uses the service.

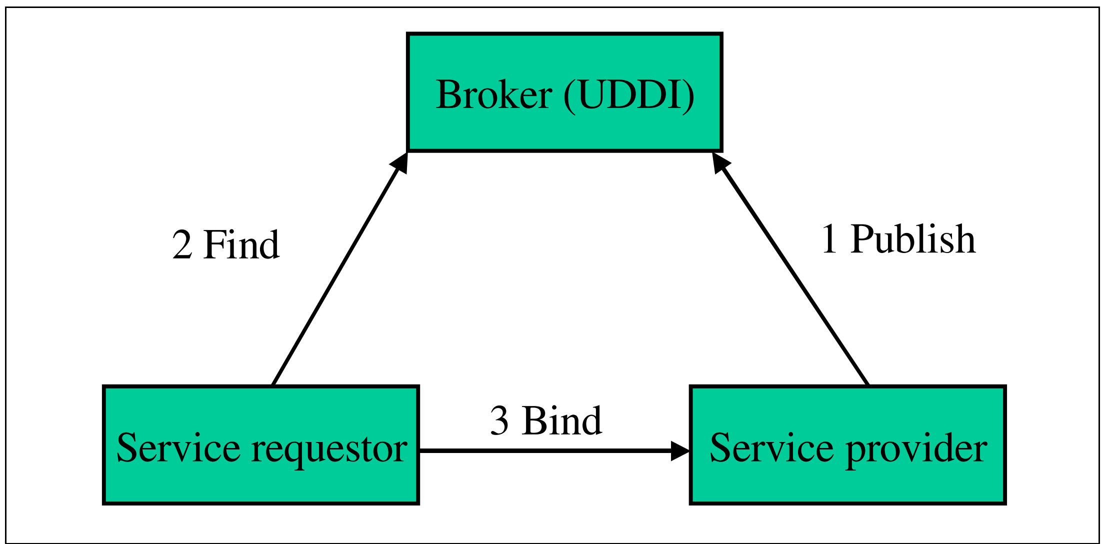

Figure 1-1 Web services model

This process is analogous to a business (a service provider) that publishes information about itself in the local phone directory (a broker). A customer (a service requestor) finds the business and its advertised service in the phone book and calls the business to use those services.

1.2 Enabling Web services

In many cases, several new technologies are used in developing a Web service. Some people may legitimately argue that Web services can be written without using any “standard” technologies. However, most people see Web services being written using these technologies:

- eXtensible Markup Language (XML)
- Simple Object Access Protocol (SOAP)
- Web services Description Language (WSDL)
- Universal Discovery Description and Integration (UDDI)

If you want to “do Web services”, then you should be familiar with these technologies.

1.2.1 XML

eXtensible Markup Language is an extensible tag language that can describe complicated structures in ways that are easy for programs to understand. Web services depend heavily on XML.

XML uses textual data rather than binary data. Of course, binary data is generally more compact and faster to process. However, after decades of attempts to develop a standard, there is little agreement about how binary data should be exchanged. Since XML is textual data, the problems with different hardware or languages using different binary representations for even simple things, such as integers, do not occur. This helps to make XML language- and platform-independent.

It takes fractions of a second of CPU time to create and parse XML data. This adds up to far less processing time than the necessary programmer man months to convert lots of software to a new binary format. You should not be so concerned with XML parsing time. What’s more important is that the business programs talk to each other. It XML that enables this conversation.

1.2.2 SOAP

Simple Object Access Protocol is an XML protocol that is used to describe how to build and process a message that is sent between parties that participate in Web services.

SOAP is a particular set of XML tags. With the SOAP XML tags, you can define a *remote procedure call* (RPC). The outer tag is called the *SOAP envelope*. Procedures have parameters and return values that must be passed. XML tags within the SOAP envelope describe each of the parameters and the return value.

SOAP envelopes can be sent using a number of different communications protocols. The current SOAP specification, as written, only covers sending SOAP envelops over HTTP either with or without the HTTP Extension Framework. However, the specification makes it clear that SOAP can also be extended to other protocols. For example, you can send SOAP envelopes through e-mail via Simple Mail Transfer Protocol (SMTP) or over a messaging products such as WebSphere MQ. For more information about SOAP, see:

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

Since XML and HTTP are language- and platform-independent, they let you talk to any service provider’s Web server that understands HTTP and XML. As an example, all publish, find, and bind operations in Figure 1-1 are usually sent as SOAP envelopes over HTTP where the service provider and broker run in Web servers.

Why do Web services usually use SOAP? Existing binary standards may be faster, so why not just use an existing RPC mechanism such as Java's Remote Method Invocation (RMI) or CORBA's Internet Inter-ORB Protocol (IIOP)? A problem is that these binary protocols use TCP/IP ports that security-conscious network administrators block at their firewalls. You can launch a large “free our IIOP packets” campaign, but it's doubtful that already beleaguered administrators would quickly open their firewalls to binary data that they can't easily examine and understand. HTTP can already pass through most existing firewalls (unless the

administrator is completely paranoid). Furthermore, since SOAP is text data rather than binary data, administrators can visually examine any data they want that makes them more comfortable.

Note that the current implementation of SOAP also supports message encryption between client and server using Secure Sockets Layer (SSL) and digital signatures.

1.2.3 WSDL

Web services Description Language is a different set of XML tags that are used to describe the interface to a Web service. (WSDL is similar in function to CORBA's Interface Definition Language (IDL).) When a Web service application is published to a broker, the application includes WSDL to describe which parameters must be passed on the SOAP request and which return values come back. IBM's Web services Toolkit includes tools to create WSDL from existing programs and to convert WSDL for a Web service into a Java class that can call the service using SOAP. You can find this Toolkit on the Web at:

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

1.2.4 UDDI

Universal Discovery Description and Integration is a specification for distributed Web-based information registries of Web Services. UDDI is also a publicly accessible set of implementations of the specification that allow businesses to register information about the Web Services they offer so that other businesses can find them.

UDDI provides SOAP APIs that describe the interface of a service broker. You can search a UDDI registry for a provider by company. Or, you can search for all providers of a certain type of service (because services may be classified using standard industrial taxonomies). You can even get the technical specifications for calling the provider's Web service (for example, how to talk (or bind) to the service).

As software resources, Web services need to be incorporated into centralized repositories so they can be found. Prior to UDDI, there was no easy way to obtain information about the different services that companies support. There was no single point of access to let you target all possible trading partners and their respective services. Without UDDI, service requestors or providers would have to support multiple repository types.

In addition to this standardized broker functionality, the UDDI.org founding companies have launched the publicly accessible, jointly operated, multisite *UDDI Business Registry*. When you publish to any one of the participating companies' registries, your information is automatically propagated to all other registries. For more information, see the UDDI Executive white paper at:

`http://www.uddi.org`

1.2.5 Alternative approaches

Why do we need these new technologies to support Web services rather than augmenting existing Web technologies and protocols?

Augmenting HTML (primarily a visual presentation technology) won't work because even minor changes in the HTML format can confuse programs trying to parse it.

Why not generate Web pages that return XML? That's basically what SOAP does. However, you also need XML in the HTTP request. That is because Web services can do much more than simply retrieve a page of data. Web services are remote procedure calls, and their input

parameters can be XML, which is more complex than HTTP's GET and POST form commands can handle.

SOAP prescribes standard ways to identify which method to call and the structured parameter data being passed. The SOAP envelope can also contain a header section that includes additional information about the call (for example, security and transaction information).

1.2.6 Other technologies

XML, SOAP, WSDL, and UDDI are just the start on the technologies that can or must be used when building Web services. See Chapter 7, “Other topics” on page 57, for more information.

1.3 Pulling it all together

Becoming a service provider is a relatively straightforward process:

1. You write a service implementation. You can write the implementation entirely in Java, or you can write a servlet running in an application server to call an existing legacy application.
2. Using the appropriate tool from the toolkit, you generate and edit the WSDL that describes your service’s interface.
3. You publish your interface to a UDDI public or private broker and wait for the service requests to start pouring in.

Depending on your service's function, you may need to write a SOAP-enabled client to access your service. Hopefully, some Web service interfaces (in WSDL) will become widely used and published in UDDI independently of any specific service applications. Then a common Web service requestor (client) for that interface could be written to work with any Web service provider that support the common interface. As such clients grow in popularity, more and more service providers will support the common WSDL interface.

Service provider applications can be built easily with Java servlets that bridge between HTTP SOAP requests and other components. The current SOAP toolkit has a flexible architecture so it can call Java Beans, Enterprise JavaBeans (EJBs), or Component Object Model (COM). Servlets can also bridge to your existing legacy applications even those written in RPG and COBOL. By doing so, SOAP servlets can convert existing applications into Web services.

But let's consider the SOAP mechanism. In Apache SOAP, the SOAP server is implemented as a Java servlet (called *rpcrouter*) that runs in an application server such as WebSphere. The servlet parses the incoming SOAP envelope. Then, it uses data in the envelope to select which Java class and which method to call.

The servlet also handles converting parameters and return values between XML and Java data types. Writing a Web service is usually a matter of writing a Java class. A HelloWorld Web service is merely a class that returns the string to the SOAP servlet. That's all there is to it. More interesting Web services require more interesting code. See 1.5.2, “Categories of Web services” on page 7, for a list of Web service possibilities.

The Web service concept has gained so much attention, including from iSeries shops, because they provide a handy way to Web-enable existing legacy code such as RPG. The SOAP servlet can call a Java class that uses JDBC or the IBM Toolbox for Java (for example, CommandCall, ProgramCall, or ServiceProgramCall) to access legacy code and data.

We recommend that you start with Web services by focusing only on SOAP and WSDL where you control both requestors and providers. By skipping the broker (for example, skipping

UDDI and the publish and find steps), you don't have to dive into the deep end of Web services just yet. In this approach, the service requestor is bound to a predetermined service provider. For examples, see:

http://xml.apache.org/soap

This means that you must first deploy Web services inside your business or between a well-defined set of business partners. This strategy eases the impact of change as you learn and experiment.

You can publish your Web service to a private or public UDDI registry later.

1.4 Substance or hype

Are Web services just another good idea that will come and go with little impact? In this rapid-fire IT world, no one can make guarantees. But we can make an informed evaluation. There are many reasons to believe Web services will be a part of our future.

First, Web services specifications fill an existing need. Prior to SOAP, WSDL, and UDDI, e-commerce applications brought many different approaches to connect buyers, suppliers, marketplaces, and service providers. These new Web technologies enable programs to find and communicate with each other on a global scale without large investments in technology infrastructure.

Second, Web services are built on, or are similar to, numerous proven architectures and technologies. All major distributed architectures include:

- A distributed or remote procedure call mechanism (such as SOAP)
- An interface description language (such as WSDL)
- A central registry (such as UDDI)

Web services themselves are built on standard technologies including TCP/IP, HTTP, SSL, and XML.

Third, some of the biggest players in IT are behind the Web services initiative. IBM and Microsoft cooperated on the SOAP specification by merging new technologies that each company was developing. IBM, Microsoft, and Ariba were the founding members of UDDI.org, which has now grown to more than 220 supporting organizations. At JavaOne 2001, Sun promoted Web services as a major theme. The e-business XML (ebXML) organization recently dropped its own XML-based RPC protocol and adopted SOAP. For more information, see:

http://ebxml.org

If this many disparate companies and consortiums are in agreement, there's a good chance Web services will succeed. Regardless of whether SOAP, WSDL, and UDDI are the best solutions in technical terms, what matters is that the major IT players agree upon standards that facilitate communication, interoperability, and integration.

Fourth, there's a lot of code behind Web services, which represents a large developer investment. Apache has its SOAP open-source group, and IBM and Microsoft offer free Web services toolkits for download. The public UDDI Business Registry is up and running.

Finally, Web services have a low barrier to entry. If you use one of the free toolkits, your first Web service can be up and running in hours. Remember that a low barrier to entry is one of the reasons the HTML-based Web originally grew so quickly.

Admittedly, the Web service architecture isn't complete, fully implemented, or shrink-wrapped. It's an emerging set of technologies and protocols, and some of the higher-level functions aren't yet sufficiently evolved. For a look at some of the innovations in the works, see the ebXML work from OASIS at:

http://www.oasis.com

The functionality is available to put Web services to work for your organization today.

1.5 Business value

You can convert just about anything you currently find on the Web into a Web service by:

- Writing a Web service implementation
- Generating the WSDL to describe your Web service interface
- Optionally publishing the interface to a UDDI broker

For a Web service to be useful and successful, it needs to fill a need. Answering these questions can help you understand the value of converting a legacy application to a Web service:

- Does the legacy application implement a business function that can be valuable to other businesses?
- Can you lower costs by integrating a business process with a business partner?
- Can you improve a business process?

1.5.1 Web service provider types

Most Web services can be grouped into two categories:

- **Information source**: Web services that provide information to consumers and other companies. Examples of information source Web services are a stock quote service, a news clippings service, or a catalog order service.
- **Integrator**: Web services that integrate the function of multiple Web service. Examples of an integrator service may be a credit check service, a travel agent service, or an auction Web service.

The integrating Web services are an interesting possibility. Of course, they can only exist when information source Web services become available.

1.5.2 Categories of Web services

The following list of Web services categories includes sample business propositions for each category. Note that a given Web service may fall into multiple categories:

- Web services that specialize in large, slowly changing databases. These services provide organized access to data and keep that data current.
  - Geographical data (for example, MapQuest)
  - Credit checks based on credit history
  - Libraries or catalogs of audio or video recordings, books, or electronic documents
- Web services that specialize in rapidly changing databases. These services publish content and ensure that content is current.
  - News, weather, and sports
  - Stock quotes

  - Currency conversion rates
  - Current status of manufacturing line reports (internal)
- Web services that specialize in customer-specific data. Users select a broker once and upload their information to the selected service for later access.
  - An outsourced general ledger or payroll service
  - An employee telephone listing service
  - Personalization services that offer customers a tailored set of information based on their surfing histories
  - The monitoring of spending histories to detect suspicious credit card transactions
- Web services that provide a complex algorithm or other transformation.
  - Cryptographic encryption and decryption
  - Algorithmic solutions to business problems (for example, traveling-salesman problems)
  - A complete configuration for a computer or a car that satisfies a set of constraints
  - Transformation services to Wireless Markup Language (WML) or other kinds of transcoding
- Web services that provide standardized access to non-Web service services.
  - Sending and receiving e-mail, voice mail, or text messages
  - Checking the current status of a mailed package
  - Viewing documents that are part of a company's financial system
  - Processing a credit card payment
- Web services that integrate data from multiple Web services.
  - Portals
  - Shopping directories from multiple vendors that help users find the best deals
  - A concierge service that locates and integrates multiple personal services into a single package
- Web services that provide particular brand-name “approvals”.
  - A specific stockbroker's recommendations
  - Consumer watchgroups' survey results
  - A specific Certificate Authority's (CA's) digital signature

2

Understanding the sample RPG application

This chapter introduces the RPG applications that form the base for most of our Web services. There are two basic applications:

- **The Loan Service application**: This application represents a type of application that can readily be converted to a Web service. It has a simple user interface component that is separated from the business logic. This allows you to call the business logic module directly.
- **The Order Entry application**: This application represents a different type of application, where we use the experience of the RPG programmer to construct the logic of the application, but it is purposely written to act as part of the Web service.

In addition, we created an RPG-based socket server to act as a proxy between the Java code and the RPG code.

Note: You can find the instructions for downloading and installing the sample applications in Appendix A, “Additional material” on page 61.

© Copyright IBM Corp. 2003. All rights reserved.

