

NetBeans IDE Field Guide

Copyright © 2005 Sun Microsystems, Inc. All rights reserved.

Table of Contents

Extending J2EE Applications with Web Services	1
Consuming Existing Web Services	2
Implementing a Web Service in a Web Application	7
Creating a Web Service.....	7
Adding an Operation.....	10
Compiling the Web Service.....	11
Creating Web Services from WSDL.....	13
Implementing Web Services within an EJB Module.....	14
Testing Web Services	15
Adding Message Handlers to a Web Service.....	19

Extending J2EE Applications with Web Services

In the following sections, you will learn how easy it is to create and add Web services to J2EE Applications (both Web applications and EJB Modules) and how to publish them so that they can be used by other applications and tested from within the IDE.

What is a Web service

The W3C organization defines a Web services, as follows: "A Web service is a software system identified by a URI whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols." The implementation for a Web service can be done in any language, and a Web service can be accessed by many different platforms, since the messages are XML-based. The J2EE 1.4 Platform has specified Web services creation for Web applications (J2EE Web Tier-based) and EJB Modules. NetBeans IDE 4.1 supports the creation of JSR 109 Web services in J2EE Applications, as well as the consumption of published Web services, within J2EE Applications.

Web services allow applications to expose business operations to other applications, regardless of their implementation. This is possible via the usage of the following standards:

- **XML, the Common markup language for communication**--Service providers, who make services available, and service requestors, who use services, communicate via XML Messages.
- **SOAP, the Common message format for exchanging information** —These XML messages follow a well defined format. Simple Object Access Protocol (SOAP) provides a common message format for Web services.
- **WSDL, the Common service specification formats**--In addition to common message format and markup language, there must be a common format that all service providers can use to specify service details, such as the service type, the service parameters, how to access the service, etc. For example, Web services Description Language (WSDL) provides Web services with common specification formats.

Consuming Existing Web Services

In order to be usable by other applications, a Web service must publish a WSDL file. (This can be done via a UDDI registry. The IDE does not let you publish web services to a UDDI registry, although the WSDL files that you use can come from a variety of sources, including a UDDI registry.) It is this WSDL file which is used by the application to construct the necessary artifacts. The NetBeans Web Service Client wizard automatizes the creation process and updates the deployment descriptor files with the appropriate `<service-ref>` elements.

To create a WSDL file:

1. Open the the Web Service Client wizard by right-clicking the node of a Web application project and choose New | Web Service Client.



Figure 9-1

New File wizard: Create Web Service Client

2. In the wizard, pick a WSDL file either from the URL of the running service or from a local directory on your system.

Make sure you pick a package name for the generated interfaces that will be used by your user code to access and interact with this Web service. Two types of Web service clients can be generated:

- **JSR-109 (Enterprise Web services).** Enhances JSR-101 by defining the packaging of Web services into standard J2EE modules, including a new deployment descriptor, and defining Web services that are implemented as session beans or servlets. This is the recommended and portable (via the J2EE 1.4 specification) way.
 - **JSR-101 (JAX-RPC).** Defines the mapping of WSDL to Java and vice versa. It also defines a client API to invoke a remote Web service and a runtime environment on the server to host a Web service.
3. Click the Finish button.

In the Project window, you should now see a new logical node under the Web Services References node (as shown in Figure 9-2). Explore the children of this node: for each Web service operation defined in the WSDL file, the IDE shows a node that has this operation name and a popup menu that allows you to test this operation directly within the IDE without writing a single line of code.

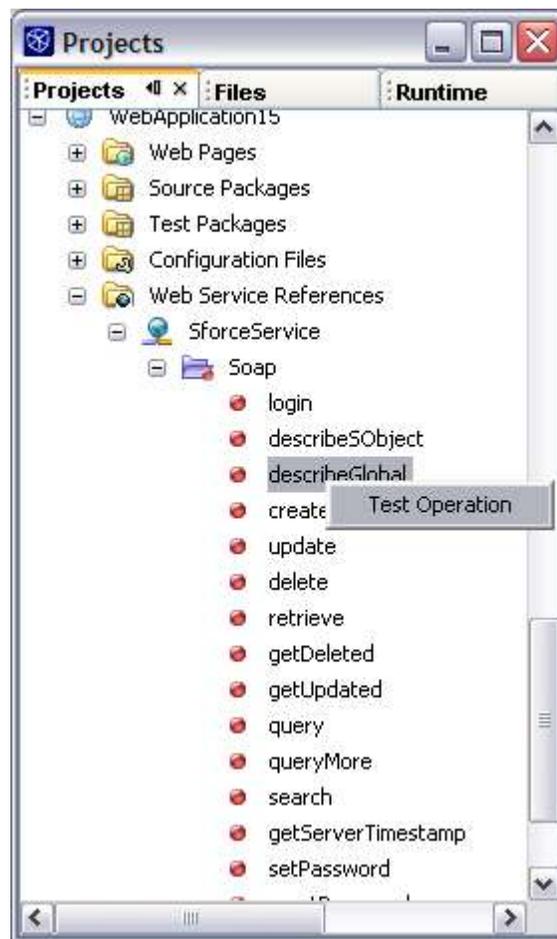


Figure 9-2
Projects window with populated Web Service Reference node

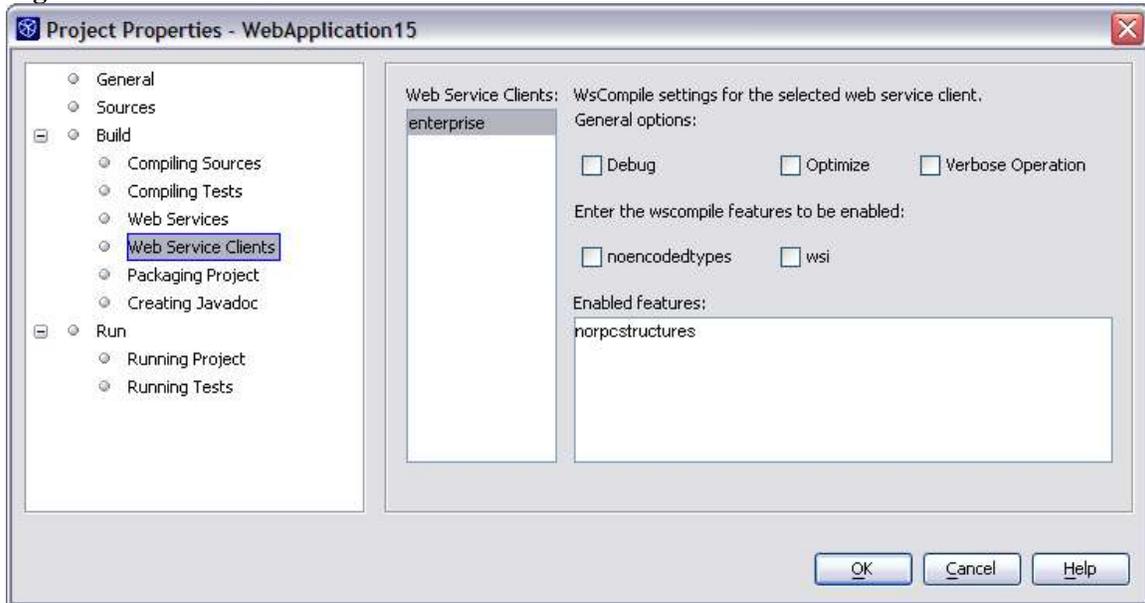
The necessary Web service Reference entry in the `web.xml` for this Web application has been correctly updated, so that you can use this Web service from either a servlet or a utility Java class that is within your Web application.

```
<service-ref>
  <service-ref-name>service/SforceService</service-ref-name>
  <service-interface>com.acme.SforceService</service-interface>
  <wsdl-file>WEB-INF/wsdl/enterprise.wsdl</wsdl-file>
  <jaxrpc-mapping-file>WEB-INF/wsdl/enterprise-mapping.xml</jaxrpc-
mapping-file>
  <port-component-ref>
    <service-endpoint-interface>com.acme.Soap</service-endpoint-
interface>
  </port-component-ref>
</service-ref>
```

While, most of the time, the IDE will select the correct `wscmpile` tool options, you can further control the `wscmpile` tool used for the client code generation from the WSDL file via the Project Properties dialog box. Right-click the project's main node and choose Properties to open the dialog. In the dialog, select the Web Services | Web Services Clients node (as shown in

Figure 9-3). For example, you can use checkboxes to set or disable the debug, optimize, and verbose flags and enter which wscompile options to use.