2.1 The legacy RPG applications

This section covers the legacy RPG applications that form the basis for our Web services. Each of the legacy applications are written in basic ILE RPG.

2.1.1 The Loan Service legacy application

The loan service is a simple representation of a credit application system. A customer can apply for a loan or a line of credit, and the application either accepts or rejects the loan or line of credit based on the salary of the applicant and past loans. The maximum total loan amount for any one applicant is 10% of their annual salary. If a line of credit is established, future loan applications can be automatically approved.

The structure of the RPG programs for this application is nicely separated between user interface and business logic. Our simple program (APPL02) provides the standard 5250 user interface to the application. And we have another program (APPL01) that provides the business logic.

An application of this nature lends itself naturally as a Web service. The program containing the business logic can simply be called as part of the application.

Executing the loan application using the 5250 terminal interface

You can execute the loan application from an OS/400 command line using the following two commands:

```
ADDLIBLE ITSOWS
CALL GETLOAN
```

The application user interface appears as shown in Figure 2-1.

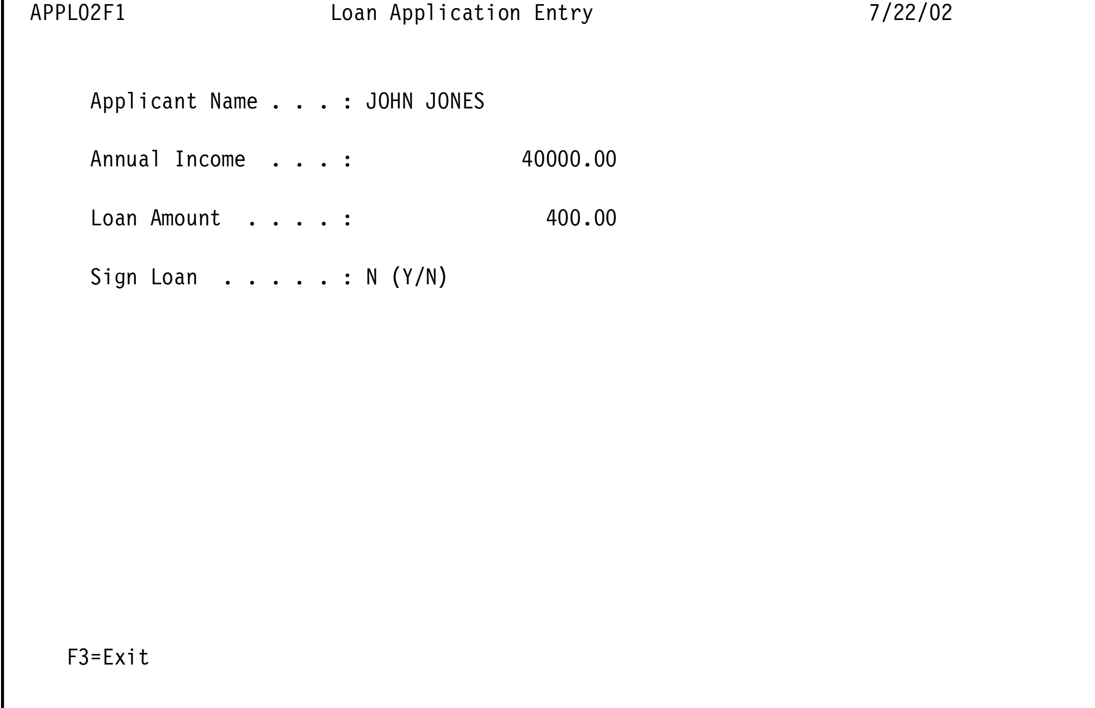

Figure 2-1 Loan Application Entry display

To apply for a loan, simply enter a customer name, an annual salary, and a loan amount, and then press Enter. If the loan amount falls within the limits, the loan is approved as shown in Figure 2-2.

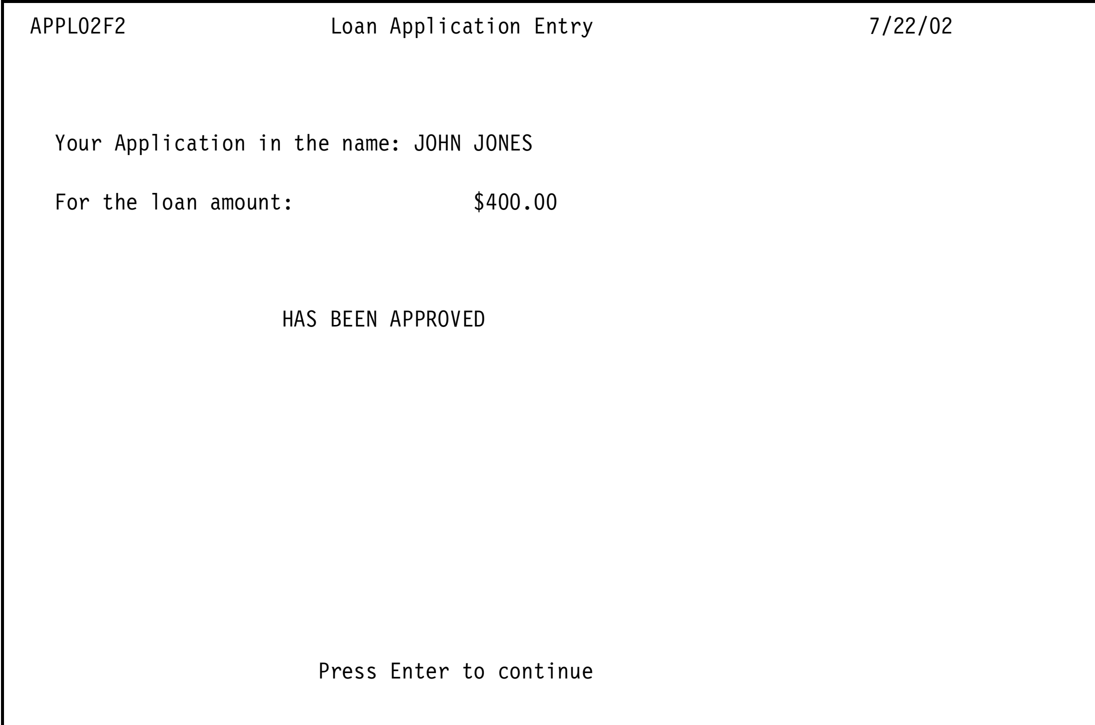

Figure 2-2 Loan approval

2.1.2 The Order Entry legacy application

The Order Entry application is intended to illustrate a classic RPG application where the user interface and the business logic is integrated into one program. Applications of this nature require some work to convert into Web services. The programmer is required to write a specific implementation of the application for use by the Web service.

Even though this approach means that it may seam easier to write the entire application in Java, it has the following benefits:

- It is possible to simply replicated blocks of logic, conserving time and effort.
- Calculations are consistent between the legacy application and the Web service.
- RPG programmers can contribute to the project without retraining.

Executing the Order Entry Application using the 5250 terminal interface

The 5250 interface for the Order Entry application is deliberately incomplete. We have restricted the user interface to a simple product search. This product search is not replicated into the Web service. Instead we provide a different methodology to achieve the same functionality, but both are based on RPG code.

You can run the loan application from an OS/400 command line using the following two commands:

```
ADDLIBLE ITSOWS
CALL PROD02
```

The application user interface appears as shown in Figure 2-3.

```
PROD02C1          Novelty Furniture Store Limited              7/22/02

                           PRODUCT SEARCH

  Enter Search Word . . .: BEAN


  Product Description                      Price
  BEAN BAG, GREEN                                        $40.00
  BEAN BAG, YELLOW POLKA DOT                             $45.00
  BEAN BAG, PURPLE                                       $40.00
  BEAN BAG, YELLOW AND PINK                              $40.00





                                                                    +




   F3=Exit
```

Figure 2-3 The product search

You can search for any given product by entering a partial description and pressing Enter.

2.2 The Web services implementations

This section describes the actual conversion from RPG applications to Web services. There are several different ways to achieve this. There are also a number of problems that act as the drivers for how it is best done.

2.2.1 Thread scoped versus job scoped

The main issue facing any programmer, intending to convert an RPG programs to a Web service, is the fact that RPG by default is not thread-safe. If two or more Java threads call the same RPG program at the same time, there is a good chance the result is garbage.

RPG IV has a thread-safety option that you specify on your H spec:

```
THREAD(*SERIALIZE)
```

When you code this, the RPG compiler ensures that only one thread is ever active in one module, so the static storage in the module is protected. You still have to ensure that everything you call is thread-safe and that you handle shared storage (such as IMPORT/EXPORT fields) in a thread-safe way.

The serialization of the calls to the module can cause some problems:

- The RPG module can become a bottleneck if the procedures are long running.
- You can get deadlock if you have a procedure in module A that calls a procedure in module B, and module B has a procedure that calls a procedure in module A. If thread1 is

in module A, thread2 is in module B, and they try to call to the other module, both threads will wait forever.

The RPG IV runtime is thread safe. RPG II and RPG III (other names are S36 RPG, OPM RPG, and RPG 400) are not thread safe.

It's not possible to have thread-scoped files using RPG's file support. If the file is left open, the next thread takes the old file state. It is possible to have some of the storage thread-scoped by using allocated storage or userspaces. The program has to know which thread is active (via a parameter from the caller) to see which basing pointer or userspace to use.

To perform thread-scoped I/O, you have to use the C runtime I/O functions, which are handle-based.

2.2.2 Calling an RPG program from Java

There are a couple of different ways you can call a RPG program from Java. You can use a simple JNI native call, or you can use Program Call Markup Language (PCML) through the IBM Toolbox for Java.

PCML is the simplest way to create a Java Bean that interfaces directly with a native iSeries program. WebSphere Development Studio Client for iSeries (Development Studio Client) even has a program call wizard, which makes the task of building a PCML Java Bean very easy. The drawback to using PCML is mainly performance, which can be a problem for a heavily used Web service. PCML still needs to handle the thread-scoping issue and is an interpreted language.

2.2.3 The socket server

To solve the thread and performance issues, we choose to implement an RPG-based TCP/IP socket server instead. By using a socket server, we have a central server controlling access to our RPG program. The Java-based Web services now communicate with our RPG program using TCP/IP. The socket server delegates each incoming request to a separately submitted job, running the RPG code. This guarantees one job per thread and that a thread can make multiple calls to the RPG program in sequence without compromising thread safety.

We found sample source for an RPG-based socket server on the Web at:

http://www.netshare400.com

We used this sample code in our conversion of the Loan Application and Order Entry Web services. In our implementation, we have the main RPG socket server (MAIN1) listening on port 27850. When a request comes in, the socket server locates a new available port and submits a new RPG program to batch, which listens on this port.

2.2.4 The Loan Application Web service implementation

The Loan Application Web service consists of two new modules. APPL03 is the RPG program that directly converts TCP/IP messages into RPG program calls. It also has the ability to provide a list of available loans. This is a feature that the standard Loan Application program doesn’t have.

The main features of the Web service are implemented by calling the RPG program APPL01, which is the same program that the legacy application calls.

There is also a special socket server (APPL04) spawned by MAIN1, for servicing loan applications. The spawned socket server manages conversion between ASCII and EBCDIC and calls APPL03 for the execution of each message.

2.2.5 The Order Entry Application Web service implementation

Just like the Loan Application Web service, the Order Entry Application Web service consists of two new modules. PROD03 is the RPG program that directly converts TCP/IP messages into RPG program calls. It manages the retrieval of products and product groups, placing an order and the shopping cart items.

Again, there is also a special socket server (PROD04) spawned by MAIN1. The spawned socket server manages conversion between ASCII and EBCDIC and calls PROD03 for the execution of each message.

You can look at the source code after installing the sample applications on the iSeries server.

2.2.6 Possible improvements in the code

The code samples provided here are deliberately brief and simple. There are many possibilities for improving the performance of the code that have been omitted for expediency and clarity of code. Some example are:

- Implementation of a connection pool to re-use spawned RPG programs

  A simple stack can be used to keep a list of already started spawned socket servers. This allows current instances to be re-used between threads.

- Returning multiple items in the same TCP/IP message

  Each TCP/IP message has the fixed length of 1 Kb. In the case of a product search, we only return one product for each call. It makes sense to return as many items as possible within the 1 Kb limit by using an array.

3

Web services application design and architecture

This chapter contains an overview of Web services design and architecture. It explains the following topics:

- Web services design considerations
- RPG Web services application architecture
- Store Web service application architecture

© Copyright IBM Corp. 2003. All rights reserved.

3.1 General design guidelines

A Web services application can have three participants:

- A service requestor
- A broker
- A service provider

This chapter describes the design process for Web service provider. Chapter 5, “Web service client” on page 45, describes architecture approaches for service requestors. The service broker (UDDI) architecture is defined by various open source organizations and is not covered in this book.

3.1.1 Getting started

Consider what assets you currently have and where you want to start development. There are three basic approaches:

- **Top-down**: Web service development starts with defining a Web service interface (WSDL) or using an existing WSDL followed by choosing an implementation language and platform.
- **Bottom-up**: Existing applications are converted to Web services by adding functionality to handle SOAP requests and defining WSDL.
- **Meet in the middle**: This is a combination of the other two approaches.

The bottom-up or the meet-in-the-middle approaches are good choices for companies that want to use existing applications as a Web service implementation. The design process starts with evaluating existing applications and deciding if it is feasible to convert them to Web services.

3.1.2 Architectural issues

Consider any architectural issues and guidelines. If you are starting with an existing body of code, you may have to restructure that code to make it suitable for further work. Important architectural issues to consider are:

- **Ensure there is a clear separation between the business logic and the presentation logic**: For a Web service implementation, the SOAP processing is the “presentation” logic.
- **Stateless versus stateful applications**: Stateful applications may have complex control flow and require a greater effort to convert to a Web service. Try to remove all state information from service provider code.
- **Self-containment**: Build components that can operate independently of other programs.
- **Reusability**: Build components that can be used in more than one application. This can be a difficult problem, in general, which is often largely due to business rather than technical issues.
- **Composition**: Develop components that can be combined into applications.

3.1.3 Implementation issues

Consider any implementation issues. The interface to Web services are based on such industry standards as HTTP and XML. This means that Web services are platform and language independent, but there are still many options. Consider the following important implementation issues:

- Choose an appropriate implementation platform and language. Any system that has the ability to process HTTP, XML, and SOAP can implement Web services. However, this is easier in some languages that others. Java is particularly well-suited to these tasks. We show how to combine legacy applications with the new technologies to create a Web service.
- Choose parameter types. Table 3-1 shows the types that are already supported by SOAP and literal XML encoding. If you need to use other types, you have to write custom SOAP serializers and deserializers.

Table 3-1 Types and mappings for SOAP and Literal XML encoding

Type	Encoding	Mapping support
Java primitive types such as int, float, and so forth, and their wrappers (java.lang.Integer, java.lang.Float, and so on)	SOAP	Default, custom
org.w3c.dom.Element	Literal XML	Default
org.w3c.dom.Element	SOAP	Custom
java.lang.String	SOAP	Default, custom
java.util.Date java.util.GregorianCalendar	SOAP	Default, custom
Java arrays java.util.Vector java.util.Hashtable java.util.Map	SOAP	Default, custom
java.math.BigDecimal	SOAP	Default, custom
javax.mail.internet.MimeBodyPart java.io.InputStream javax.activation.DataSource javax.activation.DataHandler org.apache.soap.util.xml.QName org.apache.soap.rpc.Parameter	SOAP	Default, custom
java.lang.Object (a deserializer for null objects only)	SOAP	Default, custom
other	any	Custom

3.2 Service requestor architecture

A Web service requestor sends SOAP messages to a provider and receives SOAP messages in return. A Web service requestor can be a:

- Web application
- Legacy application running on iSeries or another platform
- Visual Basic desktop application
- Another Web service

3.2.1 An application is a requestor

The simplest architecture for the requestor is an application, which is running on the requestor's machine, that builds a SOAP request, sends it to the provider, waits for and receives the SOAP response, and then interprets the response (see Figure 3-1).