Figure 9-3



Project Properties dialog box with Web Service Clients pane displayed

Now, you want to call an operation for this Web service within your Java code. Right-click the location in your Java source code where you want to insert some code and choose Web Service Client Resources | Call Web Service Operation as shown in Figure 9-4. Then select the operation you want to invoke.

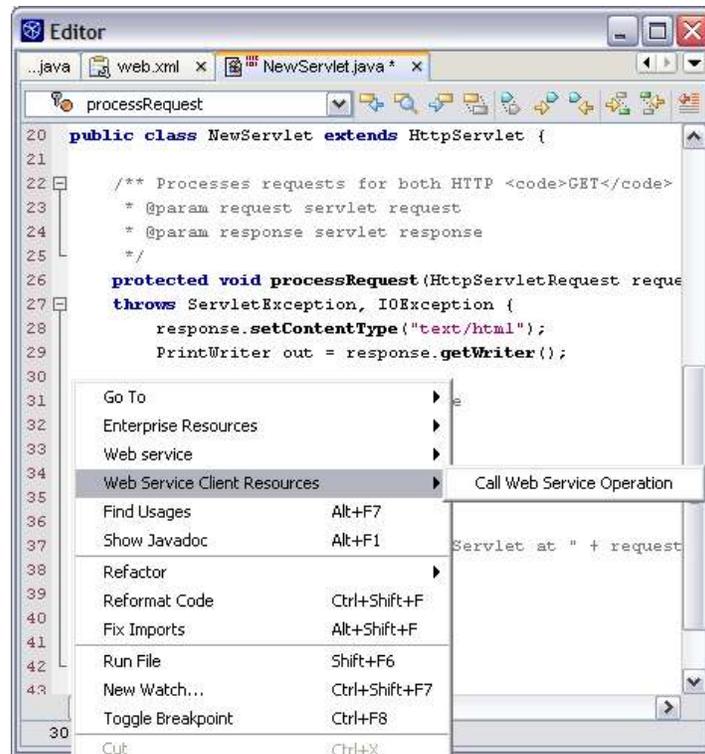


Figure 9-4
Generating code for calling a Web service operation in the Source Editor

You can then see new code that the IDE has added within your Java file.
 You should see the following try/catch block:

```

try {
    getNameFinderWebServiceRPCPort().nameFinderSearch(
        //TODO enter operation arguments */);
} catch(java.rmi.RemoteException ex) {
    // TODO handle remote exception
}

```

And you should also see some private methods that do the Web service reference lookup from the initial context, and the RCP Port accessor for the selected operation as shown below;

```

private unknown.NameFinderWebService getNameFinderWebService() {
    unknown.NameFinderWebService nameFinderWebService = null;
    try {
        javax.naming.InitialContext ic =
            new javax.naming.InitialContext();
        nameFinderWebService = (unknown.NameFinderWebService)
            ic.lookup("java:comp/env/service/NameFinderWebService");
    } catch(javax.naming.NamingException ex) {
        // TODO handle JNDI naming exception
    }
    return nameFinderWebService;
}

```

```
private unknown.NameFinderWebServiceRPC
getNameFinderWebServiceRPCPort() {
    unknown.NameFinderWebServiceRPC nameFinderWebServiceRPCPort =
null;
    try {
        nameFinderWebServiceRPCPort =
            getNameFinderWebService().
getNameFinderWebServiceRPCPort();
    } catch (javax.xml.rpc.ServiceException ex) {
        // TODO handle service exception
    }
    return nameFinderWebServiceRPCPort;
}
```

NetBeans IDE Tip

Notice the // TODO statements in the added code. NetBeans J2EE wizards always add some TODO statements so that you can quickly find in the Java source code the areas you need, such as a business logic that you need to complete. You can view all outstanding TODO statements in a single list by choosing Window | To Do.