The application can provide either a text or GUI user interface. Note that the interface between the requestor and the provider is a public interface based on SOAP, but that the interface between the requestor and the user interface is private to the application.

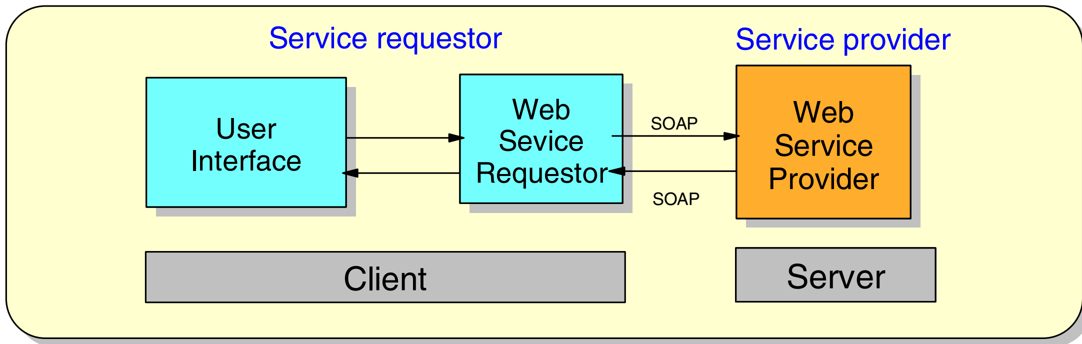

Figure 3-1 Requestor is an application

3.2.2 An application server is a requestor

Another likely situation is to use Web services in the deeper layers of the architecture, and to use a standard browser for the user interface. This prevents every user from having to install a special application on their client computer. In this case, an application server is written that interacts with the client's browser and contains a requestor to interact with the Web service. In this architecture, the application server acts as a bridge between the browser and the provider (see Figure 3-2).

The interface between the requestor and the provider is public and based on SOAP. The interface between the browser and the application server is also public, but is based on HTTP. The interface between the requestor and the rest of the application server is private.

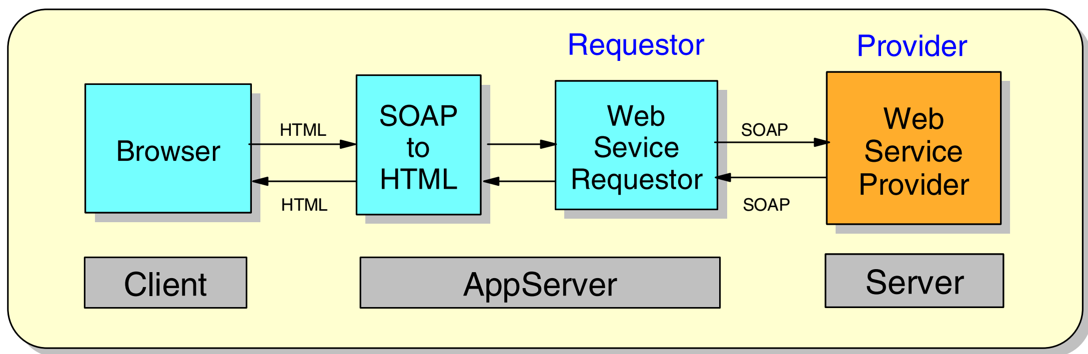

Figure 3-2 Application server is a requestor

3.2.3 Integrating multiple Web services

An integrating Web service is both a provider to some requestors and a requestor to other providers. An integrating Web service combines the functions of other Web services and creates a new and different service (see Figure 3-3).

For example, a Travel Agent Web service requests the services provided by an Airline Reservation Web service, a Hotel Reservation Web service, and a Car Rental Web service. The Travel Agent integrates these services into one service and provides a complete travel reservation service to its requestors.

The interface between requestors and provider is public and based on SOAP.

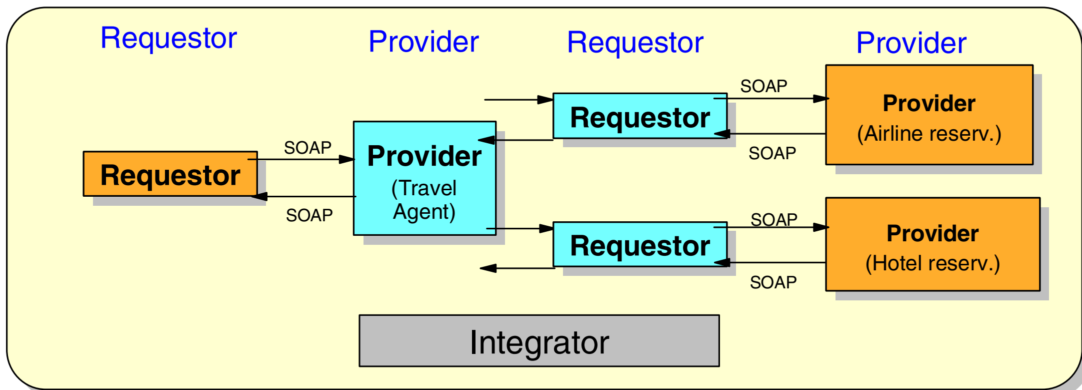

Figure 3-3 Integrating multiple Web services

This is one of the most interesting architectures of all and really begins to show off the power and capabilities of Web services. Since integrating Web services can only be written after other Web services are available, integrating Web services do not emerge as quickly, but they do develop.

Now that we have covered the general architectures of Web service requestors, let's move on to the general architectures of Web service providers.

3.3 General service provider architecture

This section focuses on the architecture of the service provider. Since all the interactions between the service requestor, service provider, and broker use SOAP calls, our architecture needs to clearly identify how SOAP calls are handled. We use the interaction between a service requestor and a service provider to describe this interaction.

First, we must decide on the communications protocol used between the service requester and service provider. There are a number of protocols that have been used in the past including HTTP, SMTP (e-mail), and messages (for example, WebSphere MQ or JMS). HTTP is the most common choice since it allows requestors and providers to communicate over the Internet. It is also the choice we make in this book. Therefore, our first architectural decision is whether to implement the service provider as a Web server or as part of an application server.

SOAP messages are XML documents that can contain a lot of highly-structured information. Rather than requiring every Web service to duplicate the work required to parse and interpret SOAP messages, the next step in the architecture is to separate out, as much as possible, the code that handles the SOAP message (the SOAP Server piece) from the rest of the service provider implementation (the implementation piece). Fortunately, this separation is not difficult to achieve and leads to the architecture shown in Figure 3-4. The SOAP server should accept incoming SOAP message requests over HTTP, parse the SOAP message, and pass the information to the Service Provider Implementation.

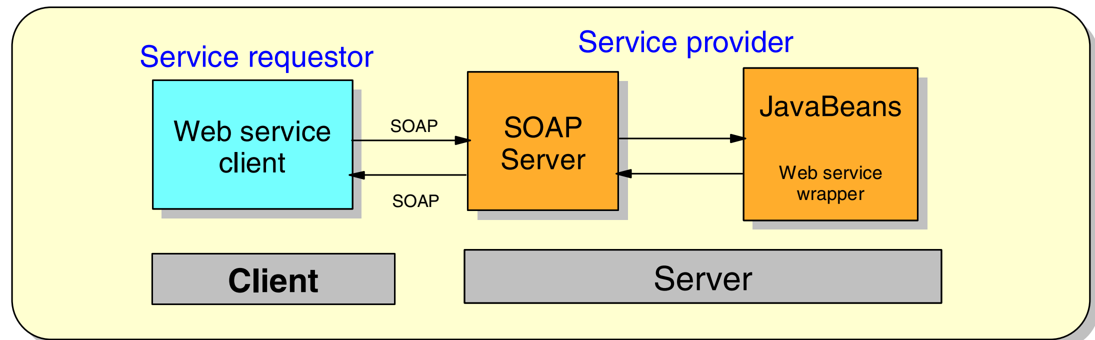

Figure 3-4 General service provider architecture

Note that while the interface between the requester and the SOAP Server are defined by SOAP, the interface between the SOAP Server and the Service Provider Implementation are *not* covered by any Web service standard. This definition of this internal interface does not matter as long as the SOAP Server and the Service Provider Implementation agree on it.

We recommend that you write the service provider implementation piece in Java because of its ease in working with XML, its portability to multiple application servers, and because tools exist to help build the necessary interfaces. There are existing standards that should be considered for this interface. The two obvious choices are:

- **Java Bean**: A Java Bean is essentially any Java class that has a zero argument constructor and getter/setter methods for the instance variables.
- **Enterprise JavaBean**: If the Service Provider Implementation is implemented as a Session EJB, then the EJB container provides the bean with security and transaction control.

The choice between these interfaces may be based on whether transaction control is required (at this layer), the skill level of the developers, or even personal preference.

Choosing either a Java Bean or EJB interface means the SOAP server has the following responsibilities:

- Convert parameters in the SOAP XML request into Java data types (often called *deserialization*).
- Create and find the appropriate Java Bean or EJB instance.
- Call the proper method.
- Convert the returned value (if any) from a Java data type into a SOAP XML response (often called *serialization*).

The Development Studio Client uses a SOAP server from Apache's SOAP project (`http://xml.apache.org`), which supports implementations written as Java Beans, EJBs, or Bean Scripting Framework (BSF) scripting language. The SOAP server has built-in mappings between SOAP and Java for widely-used parameter types. It also defines a generic mechanism (serializers and deserializers) for handling other types.

This basic architecture for service providers still leaves plenty of freedom in how a service provider can be implemented. However, it separates the majority of the SOAP processing from the service provider-specific implementation. This separation is the same as the separation of the model from the view/controller (or presentation) in a Model-View-Controller (MVC) architecture.

3.4 Service provider architecture for RPG applications

The basic architecture works well for such languages as Java that interface well with Web servers. But how can we extend the basic architecture to support legacy languages such as RPG? We have to further subdivide the service provider implementation into two pieces:

- A Java piece that communicates with the SOAP Server
- An RPG piece that contains the existing business logic

How should we bridge between the Java piece and the RPG piece?

One possibility is to use the Java Native Interface (JNI) support that allows Java to call RPG. This will work functionally, but it will not scale. For all but a few Web services, providers must plan for multiple simultaneous requests, which requires multiple jobs or threads. RPG was not originally designed to be multi-threaded, and some of its functions are inherently non-thread-safe. Recent changes to the RPG run-time allow RPG programs to be run in a multi-threaded environment, but all access is serialized so that only one thread at a time can run an RPG procedure. That is, multiple threads can call an RPG program, but only one is allowed to run at a time, while all the rest wait. This feature of RPG prevents JNI calls from Java to RPG procedures from scaling.

Since multiple threads will not work, multiple RPG jobs are required. If the RPG procedures are run in separate jobs (from the application server), then some mechanism must be used to communicate between the application server job and the multiple RPG jobs. There are several options here including JDBC, DataQueues, WebSphere MQ, or messages sent over TCP/IP sockets. In our experience, the fastest method of communication between the application server and the RPG jobs is sending messages over a socket. The contents of the socket messages are characters, so integers are sent as the characters “123”, rather than as a binary-encoded value.

For the same reasons that we split the Web service provider into a SOAP server and a service provider implementation in the basic architecture (see Figure 3-4), we want to further split the RPG piece into a socket server that handles most of the details of the socket communication and the business logic. This separation is not difficult. Both of these pieces are written in RPG. The business logic is packaged into ILE RPG procedures and bound into *SRVPGMs. The socket server is a fairly generic ILE RPG *PGM that calls the proper procedures in the *SRVPGM.

Adding this to the generic architecture gives a service provider architecture for Web services that call RPG procedures as shown in Figure 3-5.

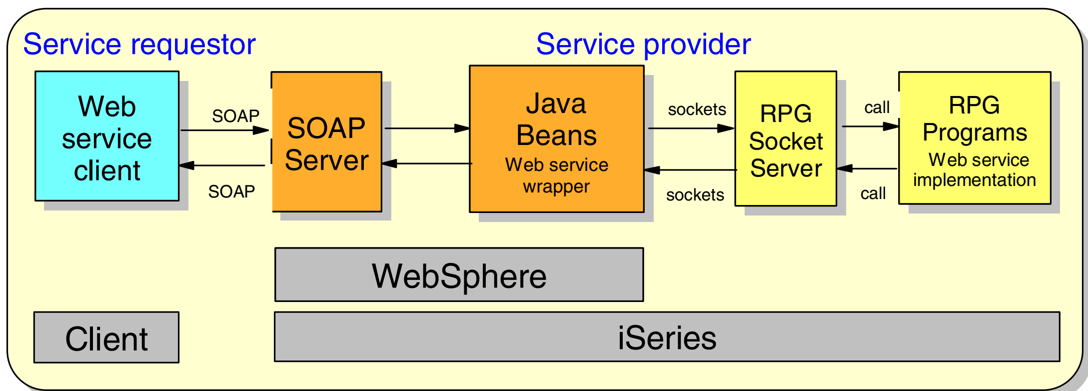

Figure 3-5 Web services architecture for accessing RPG applications

Another task is to create Web service deployment descriptor files. These files must be created manually if a Web service is implemented in RPG.

3.5 Our architecture

Given the general architecture for Web services that include access to RPG applications, we need to fill in several details. We choose a specific set of products and technologies to implement our Web services:

- We write a Java Bean for each of our three Web services: Store, Order Entry, and Loan. The Store Web service is an integrating Web service. It combines the Order Entry and the Loan Web services. The Store Web service contains business logic of an online retailer application.
- We use WebSphere Application Server 4.0, which includes a SOAP server. The SOAP server consists of servlets and JSPs running in a WebSphere application server that listen for incoming requests from the service requester. When a request arrives, the SOAP server searches through its list of registered Web services and routes the request to the specified Java Bean for the Web service. We take advantage of the deployment tools to help build and register the Web service Java Beans.
- The Web service Java Beans construct messages and send them over TCP/IP sockets to the RPG socket server.
- We use the RPG socket server. The socket server receives the message, converts it to a usable format, and forwards the request to one of the RPG business logic programs.
- The RPG business logic procedure processes the message and generates the response data.
- The response data is passed back through the chain and results in a SOAP response being sent back to the Web services requester.

On top of this basic architecture for RPG application, we need to consider several other questions and options.

3.5.1 System configuration

Of course, the RPG socket server and the RPG business logic pieces must run on an iSeries server. But the application server, the SOAP server, and the Java Beans pieces do not contain any code that is specific to the iSeries. Therefore, they can run on iSeries or any other platform. Even if these pieces run on an iSeries server, they can run on the same machine or a different machine as the RPG pieces.

Maintenance is usually easy if both pieces are on the same iSeries machine. However, if scalability is a concern, spread the workload over multiple machines.

3.5.2 User interface

How is the user interface handled? The Web requester is a Web application that interacts with the Store Web service, converts the data to HTML, and sends it to the user as an HTML page. None of the Store, Order Entry, or Loan Web services have a browser-based interface.