Implementing a Web Service in a Web Application

In the IDE, you can create a Web service by implementing an existing WSDL file, exposing existing code, or creating one from scratch. The IDE generates deployment information in the deployment descriptors, the necessary Java code that describes this Web service (the default implementation, the Service End Point Interface also called “SEI”, and a compilation target in the Ant build script.)

A simple way to create a Web service is to start from scratch.

Creating a Web Service

To create a new Web service:

1. On a Web application's project node, right-click and choose the New | Web Service.
2. In the New Web Service wizard (as shown in Figure 9-5), specify a name and a Java package, select the From Scratch radio button, and click Finish.

It is strongly recommended that you do *not* use the “default package” (which would occur if you left the Package field blank).

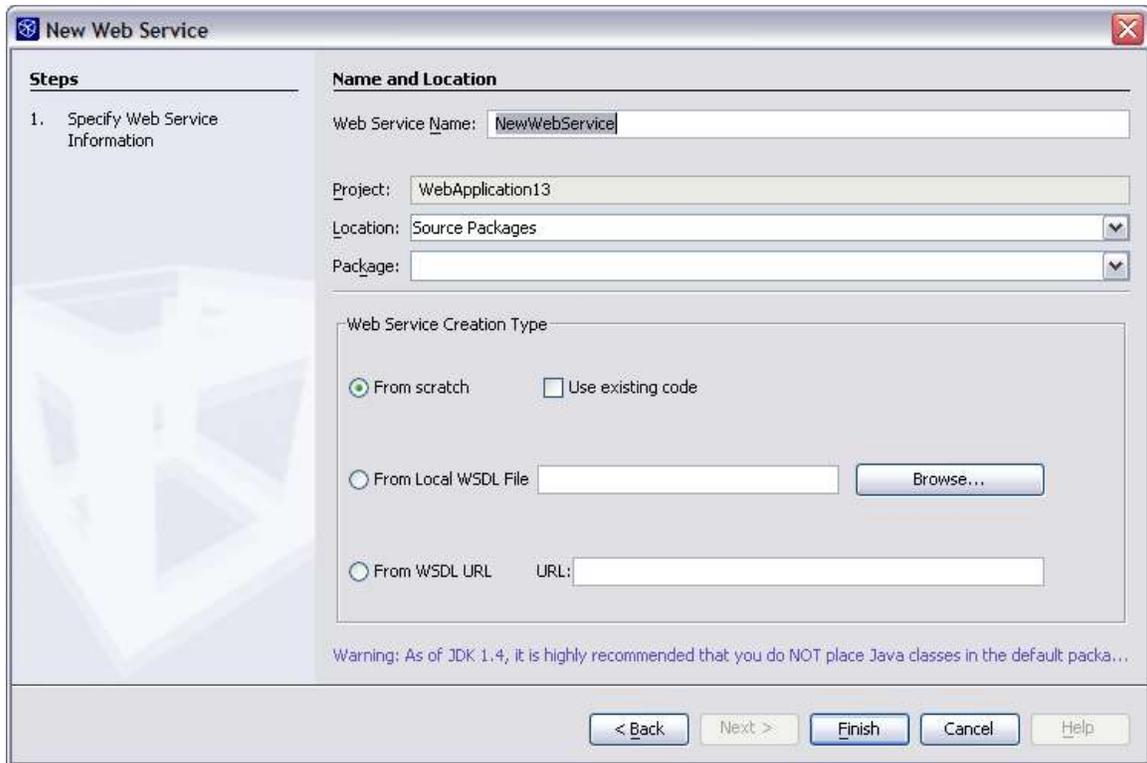


Figure 9-5
New Web Service wizard

A new Web service with no operations is added to your project. You can see a logical node representing this new component in the Web Services node in the Projects window. A new servlet entry is added to the `web.xml` file and a `webservice-description` entry is added in the `webservices.xml` description file (a sibling of the `web.xml` file). Most of the time, you don't need to worry at all about these entries as they are automatically updated by the IDE when necessary.

As a developer, the Projects window (as opposed to the Files window) will be the one you will use the most. It synthesizes the implementation details of a Web service (i.e. its implementation class and its SEI (Service Endpoint Interface)) and allows you to:

- Explore the existing operations.
- Add new operations for the Web service.
- Register the Web service to the runtime registry that will allow you to test it from within the IDE.
- Configure any Web service message handler classes that might be associated with it.

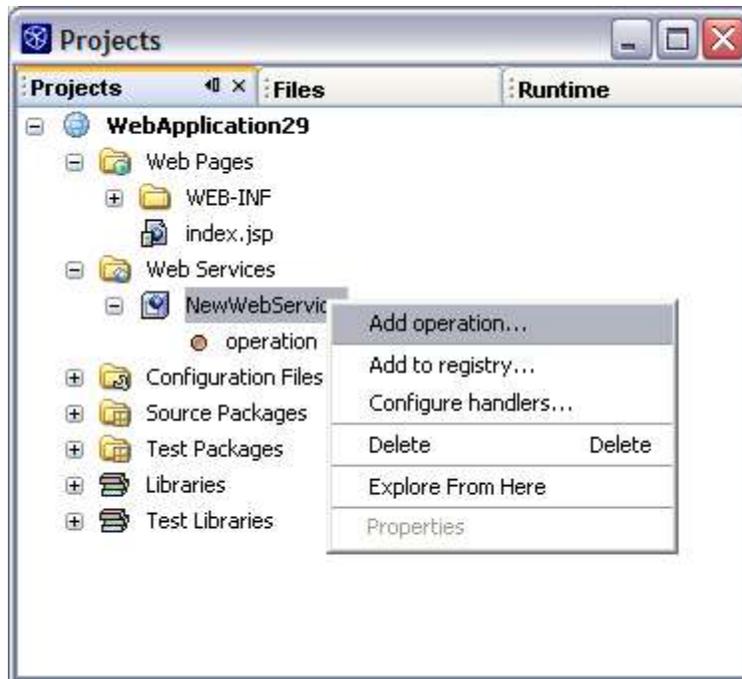
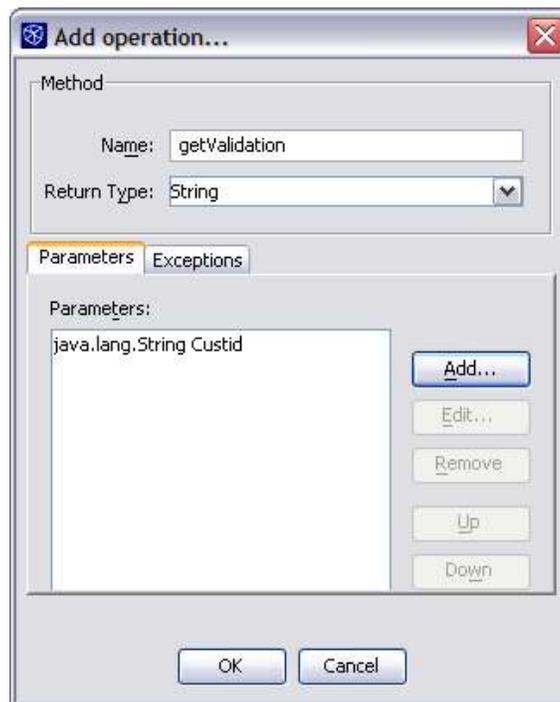