For example, how does the Order Entry Web service contribute to the presentation of the items to the end user, even though it is far removed from the Web application that will generate it? Here are a few possibilities:

- Order Entry returns a visual, presentation-ready block of XHTML to the Store Web service, which passes it on to the Web application. This is probably not the ideal solution since it definitely mixes model and viewer/controller in a the deeply nested Order Entry Web service.
- Order Entry returns a data-only (model) block of XML to describe the information about a catalog item. This may be the solution that most people would choose since there is a clear separation between the model and viewer/controller. However, it requires the requester Web application to construct its own presentation from scratch. Considering that Order Entry is provided by a separate business entity who may add or change data from time to time, they are forcing all their customers to change with them.
- Order Entry provides the same data-only block of XML as described in the previous choice and provides an optional URL to an XSL style sheet that can be used to convert the data-only block of XML to XHTML. The stylesheet provides Order Entry's default recommendation for the presentation. Providing an optional stylesheet allows the Web application to either:
  - Use the Order Entry's prepackaged XSL stylesheet to create the presentation
  - Provide its own customized XSL stylesheet to create the presentation.

Note that these decisions affect both the top presentation layer (the Web application) and a deeply nested Web services (Order Entry). Option 3 is slightly more complicated than option 2, but it gives the Web application (as well as the Shop service) the ability to optionally modify Order Entry's recommended presentation.

A similar situation occurs for the presentation of Order Entry's item images. The Web application should not have to store and maintain a copy of all the item images. Rather, Order Entry should provide one, or even multiple, image URLs of each item. The Web application can choose to use or ignore the image files available on Order Entry's Web site.

Yet another instance of this problem exists for form validation. Order Entry ultimately defines the validation rules for a catalog item. These rules need to be propagated through the Store service and out to the Web application's user interface. This may be solved by Order Entry providing the URL of an optional JavaScript file that contains the validation rules for the various fields. Single-field validation routines are fairly straightforward. However, things are more complex for multi-field validation routines since all of the fields may not be displayed on the same HTML form. Of course, a JavaScript file may be useless if Order Entry is used by a non-browser client.

These situations arise primarily because different Web services are provided by separate business entities. The architecture we are building attempts to design Web services that are as useful as possible to a number of different clients:

- Those who want simple and quick access to a Web service and are willing to accept a "sub-optimal" presentation
- Those who want complete control of the presentation and are willing to invest the extra effort to provide a highly polished presentation

We choose the simplest approach to keep the code as simple as possible. Our Web services run data-only XML, and the Web application completely determines the presentation.

3.5.3 Stateful versus stateless Web services

Stateless services are usually faster than stateful services because they do not have to incur the overhead of mechanisms to track a single user's interaction with the site.

The HTTP protocol is inherently stateless. Each request or response is unrelated to every other request or response. A stock quote Web service and a news clipping service are good examples of Web services that are likely stateless.

But there are services that must maintain their state between requests. A shopping cart application, such as the one we are building, is a good example. Application servers have "session" support so that all requests from a single user can be grouped together. This is important when building a shopping cart application. The user needs to make multiple requests to the application server to successfully identify a set of products and then order and pay for them.

We also need to determine which pieces must maintain their state. If the contents of the shopping cart are kept in the Java Bean, then the Java Bean is stateful, but the RPG portion may be stateless. Alternatively, the contents of the shopping cart can be maintained in a stateful RPG layer, while the Java Bean is stateless. These options need to be evaluated.

In our design, the RPG is stateful since it maintains the shopping cart. The Java Beans are essentially stateless. While it may be easier and more desirable, in many cases, to make the Java stateful and the RPG stateless, we choose this approach since it is more likely to match Web services that are built from existing RPG code. As you can see from examining the code, there is still state in the Java layer to track the user's progress through the operations.

3.5.4 Workflow management

Simple Web services, such as a Stock Quote Web service, have a simple process model: the entire interaction is a single request followed by a single response. But more complex Web services have more complex interactions, often requiring a complex sequence of requests and responses. Process management is also sometimes referred to as *workflow*.

One approach is for the nested services (Order Entry and Loan) to be stateless. This often means the size of each request and response grows so that they contain the existing state. For example, Order Entry could have a single requestor response method to obtain item information. To place an order for multiple items, it would use a single request or response where the request lists all of the items being purchased. This essentially moves the process management task from Order Entry to Store Web service.

But there may be cases where multiple Web services are stateful. Properly coordinating the user's progress through the state of all Web services being used is a non-trivial problem. Frameworks and approaches are being developed to address this kind of problem for Web services.

3.5.5 The Store Web service application architecture

When Web services are developed as self-contained components, they can be easily combined into more complex Web services. For example, we developed two independent Web services – the Order Entry Web service and the Loan Web service – and combined them into the Store Web service. The Store Web service is a service that handles typical online retailer requests such as:

- Displaying items for sale
- Placing an order
- Providing financing

The Store Web service application architecture is shown in Figure 3-6.

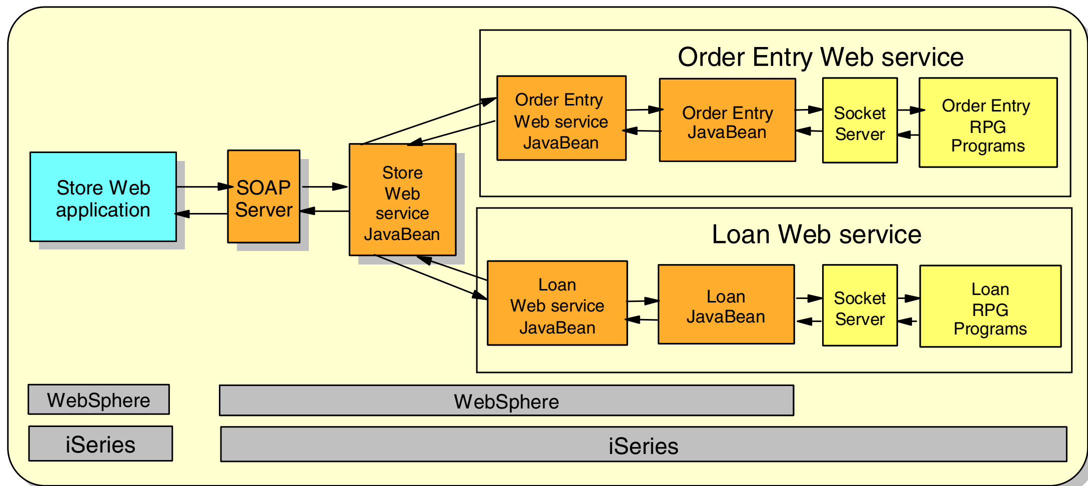

Figure 3-6 Store Web service architecture

We implemented each Web service independently for reusability purposes. This design gives us the flexibility to use the developed Web services in multiple applications. For example, the Order Entry Web service can be included in an order management and warehousing application, and the Loan Web service can be a part of a banking application.

The Store Web service client is a Web application that runs in WebSphere Application Server on iSeries. The Web application consists of servlets and JSPs that make calls to the Store Web service (see Chapter 5, “Web service client” on page 45, for more details).

The Store Web service is implemented as a Java Web service. The Store Web service combines two Web services – the Order Entry Web service and the Loan Web service – which are also implemented in Java. Each Web service uses a corresponding Java Bean to make calls to RPG programs. We discuss the implementation of each component in Chapter 4, “Implementing the Store Web service” on page 27.

To keep things simple in this Redpaper, we chose not to introduce a UDDI broker into the code. If you are trying to develop a general purpose Store Web service that can use different Order Entry and different Loan services, you need to extend this basic application.

4

Implementing the Store Web service

This chapter describes the process of implementing the Store Web service. The Store Web service is an integrating, statically bound Web service. This means that the Store Web service contains other Web services and the location and interface of these Web services is known at design time (UDDI is not used to locate the service).

The Store Web service was designed using the meet-in-the-middle approach. We defined the Store Web service interface based on business requirements of a potential Web service client. At the same time, we evaluated functionality of the Order Entry and the Loan RPG applications. Next, we modified the Store Web service interface to fill the gaps between the business requirements and functionality provided by RPG applications.

The bottom-up approach was used to implement the Order Entry and Loan Web services. We completed the following tasks to implement the Store Web service:

- Implemented RPG application access Java Beans
- Implemented Web service wrapper Java Beans
- Implemented the Store Web service Java Bean

This chapter describes the implementation of each component. It also discusses Generating Web service deployment descriptor files.

Note: You can find the instructions for downloading and installing the sample applications in Appendix A, “Additional material” on page 61.

© Copyright IBM Corp. 2003. All rights reserved.

4.1 Creating RPG application access Java Beans

Java Beans are Java classes that expose their functionality with a public interface. Over the past few years, Java Beans have become a standard component of a Web application architecture. Java Beans can be used in different kinds of applications: Web, Web services, and desktop (Swing-based applications).

RPG application access Java Beans contain code to access RPG applications. We refer to them as *access Java Beans* throughout the rest of this Redpaper.

4.1.1 Defining public interface for the access Java Beans

The creation of an access Java Bean starts with defining its public interface. The first step is to examine RPG programs that will be invoked from an access Java Bean and determine the required parameters. We created the following public interfaces to call the Order Entry and the Loan RPG programs:

- OrderClerkBean access JavaBean to access the Order Entry RPG application:
  - getCategories()
  - getCategoryItems(java.lang.String categoryId)
  - getItemDetails(java.lang.String itemId)
  - addItem(java.lang.String customerId, java.lang.String itemId, java.lang.String quantity)
  - getSelectedItems(java.lang.String customerId)
  - placeOrder(java.lang.String customerId)
- LoanOfficerBean access JavaBean to access the Loan RPG application:
  - establishLineOfCredit(java.lang.String customerId, double salary)
  - getLoans(java.lang.String customerId, double salary, double amount)
  - signLoan(java.lang.String customerId, double salary, double amount, java.lang.String loanId)

4.1.2 Connecting to an iSeries server

We use Java sockets APIs and a socket server (see Chapter 2, “Understanding the sample RPG application” on page 9) to establish a connection to an iSeries server. Example 4-1 shows the implementation of the `connect()` method.

Example 4-1 Implementation of the connect() method

```
// Method Description: establish the socket connection
public boolean connect(){
   boolean result = false;

   try {
      System.out.println("Connecting to AS/400");

   // Host is the IP address of the iSeries server
   // Port is the port number on which the socket server is listening
   socket1 = new Socket(HOST, PORT);
   PrintStream main =
            new PrintStream(new BufferedOutputStream(socket1.getOutputStream()), true);
      InputStream reply = socket1.getInputStream();

      // Form the request to start a new socket server
      //              Key Word       User      Password  Program   Library
```

```
      sendBlock(main, "new           ITSO      JAVA1A    PROD04    ITSOWS    ");
      String answer = readBlock(reply);

      if (!answer.equals("(INVALID)")) {
         // We have started a new socket server.
         // Acknowledge and get the new port number
         sendBlock(main, "(ok)");
         int port = new Integer(answer).intValue();

         // Open a connection to the new socket server
         System.out.println("Connecting to private socket server on port " + answer);
         socket2 = new Socket(HOST, port);

         // The socket available at this point should only be used by a single
         // session/thread at any one point. After each method call defined
         // in this class, the socket can be handed over to a pool for use by
         // other sessions/threads.

         // Use send and receive streams to make calls to RPG program methods
         sendStream = new PrintStream(new BufferedOutputStream(socket2.getOutputStream()),
                                      true);
         receiveStream = socket2.getInputStream();

         result = true;
      }

      } catch (IOException ioe) {
         // Log the error
         System.out.println(ioe);
      }

   return result;
}
```

Note: Connection pooling is the preferred method to establish socket connections to an iSeries server in the production environment. Connection pooling provides better performance and uses less server resources.

4.1.3 Calling an RPG program

RPG programs accept and return records in a fixed format. The length of each parameter is important in sending a request and processing results. Example 4-2 shows the implementation of the `getCategoryItems()` method. Notice that the returned parameters are read according to the fixed message format.

Example 4-2 Implementation of the OrderClerkBean access bean getCategoryItems() method

```
public Vector getCategories() throws Exception{

      String id = null;
      String description = null;
      Category category = null;
      Vector allCategories = new Vector();

      if(connect()){
         PrintStream send = getSendStream();
         InputStream receive = getReceiveStream();
```

```
         // Get all categories
         sendBlock(send, "FIRSTCAT");
         String data = readBlock(receive);
         while (data.substring(0, 4).equals("(OK)")) {
            id = data.substring(10,20).trim();
            description = data.substring(20).trim();
            category = new Category(id,description);
            allCategories.add(category);
            sendBlock(send, "NEXTCAT");
            data = readBlock(receive);
         }
      }
      // Close socket connections
      socket1.close();
      socket2.close();

      return allCategories;
}
```

See the entire source code of the sample applications for more details.

4.2 Creating Web service wrapper Java Beans

We can convert OrderClerkBean and LoanOfficerBean access Java Beans to Web services without building Web service wrappers. However, we decided to build the wrappers – OrderClerkService and LoanOfficerService Java Beans – to add an additional level of abstraction:

- OrderClerkBean and LoanOfficerBean access Java Beans contain only the logic to access RPG programs. They can be used in other applications that don't implement the Web services architecture (Web applications or Swing-based applications).
- Web service wrapper Java Beans implement the functionality to handle SOAP requests.

4.2.1 Defining a public interface for Web service wrapper JavaBeans

The public interface of Web service wrapper Java Beans is almost identical to the public interface of access Java Beans. The differences are in the input and output parameters for each method.

Web service methods can have parameters of three types:

- Java primitives (int, double, etc.) and java.lang.String class
- Parameters of type org.w3c.dom.Element
- Custom Java classes

Using each parameter type has its advantages and disadvantages. Using primitives doesn't require additional code to handle SOAP requests and makes the method signature self-explanatory. On the other hand, primitives can't be used if the method needs to return more than one value.

The parameter type org.w3c.dom.Element is the most universal type because it encapsulates any type and number of parameters. This type is a Java interface that represents an XML document. The main disadvantage of this parameter is the performance cost associated with parsing and writing XML.

Custom Java classes can be used as Web service method parameters. However, developers have to provide additional classes to handle mapping between the custom Java class and SOAP types. Mapping takes place when the SOAP server sends and receives SOAP messages to and from Web services. Both primitives and org.w3c.dom.Element parameter types use mapping provided by the SOAP server.

> **Note:** Refer to *Web Services Wizardry with WebSphere Studio Application Developer,* SG24-6292, for more information about parameter types supported in SOAP, encoding, and parameter mapping.

We use primitives and instances of java.lang.String as input parameters in all public methods of OrderClerkService and LoanOfficerService JavaBeans.

The output parameters of methods in OrderClerkBean and LoanOfficerBean access Java Beans are either primitives, instances of java.lang.String class, or instances of java.util.Vector class. All methods of the wrapper Java Beans have a return type of org.w3c.dom.Element. We chose org.w3c.dom.Element because of its flexibility. Using XML, we can return parameters of more than one type and include error messages.

Example 4-3 shows the implementation of the OrderClerkService `getCategoryItems()` method. In this method, we use the OrderClerkBean access Java Bean to retrieve category items. Then we create an XML document and write the contents of each Item object into an XML node "item". If an error takes place, the error message is returned as an XML document.