Figure 9-6
Adding an operation to a Web service

Figure 9-7

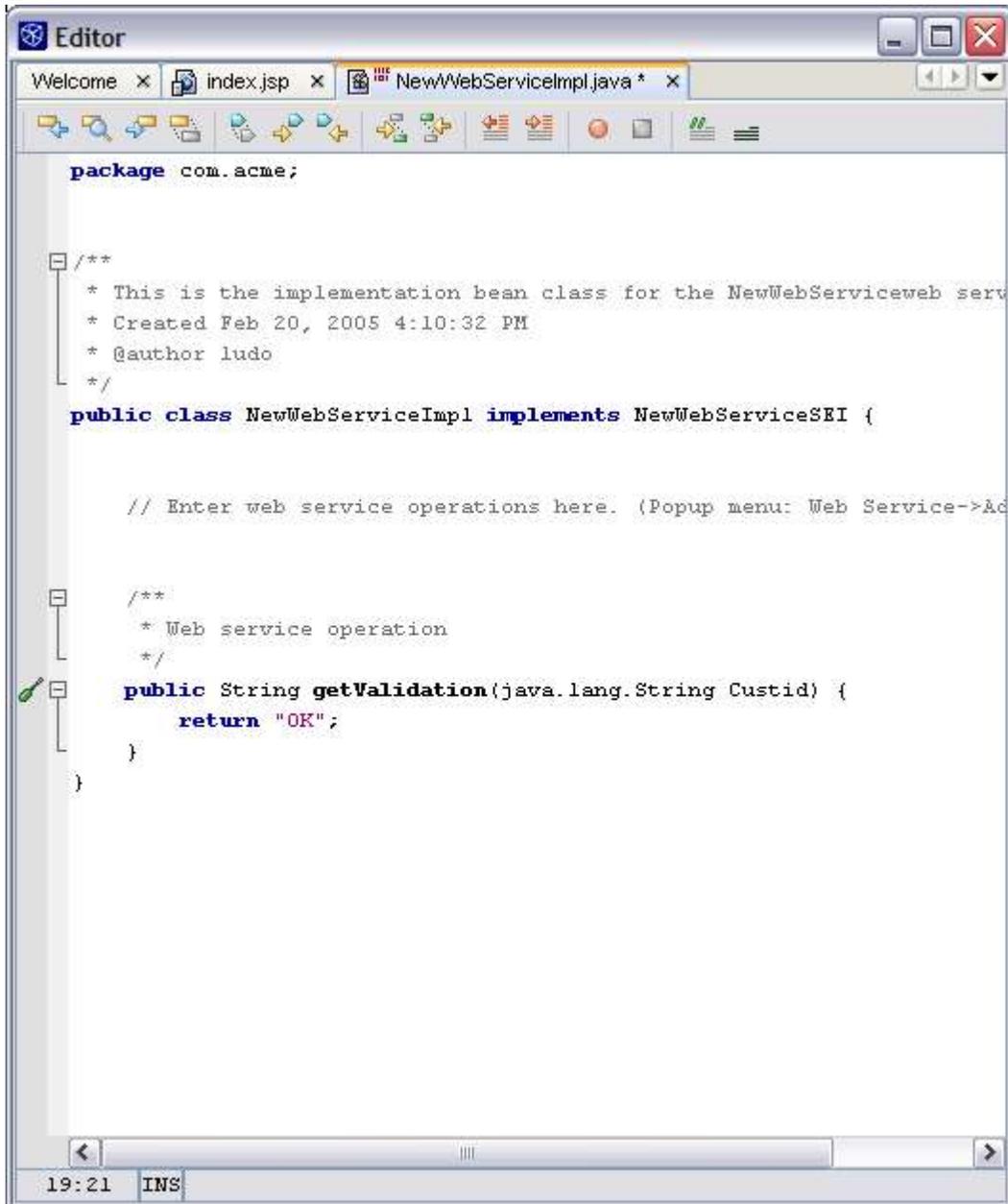


Add Operation dialog box

Adding an Operation

Choose the Add Operation (either by right-clicking the Web service's node in the Projects window, as shown in Figure 9-6, or by right-clicking within the Web service in the Source Editor). In the Add Operations dialog box (as shown in Figure 9-7), you can configure the list of parameters for this operation as well as the return type and any exceptions.

Figure 9-8



Adding an operation to a Web service

The operation is added to the Java class implementing the Web service (as shown in Figure 9-8). The IDE has automatically synchronized the Service End Point interface, so that you only need to concentrate on developing the operation method body.