Example 4-3 Implementation of the OrderClerkService getCategoryItems() method

```
public Element getCategoryItems(String in_categoryId){

    Item item = null;
    Element root = null;

    try{
         // Get category items
         Vector items = orderClerkBean.getCategoryItems(in_categoryId);
         Iterator iter = items.iterator();

         // Get the top element of the XML tree
         root = getRootElement();
         doc.appendChild(root);

         while(iter.hasNext()){

            item = (Item)iter.next();

            // Create an XML node "item" for each Item object in the Vector
            Element el_item = (Element) doc.createElement("item");

            Element el_item_id = (Element) doc.createElement("item_id");
            el_item_id.appendChild(doc.createTextNode(item.getItemId()));
            // Add item_id element
            el_item.appendChild(el_item_id);

            Element el_item_name = (Element) doc.createElement("item_price");
            el_item_name.appendChild(doc.createTextNode(item.getItemPrice()));
            // Add item_price element
            el_item.appendChild(el_item_name);
```

```
                Element el_item_description = (Element) doc.createElement("item_description");
                el_item_description.appendChild(
                                                 doc.createTextNode(item.getItemDescription()));
                // Add item_description element
                el_item.appendChild(el_item_description);

                 // Add item element
                root.appendChild(el_item);

            }

     }catch(Exception e){
        // Return an error message
        root = handleError(e);
     }

     return root;
}
```

Example 4-4 shows sample XML contained by the return parameter of the `getCategoryItems()` method. A Web service client calling the `getCategoryItems()` method has to parse the output to retrieve the values returned by the org.w3c.dom.Element output parameter.

Example 4-4 Sample XML returned by the getCategoryItems() method

```
<response>
   <item>
      <item_id>1</item_id>
      <item_name>Conteporary couch</item_name>
      <item_description>Italian leather couch</item_description>
   </item>
   <item>
      <item_id>2</item_id>
      <item_name>Sleeper couch</item_name>
      <item_description>Couch with a pull-out bed</item_description>
   </item>
</response>
```

Table 4-1 and Table 4-2 show the input and output XML for all OrderClerkService and LoanOfficerService Java Beans public methods.

Table 4-1 Input and output XML for the OrderClerkService

Method	**Input parameters**	**Output XML**
getCategories()	N/A	<response> <category> <category_id>1</category_id> <category_name>Category 1</category_name> </category> ... </response>
getCategoryItems()	java.lang.String categoryId	<response> <item> <item_id>1</item_id> <item_name>Item name</item_name> <item_description>Item description</item_description> </item> ... </response>
addItem()	java.lang.String customerId, java.lang.String itemId, java.lang.String quantity	<response> <success>true</success> </response>
getSelectedItems()	java.lang.String customerId	<response> <item> <item_id>1</item_id> <item_quantity>3</item_name> </item> ... </response>
placeOrder()	java.lang.String customerId	<response> <order_number>12345</order_number> </response>

Table 4-2 Input and output XML for the LoanOfficerSrvice

Method	Input Parameters	Output XML

getLoans()	java.lang.String customerId,
	java.lang.String loanAmount,
	java.lang.String salary
		`<response>`
		`<loan>`
		`<loan_id>1</loan_id>`
		`<loan_description>Loan`
		`1</loan_description>`
		`<loan_term>6 months</loan_term>`
		`<interest_rate>5.25</interest_rate>`
		`</loan>`
		`...`
		`</response>`

signLoan()	java.lang.String customerId,
	java.lang.String loanAmount,
	java.lang.String salary
		`<response>`
		`<loan>`
		`<loan_number>10023</loan_number>`
		`</loan>`
		`</response>`

establishLineOfCredit()	java.lang.String customerId,
	java.lang.String salary,
	java.lang.String amount,
	java.lang.String loanId
		`<response>`
		`<success>true</success>`
		`</response>`

4.2.2 Generating Web service deployment files for wrapper Java Beans

Web services deployment requires several deployment descriptor files. We use Development Studio Client to generate the required files. You don’t have to complete the steps described in this chapter if you imported the Web service enabled version of the Store project (StoreWebService.ear).

Complete the following steps to generate a Web service:

1. In Development Studio Client, open the Java perspective be clicking **Perspective-> Open-> Java**.
2. Select the **Packages** tab and expand the **com.ibm.itso.webservices.order** package.
3. Select **OrderClerkService.java**.
4. Click **File-> New**.
5. Select **Web services** in the left part of the window and click **Web service** in the right part of the window. Click the **Next** button.
6. Select the **Generate a sample** check box. Accept the default values in the rest of the fields (see Figure 4-1). Click the **Next** button.

> **Note:** If you made a change to the OrderClerkService interface or if you previously generated Web service deployment files for it, select the **Overwrite files without warning** check box.

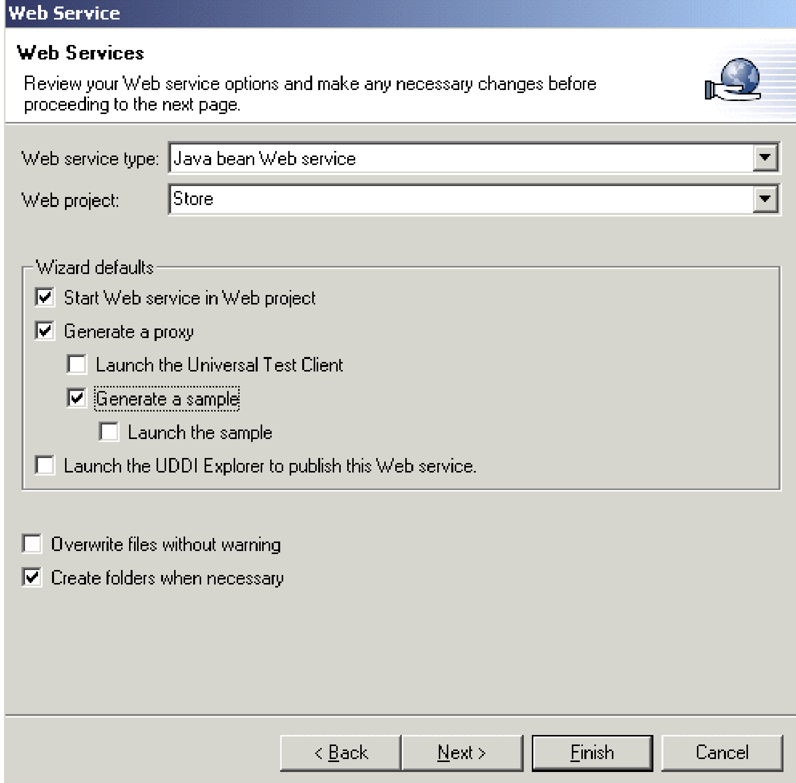

Figure 4-1 Web services wizard

7. Click the **Browse Classes** button (Figure 4-2).
8. Start typing `OrderClerkService`.
9. Select the **OrderClerkService** class and click **OK**.
10. Click the **Next** button.

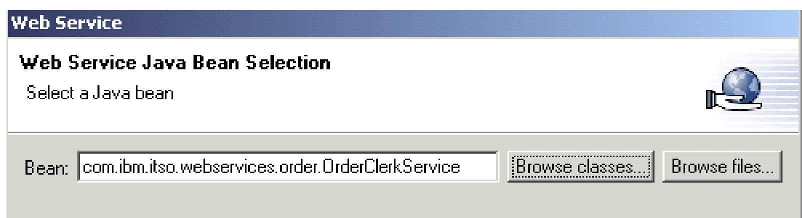

Figure 4-2 Web Service Java Bean Selection

11. Accept the defaults on the Web service Java Bean Identity Bean as shown in Figure 4-3. Click the **Next** button.

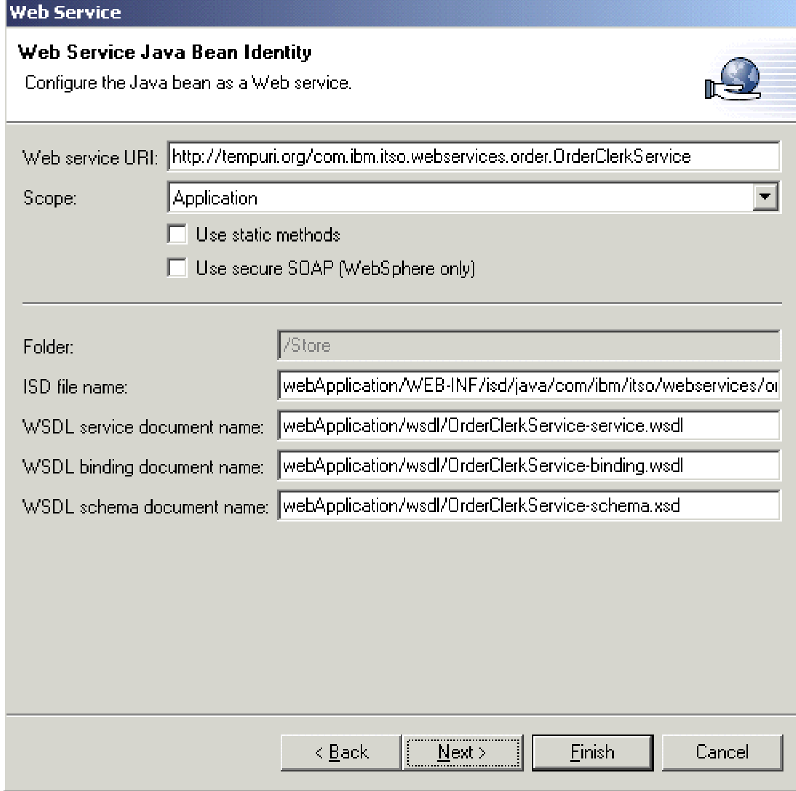

Figure 4-3 Web Service Java Bean Identity

12. Review Web service Java Bean Methods window. Five methods are deployed in the OrderClerkService:

    – getCategories()
    – getCategoryItems()
    – addItem()
    – getSelectedItems()
    – placeOrder()

    Notice that, depending on parameter type, the wizard selects a different encoding type for input and output parameters. SOAP encoding is specified for primitives and java.lang.String. Literal XML encoding is specified for parameters of type org.w3c.dom.Element (see Figure 4-4).

    Click the **Next** button.

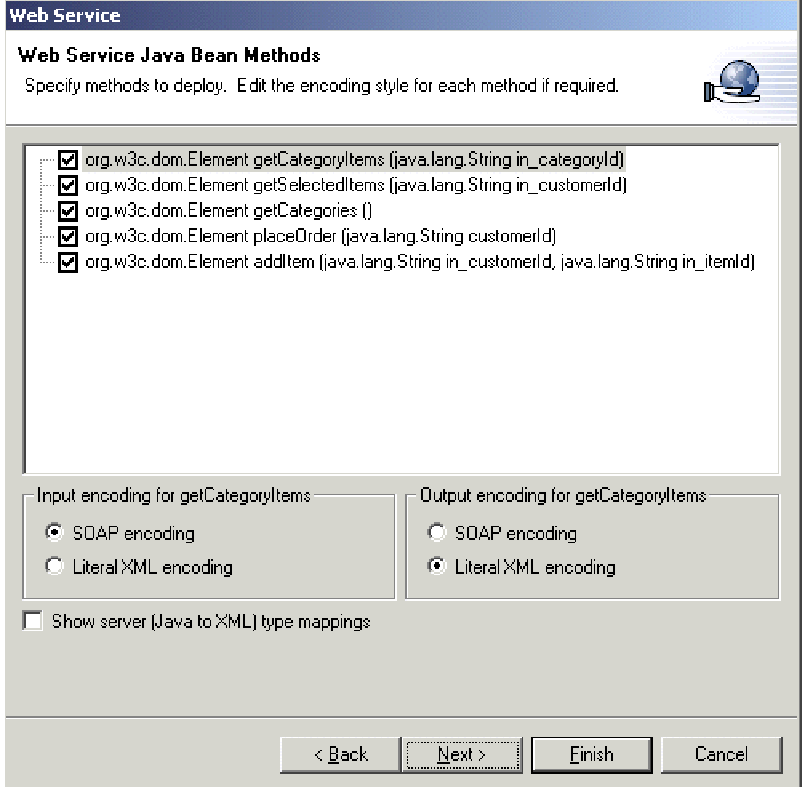

Figure 4-4 Web Service Java Bean Methods

13. Accept the defaults on the Web service Binding Proxy Generation window. Click the **Finish** button.

You have successfully prepared the Order Entry Web service for deployment. Complete the same steps to generate deployment files for the Loan Web service (LoanOfficerService.java) and the Store Web service (StoreService.java).

4.2.3 Files generated by the Web services wizard

This section reviews the files that are generated by the Web services wizard:

- WSDL files
- Java proxy classes
- Test JSPs

WSDL files

In Development Studio Client, open the Server perspective by clicking **Perspective-> Open-> Web**. Expand the **Store\Web Application\wsdl** folder. The folder has several files with the .wsdl and .xsd extensions as shown in Figure 4-5.

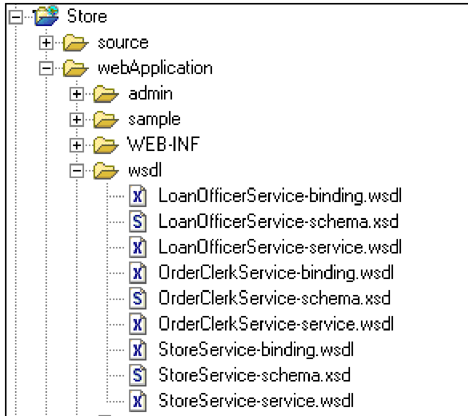

Figure 4-5 Store Web service WSDL files

The Web services wizard generated three files for each Web service:

- The service file that describes how to invoke the service
- The binding file that defines how to connect to the service
- The schema file that is used to validate incoming and outgoing SOAP messages

Open the **OrderClerkService-service.wsdl** file (see Example 4-5). This file specifies the location of the service interface file and the URL of the SOAP server (rpcrouter servlet).

Example 4-5 OrderClerkService-service.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderClerkServiceService"
    targetNamespace="http://localhost:8080/Store/wsdl/OrderClerkService-service.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"

xmlns:binding="http://www.orderclerkservice.com/definitions/OrderClerkServiceRemoteInterfac
e"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://localhost:8080/Store/wsdl/OrderClerkService-service.wsdl">
    <import
        location="http://localhost:8080/Store/wsdl/OrderClerkService-binding.wsdl"
namespace="http://www.orderclerkservice.com/definitions/OrderClerkServiceRemoteInterface"/>
    <service name="OrderClerkServiceService">
        <port binding="binding:OrderClerkServiceBinding" name="OrderClerkServicePort">
            <soap:address location="http://localhost:8080/Store/servlet/rpcrouter"/>
        </port>
    </service>
</definitions>
```

Next, open the **OrderClerkService-binding.wsdl** file. The *portType* element contains the name of the Java class implementing the service. Notice that each public method defined in the OrderClerkService is represented by the *operation* element in the wsdl file. Example 4-6 shows a partial file.

Example 4-6 OrderClerkService-binding.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    <import
location="http://localhost:8080/Store/wsdl/OrderClerkService-schema.xsd"
namespace="http://www.orderclerkservice.com/schemas/OrderClerkServiceRemoteInterface"/>
...
<portType name="OrderClerkService">
        <operation name="getSelectedItems" parameterOrder="in_customerId">
          <input message="tns:getSelectedItemsRequest" name="getSelectedItemsRequest"/>
          <output message="tns:getSelectedItemsResponse" name="getSelectedItemsResponse"/>
        </operation>
        <operation name="getCategories">
            <input message="tns:getCategoriesRequest" name="getCategoriesRequest"/>
            <output message="tns:getCategoriesResponse" name="getCategoriesResponse"/>
        </operation>
<operation name="placeOrder" parameterOrder="customerId">
            <input message="tns:placeOrderRequest" name="placeOrderRequest"/>
            <output message="tns:placeOrderResponse" name="placeOrderResponse"/>
        </operation>
        <operation name="getCategoryItems" parameterOrder="in_categoryId">
            <input message="tns:getCategoryItemsRequest" name="getCategoryItemsRequest"/>
            <output message="tns:getCategoryItemsResponse"
name="getCategoryItemsResponse"/>
</operation>
        <operation name="addItem" parameterOrder="in_customerId in_itemId">
            <input message="tns:addItemRequest" name="addItemRequest"/>
            <output message="tns:addItemResponse" name="addItemResponse"/>
        </operation>
</portType>
    ...
</definitions>
```

The Web service wizard also generated the SOAP deployment description files dds.xml and soap.xml. For more information about SOAP deployment descriptors, refer to *Web Services Wizardry with WebSphere Studio Application Developer,* SG24-6292.

Java proxy classes

The purpose of a Web service proxy class is to encapsulate the functionality of making a SOAP call. In Development Studio Client, open the Java perspective by clicking **Perspective-> Open-> Java**. Expand the folder **Store\source\proxy\soap\com\ibm\itso\webservices\order**. Open the **OrderClerkServiceProxy** class and switch to the **Outline** tab.

The OrderClerkServiceProxy class has all public methods defined in the OrderClerkService wrapper Java Bean. The org.apache.soap.rpc.Call class is responsible for making calls to the SOAP server. In each method representing the OrderClerkService interface, we set the properties on the Call object, invoke the service, and process output (see Example 4-7)

Example 4-7 Implementation of the OrderClerkServiceProxy.getCategoryItems() method

```
public synchronized org.w3c.dom.Element getCategoryItems(java.lang.String in_categoryId)
throws Exception{

String targetObjectURI =
"http://tempuri.org/com.ibm.itso.webservices.order.OrderClerkService";
```

```
    String SOAPActionURI = "";

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

    // Set the method name to call
    call.setMethodName("getCategoryItems");
    call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
    call.setTargetObjectURI(targetObjectURI);

    Vector params = new Vector();
    Parameter in_categoryIdParam = new Parameter("in_categoryId", java.lang.String.class,
in_categoryId, Constants.NS_URI_SOAP_ENC);
   // Set method parameters
    params.addElement(in_categoryIdParam);
    call.setParams(params);
   // The URL object contains the location of the SOAP server
    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 ((org.w3c.dom.Element)refValue.getValue());
    }
  }
```

The proxy classes are usually used by Web service clients and integrating Web services. In our sample application, the Store Web service uses generated OrderClerkServiceProxy and LoanOfficerServiceProxy classes. The Store Web service client makes calls to the generated StoreServiceProxy class.

Web service test JSPs

The Web service wizard generated JSPs that we can use to test our Web services. To use the test JSPs, follow these steps:

1. In Development Studio Client, open the Server perspective by clicking **Perspective-> Open-> Other-> Server**.
2. Expand the **Store\webApplication\sample\OrderClerkService** folder.
3. Select **TestClient.jsp**. Right-click it and select **Run on Server** (see Figure 4-6).

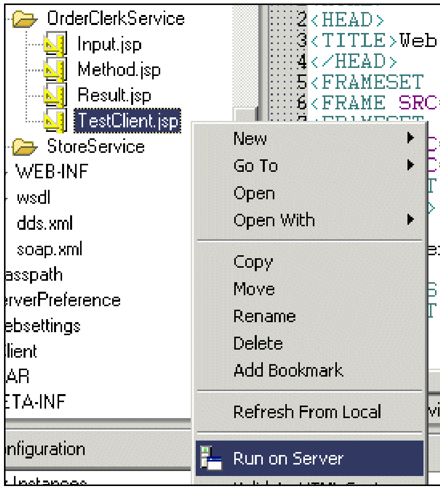

Figure 4-6 Starting WebSphere server in WSSD

4. Wait until you see the message “`Server Default Server Open for e-business`” in the Console. A browser window opens and displays the JSP shown in Figure 4-7.

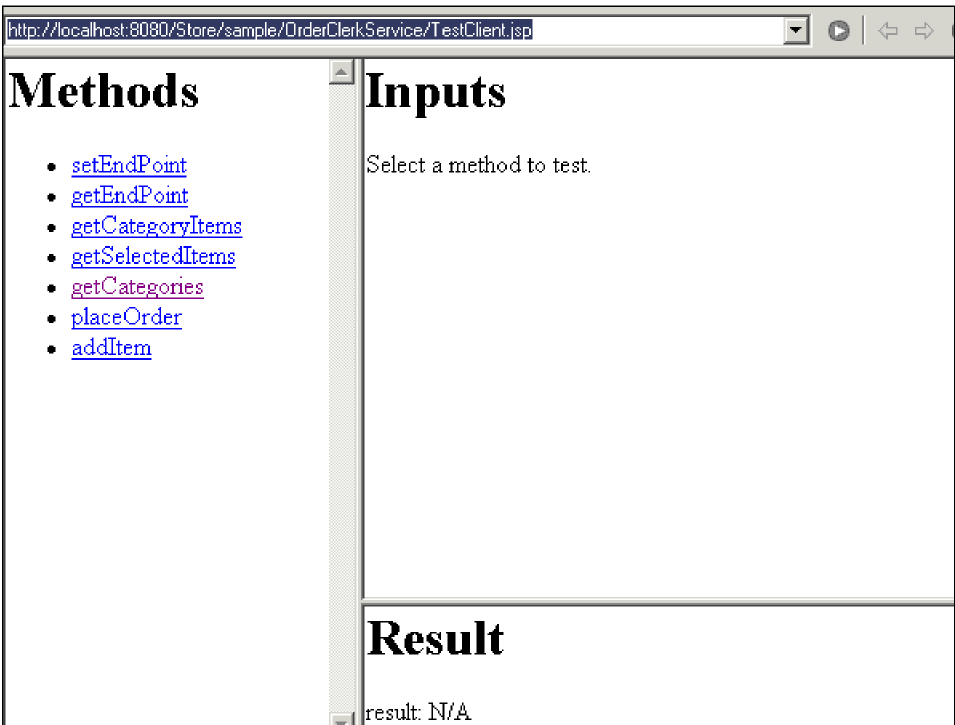

Figure 4-7 OrderClerkService test JSP

Complete the following steps to test the Order Entry Web service:

1. Make sure that the socket server is started on the iSeries (see Chapter 2, “Understanding the sample RPG application” on page 9).
2. Click the **getCategories** method.
3. Click the **Invoke** button. The structure of a response is shown in Table 4-1 on page 33.
4. You can continue testing other Web service methods. Use Table 4-1 as a reference for input and output parameters.

We can use the generated JSPs for testing a Web service and as a sample of using a Web service proxy. Open Result.jsp and review the code. At runtime, the JSP code creates an instance of OrderClerkServiceProxy object specified in the `<jsp:useBean>` tag. The proxy object is then used to make calls to Web service methods.

4.3 Implementing the Store Web service Java Bean

The Store Web service is an integrating Web service. It combines the Order Entry and the Loan Web services. The Store Web service contains business logic of an online retailer application.

4.3.1 Defining a public interface for the Store Web service Java Bean

We defined the Store Web service JavaBean public interface based on business requirements of a potential Web service client. The Store Web service JavaBean public interface consists of the following methods:

- **`init()`**: Initializes the Store service
- **`getCategories()`**: Makes a call to the `OrderClerkService.getCategories()` method
- **`getCategoryItems()`**: Makes a call to the `OrderClerkService.getCategoryItems()` method
- **`addItemsToTheShoppingCart()`**: Makes a call to the `OrderClerkService.addItem()` method
- **`getShoppingCartItems()`**: Makes a call to the `OrderClerkService.getSelectedItems()` method
- **`placeOrder()`**: Makes a call to the `OrderClerkService.placeOrder()` method
- **`login`**: Checks if a user is a registered customer
- **`createCustomer()`**: Creates a new Customer object and makes a call to `LoanOfficerService.establishLineOfCredit()`
- **`getLoans()`**: Makes a call to the `LoanOfficerService.getLoans()` method
- **`signLoan()`**: Makes a call to the `LoanOfficerService.signLoan()` method

In the `init()` method, we create three customers (see Example 4-8). Usually customer information is stored in a database, but we store it in the Java object to avoid introducing JDBC code in our sample application.

Example 4-8 Implementation of the Store Web service init() method

```
public void init(){

      // Create customer objects and store them in a map

      // Customer 1
      Customer cust1 = new Customer();
      cust1.setCustomerId("Mike");
      cust1.setFirstName("Mike");
      cust1.setLastName("Brown");
      cust1.setIncome("50000");

      customers.put(cust1.getCustomerId(), cust1);

      // Customer 2
      Customer cust2 = new Customer();
```

```
        cust2.setCustomerId("Emily");
        cust2.setFirstName("Emily");
        cust2.setLastName("Sterling");
        cust2.setIncome("80000");

        customers.put(cust2.getCustomerId(), cust2);

        // Customer 3
        Customer cust3 = new Customer();
        cust3.setCustomerId("Doug");
        cust3.setFirstName("Doug");
        cust3.setLastName("Smirnoff");
        cust3.setIncome("100000");

        customers.put(cust3.getCustomerId(), cust3);

    }
```

The implementation of most Store Web service Java Bean public methods makes a call to one of the contained Web services. The Store Web service Java Bean uses OrderClerkService and LoanService proxy classes to make calls to the contained Web services. Example 4-9 shows the implementation of the `getLoans()` method.

Example 4-9 Implementation of the getLoans() method

```
public Element getLoans(String customerId, String salary, String loanAmount){

        Element result = null;

        try{
            // loanService is a reference to LoanOfficerServiceProxy object
            result =  loanService.getLoans(customerId,salary,loanAmount);
        }catch(Exception e){
            result = handleError(e);
        }
        return result;
    }
```

4.3.2 Generating Web service deployment files for the Store Web service Java Bean

Complete the steps described in 4.2.2, “Generating Web service deployment files for wrapper Java Beans” on page 34, to generate deployment files for the Store Web service.

5

Web service client

This chapter describes the implementation of a Web service client - the Store Web Application. It explains the application architecture and shows the steps to test it.

Note: You can find the instructions for downloading and installing the sample applications in Appendix A, “Additional material” on page 61.

© Copyright IBM Corp. 2003. All rights reserved.

5.1 The Store Web application architecture

A Web service client (requestor) can be any application that can send and receive SOAP messages. This chapter describes one of the possible ways to implement a Store Web service client.

The Store Web application consists of JSPs, a servlet, and Java classes. Figure 5-1 shows the application request flow. A JSP sends a request to the Store servlet. The Store servlet contains a reference to the StoreServiceProxy class. The StoreServiceProxy class was generated by Development Studio Client when we prepared the Store Web service for deployment (see Chapter 4, “Implementing the Store Web service” on page 27). The StoreServiceProxy class makes SOAP calls to the Store Web service.

The StoreServlet uses the ParseXMLHelper class to parse XML returned by the StoreServiceProxy class. For example, the Store Web service returns a list of furniture categories as an XML document. Parsed data is saved in the StoreViewBean class. Application JSPs access the StoreViewBean to display retrieved information in the browser.

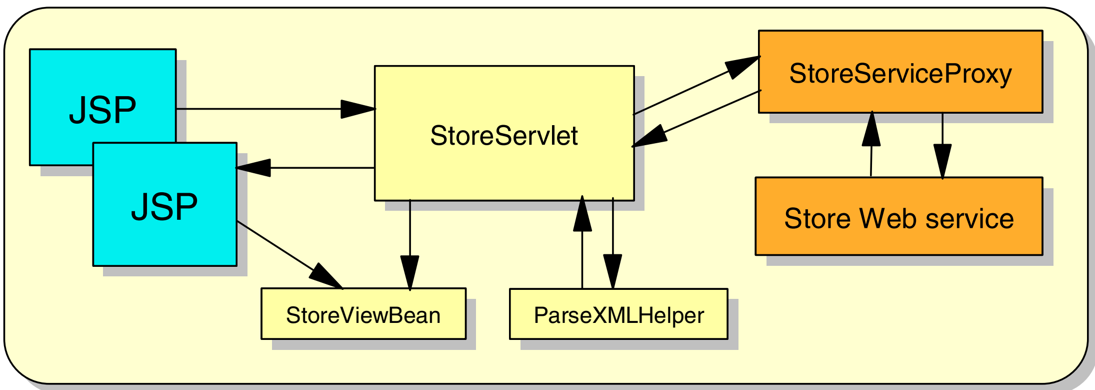

Figure 5-1 Store Web application request flow

The steps are shown in the implementation of the `StoreServlet.getCategoryItems()` method (see Example 5-1).

Example 5-1 Implementation of the StoreServlet.getCategoryItems() method

```
private void getCategoryItems(String categoryId) throws Exception{

   // Use store Web service proxy to retrieve category items
   // storeService is a reference to the StoreServiceProxy object
   Element element = storeService.getCategoryItems(categoryId);
   // Put all returned categoires in a Vector and save the vector
   // in a Java bean, the Java bean will be accessed in a JSP
   ParseXMLHelper xmlHelper = new ParseXMLHelper();
   storeViewBean.setCategoryItems(xmlHelper.getCategoryItems(element));

}
```

5.2 Testing the Store Web application in Development Studio Client

Complete the following steps to test the Store Web Application in Development Studio Client:

1. Import the **StoreWebService.ear** file into Development Studio Client and test the Store Web service (see Chapter 4, “Implementing the Store Web service” on page 27).
2. Start the socket server on the iSeries server (see Appendix A, “Additional material” on page 61).
3. Import the **StoreWebServiceClient.ear** file into Development Studio Client.
4. In Development Studio Client, switch to the Server Perspective by selecting **Perspective-> Open-> Other-> Server**.
5. Click the **Servers** tab (by default located at the bottom of the window).
6. Expand the **StoreClient\webApplication** folder as shown in Figure 5-2.

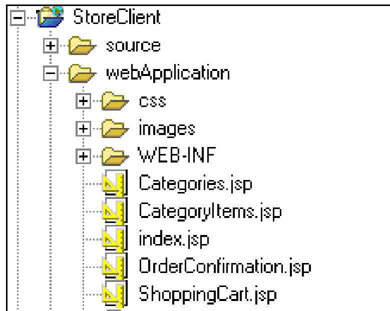

Figure 5-2 StoreClient project files

7. Select **index.jsp**. Right-click it and select **Run on Server**.
8. Your browser displays the JSP shown in Figure 5-3. Provide an existing customer ID or create a new customer ID. The existing customer IDs are Mike, Emily, and Doug.
9. Click the **Submit** button.

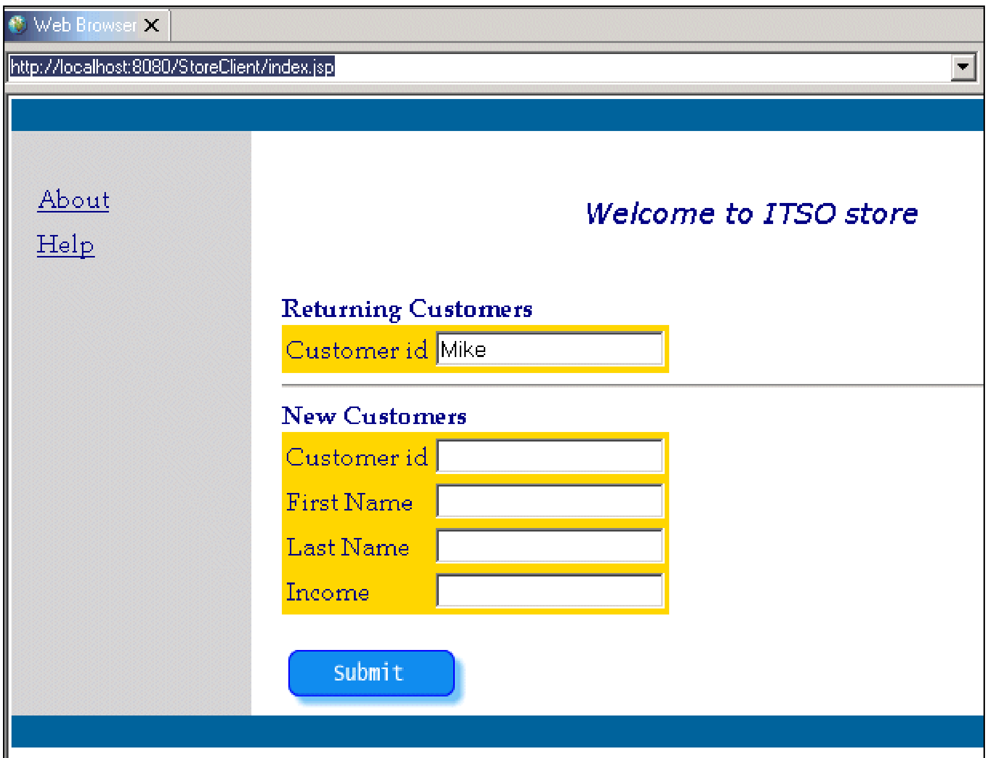

Figure 5-3 The Store Web application startup page

10. The next page displays furniture categories returned by the Store Web service as shown in Figure 5-4. Select one of the categories.

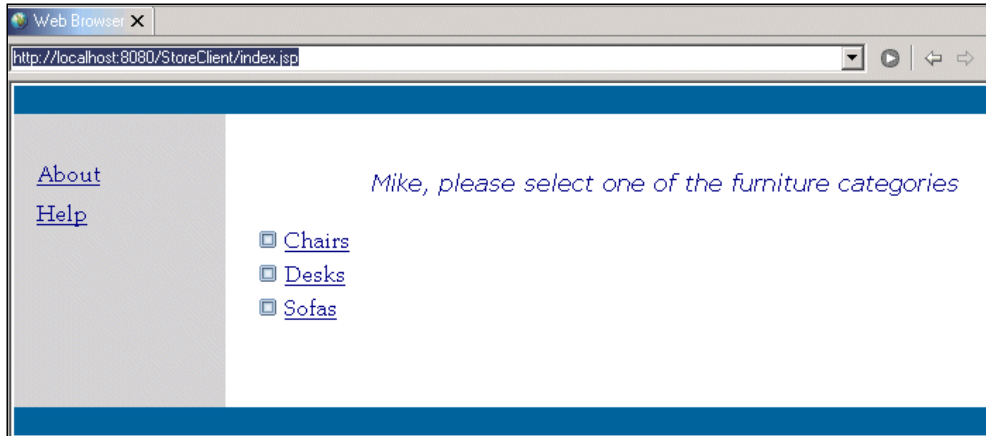

Figure 5-4 Furniture categories returned by the Store Web service

11. The next JSP (see Figure 5-5) shows the list of items in the selected category. Select several items, specify quantity for each item and click the **Add to Cart** button.

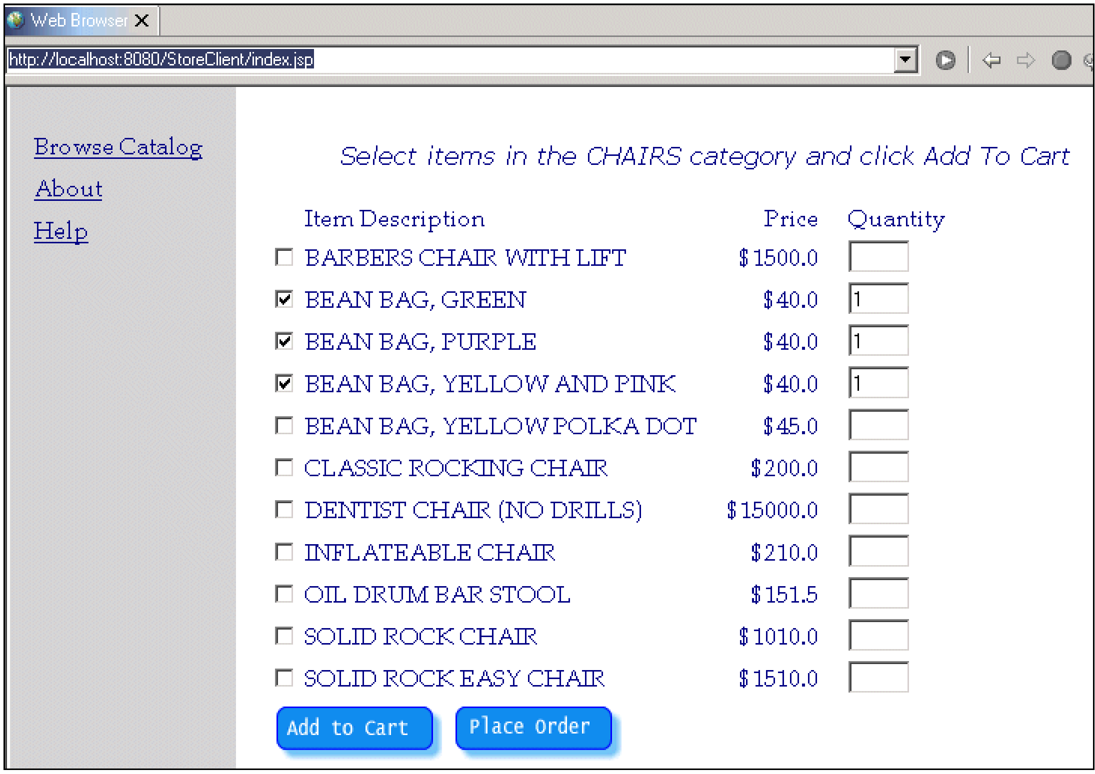

Figure 5-5   Adding items to the shopping cart

12. A confirmation message is displayed on top of the page. Click the **Place Order** button.
13. The next JSP shows a list of selected items and a list of loans available to the customer (see Figure 5-6). Select one of the loans and click the **Submit** button. Clicking the Cancel button takes you back to the Categories list.

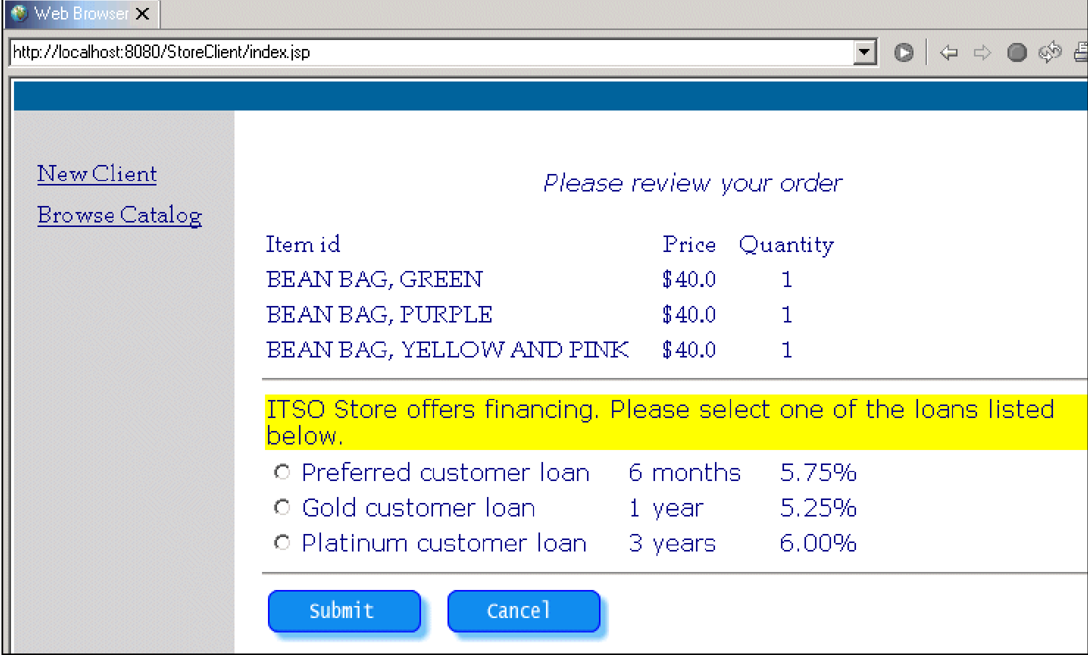

Figure 5-6   Review order and select a loan JSP

14.The next JSP displays order number and loan number as shown in Figure 5-7.

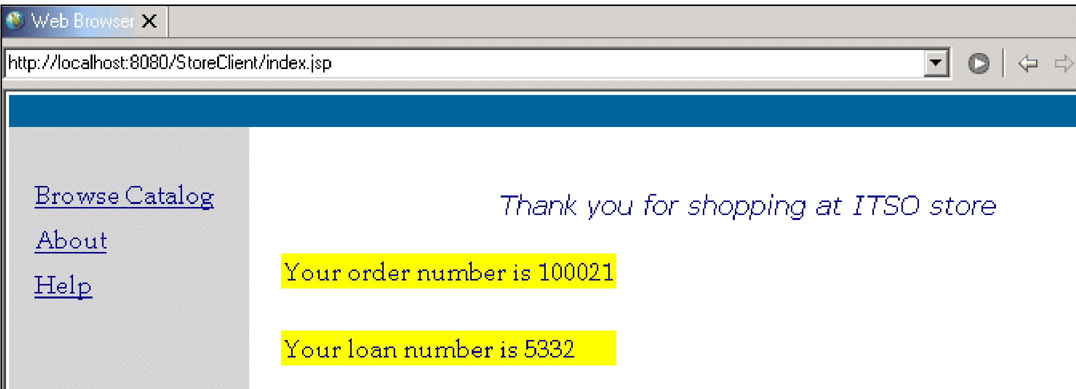

Figure 5-7 Order and loan confirmation JSP

15.Follow instructions in 6.3, "Deploying Web services in WebSphere Application Server" on page 54, to install the application in WebSphere Application Server 4.0 Advanced Edition.

6

Web services deployment

This chapter explains how to deploy Web services in WebSphere Application Server 4.0 Advanced Edition.

© Copyright IBM Corp. 2003. All rights reserved.

6.1 Exporting an EAR file from Development Studio Client

We deploy all three Web services (the Store Web service, the Order Entry Web service, and the Loan Web service) on one WebSphere application server. However, in a production environment, these Web services may be deployed on different servers.

Web services wizard uses the URL of the WebSphere application server provided in Development Studio Client to generate WSDL and proxy files. We need to replace all references to the test server with the location of the server where the Web services will be deployed. Complete these steps to replace server references:

1. In Development Studio Client, select **Edit-> Search**.
2. In the Search Expression field, enter `localhost:8080/Store/`.
3. Click the **Browse** button next to the Extensions field.
4. Click the **Select All** button.
5. Click the **Search** button. The search returns nine files (three files for each Web service).
6. Open each file and replace `localhost:8080` with the name and port number of the WebSphere Application server to be used for Web services deployment. Save the changes.

We are now ready to export the EAR file:

1. In Development Studio Client, select **File-> Export**.
2. Select **EAR file**.
3. Click **Next**.
4. In the What resources do you want to export? field, select **StoreEAR**, and specify the export location and EAR file name (see Figure 6-1).
5. Click the **Finish** button.

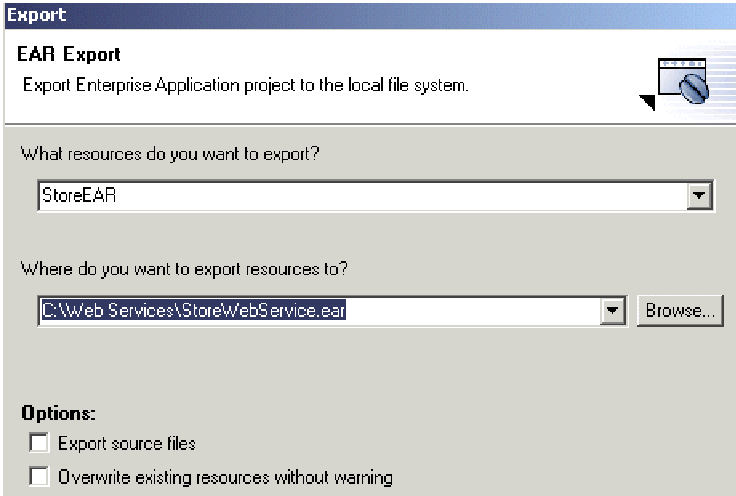

Figure 6-1 Exporting the EAR file

6.2 Deploying Web services in the SOAP server

The Web services wizard automatically includes the SOAP server (rpcrouter servlet) in the Web service application and deploys Web services in the SOAP server.

To deploy Web services in the SOAP server, follow these steps:

1. In Development Studio Client, open the Server Perspective by selecting **Perspective-> Open-> Other-> Server**.
2. Expand the **Store\webApplication\WEB-INF** folder.
3. Open the **web.xml** file and select the **Servlets** tab. Notice that rpcrouter is one of the defined servlets.
4. Expand the **Store\webApplication\Admin** folder. This folder contains JSPs used to access the SOAP server.
5. Select **index.html**. Right-click it and select **Run on Server**.
6. Click the **List all services** link. All three Web services defined in our project have been registered with the SOAP server by the Web services wizard (see Figure 6-2).

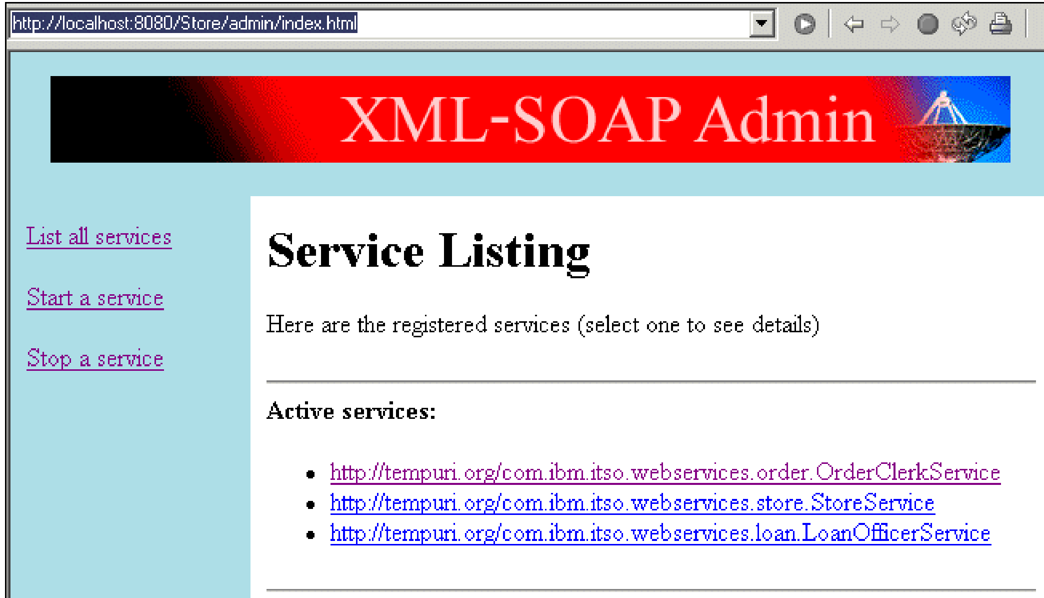

Figure 6-2 SOAP server configuration

7. Click each service and examine its properties. Information displayed in the Deployed Service Information page (Figure 6-3) is stored in the Store/webApplication/dds.xml file (also known as *SOAP deployment descriptor*) generated by the Web services wizard.

XML-SOAP Admin

Deployed Service Information

'http://tempuri.org/com.ibm.itso.webservices.loan.LoanOfficerService' Service Deployment Descriptor

Property	Details
ID	http://tempuri.org/com.ibm.itso.webservices.loan.LoanOfficerService
Scope	Application
Provider Type	java
Provider Class	com.ibm.itso.webservices.loan.LoanOfficerService
Use Static Class	false
Methods	establishLineOfCredit, getLoans, signLoan, disconnect, init
Type Mappings	
Default Mapping Registry Class	

Figure 6-3 Loan service deployment descriptor

6.3 Deploying Web services in WebSphere Application Server

Complete the following steps to install the Store Web service in WebSphere Application Server 4.0 Advanced Edition:

1. Start the WebSphere Administrative Console.
2. Select the **Install Enterprise Application** wizard as shown in Figure 6-4.

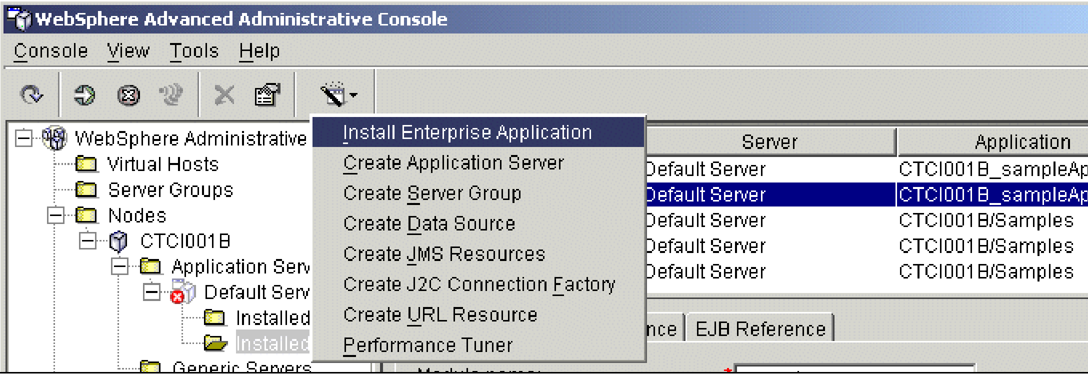

Figure 6-4 Starting the Install Enterprise Application wizard

3. Browse to the Store EAR file you exported from Development Studio Client and enter application name as shown in Figure 6-5.

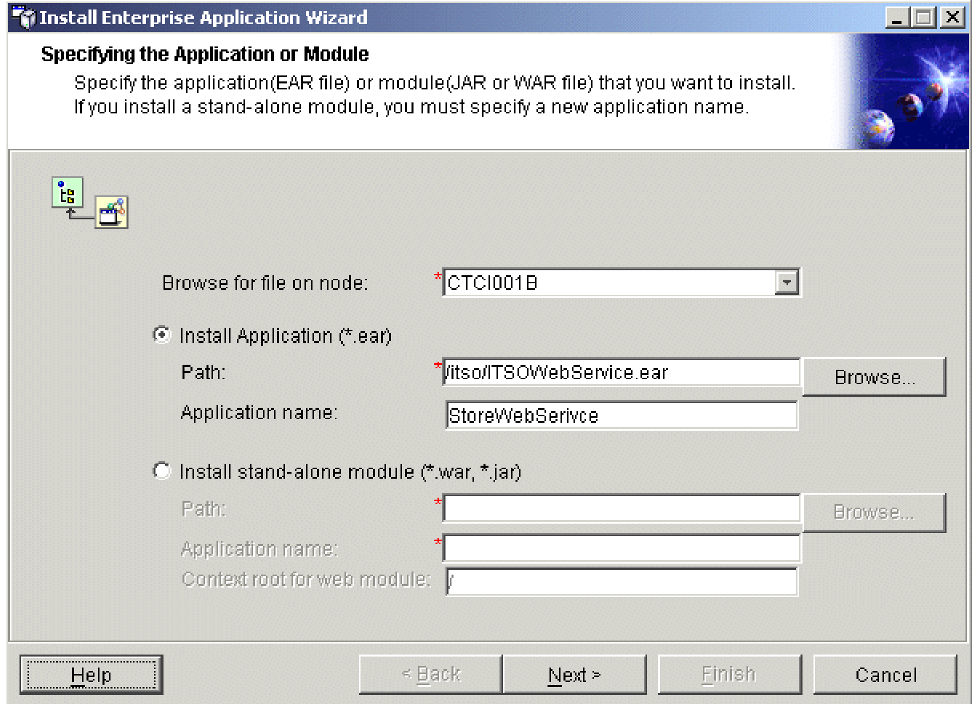

Figure 6-5 Selecting the Store Web service EAR file

4. Click the **Next** button.
5. Accept the default values in the rest of the wizard windows.
6. Click the **Finish** button.
7. Test the installation:
   a. Start the Default server.
   b. Open a browser and bring up one of the Test JSPs, for example http://<<servername:port>>/Store/sample/OrderClerkService/TestClient.jsp. Complete the steps described in 5.2, “Testing the Store Web application in Development Studio Client” on page 47.

If you want to run the Store Web Application with the Store Web service deployed in the WebSphere Application Server, you need to modify the StoreServiceProxy class file in the Store Web application:

1. In Development Studio Client, switch to the Web perspective.
2. Expand the **StoreClient\source\proxy\soap\com\ibm\itso\webservices\store** folder.
3. Double-click **StoreServiceProxy.java**
4. Replace `localhost:8080` in the value of the string URL variable with the hostname and port name of the WebSphere server.

7

Other topics

This Redpaper focuses on the basic requirements needed to build a Web service. However, there are several other topics that you must consider when designing a production-level Web service. We do not have space to address them in the scope of this Redpaper, but we felt it was important to touch on them briefly. These topics include:

- Security
- Transactions
- Trust

Furthermore, there is a flurry of activity in the industry related to Web services in general. You should check the current status of these topics before you write your own Web services.

© Copyright IBM Corp. 2003. All rights reserved.

7.1 UDDI

Universal Discovery Description and Integration (UDDI) is a registry of Web services. UDDI performs a similar function for Web services as Google and Yahoo do for Web pages or as the phone book does for telephone numbers.

UDDI is usually described as one of the "big three" technologies needed for Web services. We did not use it for our example because it was not necessary and would have complicated the discussion. But it certainly can be included. UDDI is used to support the following operations:

- **Publish**: The provider publishes the Web service to a UDDI registry.
- **Find**: A requestor searches the public registry to find an appropriate Web service.
- **Bind**: The requestor uses the information from the registry to call, or use, the Web service.

There are many scenarios for using UDDI with the Web services you write. The "classical" scenario, although it is not often used today, is to use a public UDDI registry. The industry group UDDI.org has set up such a public UDDI registry. It is intended to be like Google or Yahoo for Web services. However, publishing Web services to a public UDDI registry raises some important security and business model questions.

Many experts expect Web services to be deployed for use within an organization. A private UDDI registry delivers the same benefits without raising security or business model questions. Of course, individual departments can set up a department level UDDI registry. It may simplify things initially, but it does not encourage inter-department sharing of Web services.

A final approach, which is where most developers start, is to ignore UDDI altogether. Requestors are told exactly where to find their providers without using any UDDI search mechanism. This certainly works for a while, but the provider's details are buried in the requestor's code, which makes it difficult to change.

7.2 Security

One approach to security is to use existing protocols, such as SSL or Transport Level Security (TLS), to encrypt communication between requestor and provider. This satisfies a requirement to prevent a third-party from intercepting and eavesdropping on the conversation.

W3C is defining a standard way to encrypt XML called *XML Encryption*. It allows encrypting of the contents of a SOAP message and therefore protects the message from being read regardless where it may be intercepted in the entire business process.

W3C is also defining a standard way to digitally sign XML called *XML Signature*. One party can digitally sign a document in a way that no one else can duplicate. The signatures can be used to "prove" that the party did, in fact, sign the document. Digital signatures are used to ensure the signer can not back out of an agreement.

Digital signatures lead to a question of whether the legal court system will uphold agreements made by electronic means. That is, are digital signatures legally enforceable? The new Electronic Signatures in Global and National Commerce Act (E-SIGN) seems to support such enforcement, Be sure to consult legal council before you enter into any agreements of such importance.

XML Encryption and Signature do not contain enough information to completely specify how they are to be used for SOAP messages. IBM, Microsoft, and others have recently proposed

WS-Security, which closes most of the gap. WS-Security defines the basic structure for an element of the header portion of the SOAP envelop. This section is used to hold various security-related information in a standard format.

Security Assertion Markup Language (SAML) is another promising piece of the Web service security puzzle. It is an XML vocabulary in which multiple “authorities” can make assertions about a “subject” (either a person or a computer). An authority can assert that:

- It has authenticated the subject.
- The subject has a certain set of attributes.
- It has determined that the subject is authorized to access various resources.

As you can see, there is a tremendous amount of energy being expended on working out various security issues related to Web services.

Think through the security requirements of your Web service. Our sample application does not adequately address security needs. For example, SSL should be used, at least, to ensure the client’s loan information is not intercepted while it is being transmitted between the client and the Store, or while it is being transmitted between the Store and the Loan. An authentication mechanism (for example, userid and password) is needed to ensure the client is who they claim to be.

7.3 Transactions

SOAP is a mechanism for one requestor to call one provider. But many Web services need to make multiple calls to multiple providers. And all these calls may need to be coordinated in some way. Recently IBM, Microsoft, and BEA proposed WS-Coordination, WS-Transaction, and Business Process Execution Language (BPEL) to address such coordination.

If all the calls occur in a short duration of time, then a two-phase commit protocol can be used so that the calls succeed or fail as a single unit. However, many business processes require more time to complete and locking resources for this long period of time is not acceptable. In this case, a different coordination protocol can be used in which the original operation completes right away. If a failure occurs later, “compensating transactions” are applied to back out the original operation.

BPEL is a proposed language and execution engine that allows developers to write a Web service that integrates multiple other Web service providers. The calls to the multiple providers are “scripted” together. Failure in the script causes a roll back or compensating transactions to be applied.

7.4 Collecting revenue

Methods and business models to collect revenue from Web services have not yet been tested and shown to be successful. This may be due, in part, to expectations that everything on the Web should be free, from information to MP3 files. But there are a couple of obvious models that you may consider.

A number of businesses tried various forms of “pay per use” or “micro-payment”. That is, every time you request a stock quote from a Stock Quote Web service, you are charged some tiny amount, say 1/10 of a cent. Unfortunately, none of these business models seemed to work.

Many Internet Service Providers (ISPs) have used a subscription model where users pay a fixed fee each month and are allowed so many hours of use. Applying this model to Web services should not pose any problems.

Another way to generate revenue from Web services is to offer it free to “preferred” or “Gold” customers. It doesn’t generate revenue directly, but it may be an incentive for customers to upgrade. Similar to this approach is for business partners to allow each other free access to each others Web services. This is essentially bartering Web services.

7.5 Trust

As a requestor, assume you have determined which interface you need (WSDL), found which providers support that interface (perhaps through UDDI), have chosen one, and have ensured that they take appropriate security precautions. But how do you know that you can *trust* them?

For example, suppose you want to order 10,000 widgets for delivery at the end of the month. You make substantial investments and commitments of your own based, in part, on the provider’s statement that the goods will be delivered on time. But how do you know you can trust them to keep that commitment? There is no easy technical solution to this problem?

In the physical world, one approach is to start out with smaller, “trail” orders and build up trust. This approach can also be used in a Web services world.

Another, important approach in the physical world is for the requestor to meet the provider’s people face-to-face, “size them up”, and get a feel for their capabilities and his commitment. Face-to-face meetings do not translate to the Web services world.

Additional material

This Redpaper refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this Redpaper is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/REDP0192`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the Redpaper form number REDP0192.

Using the Web material

The additional Web material that accompanies this Redpaper includes the following file:

File name	*Description*
ITSO_WS.zip	Sample applications

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	5 MB
Operating System:	Windows NT/2000
Processor:	Pentium III or higher
Memory:	512 MB minimum

© Copyright IBM Corp. 2003. All rights reserved.

How to use the Web material

Create the `ITSO` subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Installing the RPG applications

You need to install the RPG applications and the socket server to run the Web services sample applications. Follow these steps:

1. Create the following user profile on iSeries:
   - ID: `ITSO`
   - password: `JAVA1A`
2. Create the library TEST.
3. Create a *SAVF file in the TEST library with the name `ITSOWS`.
4. Send the ITSOWS file by FTP from your PC (in the directory to where you have extracted ZIP file) to the ITSOWS save file on the iSeries server.
5. Restore the ITSOWS library with the following command:

   ```
   RSTLIB SAVLIB(ITSOWS) DEV(*SAVF) SAVF(TEST/ITSOWS)
   ```
6. Start the socket server with the following command:

   ```
   SBMJOB CMD(CALL PGM(ITSOWS/MAIN1))
   ```
7. Your socket server is ready to accept requests on the port 27850.

Importing the EAR files

If you follow the instructions explained in this Redpaper, you need to import two .ear files into Development Studio Client. Unfortunately, the classpath information is not included in the .ear files.

To simplify the instructions, we included the .classpath files for both .ear files. After you import an .ear file, you can copy the content of a text file, included in the ZIP file, and paste it to the corresponding .classpath file.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this Redpaper.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 63.

- *Web Services Wizardry with WebSphere Studio Application Developer,* SG24-6292
- *Self-Study Guide: WebSphere Studio Application Developer and Web Services,* SG24-6407

Referenced Web sites

These Web sites are also relevant as further information sources:

- Simple Object Access Protocol (SOAP) 1.1 specification

  `http://www.w3.org/TR/SOAP`
- IBM Web Services Toolkit

  `http://www.alphaworks.ibm.com/tech/webservicestoolkit`
- UDDI home page

  `http://www.uddi.org`
- Apache SOAP Web page

  `http://xml.apache.org/soap`
- ebXML Web page by OASIS

  `http://ebxml.org`
- OASIS home page

  `http://www.oasis.com`
- Netshare400 portal dedicated to AS/400 and iSeries

  `http://www.netshare400.com`

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.

© Copyright IBM Corp. 2003. All rights reserved.

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.

Enabling Web Services for the IBM eserver iSeries Server

Redpaper

Build Web services on iSeries by using this how-to guide

Gain an in-depth understanding of all aspects of Web services

Learn about the new tools from IBM to develop Web services

Enterprises are interested in integrating multiple applications within or across companies. Integrating brings several major benefits:

- The ability to use existing host applications without modifications
- Extended business value for the customers
- Streamlined introduction of new services to the customers
- Reduction in I/T expenses on application development and maintenance
- Simplification of a user interface
- A single point of access for all customer applications

The newest integration technology of Web services provides these great promises. Web services enable the dynamic, flexible, and potentially real-time selection of any kind of business service across the Internet. Business processes running in different enterprises can bind together and cooperate as if they belong to a seamlessly designed system. However, they may be implemented by different software vendors or using different software technologies.

This IBM Redpaper explains how to develop Web services for an IBM eserver iSeries environment. It introduces you to Web services and Web services technologies. It shows how to develop a Web services sample application with WebSphere Development Studio Client for iSeries. It also shows how to enable iSeries Web services with WebSphere Application Server.

The material assumes that you have extensive programming experience in Java. As such, this Redpaper targets Java programmers who want to start developing Web services.

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