Compiling the Web Service

Now that the Web service has been added to your J2EE Web application project, you just need to call the Build Project or Deploy Project commands (available by right-clicking the project's node) to trigger an Ant build process that will invoke the wscompile tool with the correct parameters. The Web application will be packaged as a WAR file and deployed to the target server for this project. Below is an example of what the Ant output might look like.

```
init:
deps-module-jar:
deps-ear-jar:
deps-jar:
library-inclusion-in-archive:
library-inclusion-in-manifest:
Copying 2 files to C:\Documents and
Settings\ludo\WebApplication29\build\web
wscompile-init:
NewWebService_wscompile:
command line: wscompile C:\j2sdk1.4.2_06\jre\bin\java.exe -classpath
"C:\j2sdk1.4.2_06\lib\tools.jar;C:\Sun\AppServer81ur1\lib\j2ee.jar;C:\S
un\AppServer81ur1\lib\saaj-api.jar;C:\Sun\AppServer81ur1\lib\saaj-
impl.jar;C:\Sun\AppServer81ur1\lib\jaxrpc-
api.jar;C:\Sun\AppServer81ur1\lib\jaxrpc-impl.jar;C:\Documents and
Settings\ludo\WebApplication29\build\web\WEB-INF\classes"
com.sun.xml.rpc.tools.wscompile.Main -d "C:\Documents and
Settings\ludo\WebApplication29\build\generated\wssrc" -features:
documentliteral -gen:server -keep -mapping "C:\Documents and
Settings\ludo\WebApplication29\build\web\WEB-INF\wsdl\NewWebService-
mapping.xml" -nd "C:\Documents and
Settings\ludo\WebApplication29\build\web\WEB-INF\wsdl" -verbose
-Xprintstacktrace "C:\Documents and
Settings\ludo\WebApplication29\src\java\com\acme\NewWebService-
config.xml"
[creating model: NewWebService]
[creating service: NewWebService]
[creating port: com.acme.NewWebServiceSEI]
[creating operation: getValidation]
[CustomClassGenerator: generating JavaClass for: getValidation]
[CustomClassGenerator: generating JavaClass for: getValidationResponse]
[LiteralObjectSerializerGenerator: writing serializer/deserializer
for: getValidation]
[LiteralObjectSerializerGenerator: writing serializer/deserializer
for: getValidationResponse]
[SerializerRegistryGenerator: creating serializer registry:
com.acme.NewWebService_SerializerRegistry]
compile:
compile-jsps:
Building jar: C:\Documents and
Settings\ludo\WebApplication29\dist\WebApplication29.war
do-dist:
dist:
run-deploy:
Starting server Sun Java System Application Server 8
C:\Sun\AppServer81ur1\bin\asadmin.bat start-domain --domaindir
C:\Sun\AppServer81ur1\domains\ domain1
```

```
Distributing C:\Documents and
Settings\ludo\WebApplication29\dist\WebApplication29.war to
[localhost:4848_server]
deployment started : 0%
Deployment of application WebApplication29 completed successfully
Enable of WebApplication29in target server completed successfully
run-display-browser:
Browsing: http://localhost:8080/WebApplication29/
run:
BUILD SUCCESSFUL (total time: 12 seconds)
```

You can use a Web browser (as shown in Figure 9-9) to query the deployed and running Web application for the published WSDL file for this Web service. In our case, the file is <http://localhost:8080/WebApplication29/NewWebService?WSDL>

:

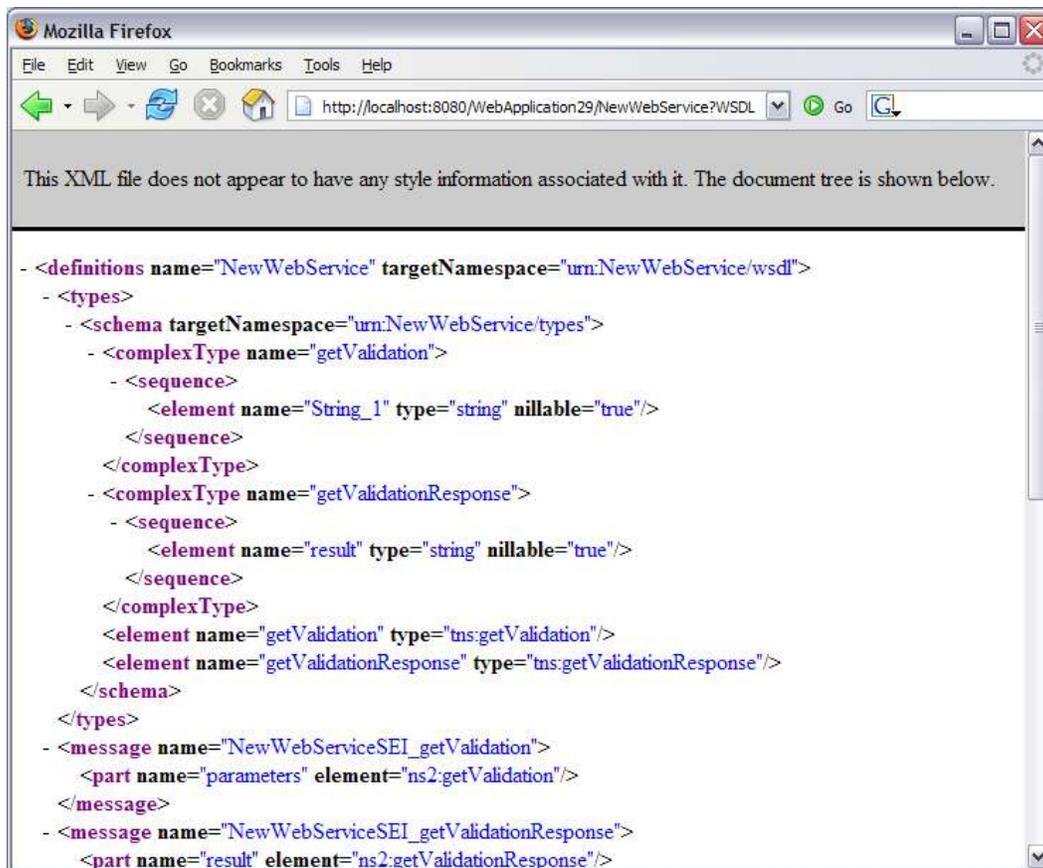


Figure 9-9
Web browser displaying the WSDL file for the Web service

Your Web service is now available to anyone and published so that other applications (J2EE applications, .Net applications, or J2ME applications) can interoperate with its operations.

Creating Web Services from WSDL

You can also create a Web service from a WSDL document. A typical scenario when this is necessary is when business partners formulate the way they will communicate in Web services. The "contract" between them would be the WSDL, in which they would agree on the data and messages that will be exchanged as well as how these messages will be sent and received. This WSDL is then used to implement the Web service.

The elements of a WSDL document can be categorized into **abstract** and **concrete** parts. The *types*, *message*, and *portType* elements describe the data that form the messages sent and received by Web services and clients, as well as the operations that will use these messages. These sections constitute the abstract portion of the WSDL. The *binding* and *service* elements describe the protocols and transport mechanisms that will be used to send and receive the messages, as well as the actual address of the endpoint. This is considered to be the concrete portion of WSDL.

When creating a Web service from WSDL in NetBeans IDE, a new WSDL is created and packaged with the Web service. The abstract portion of the original WSDL is copied to the new one. The concrete portion of the original WSDL is normalized for SOAP binding. Since the JAX-RPC runtime that is used in NetBeans only supports SOAP over HTTP binding, the WSDL is searched for the first occurrence of this binding. If found, it is copied into the new WSDL. If not, a SOAP/HTTP binding is created for the first `portType` defined in the original WSDL. Thus the Web service created from WSDL in NetBeans will always have exactly one SOAP binding and one service port corresponding to that binding. The service element that is added will be named according to the Web service name specified in the wizard, replacing the service element in the original WSDL.

To create a Web service from WSDL, click the From Local WSDL File button or the From WSDL URL button in the wizard, depending on the source of the WSDL document. When the Web service is created, classes for the service endpoint interface and implementation bean will be created. These classes will contain all the operations described in the WSDL. The implementation bean class will be displayed in the Source Editor and you may then enter code to implement these operations. If the WSDL describes operations that use complex types, classes for these types (known as value types) are also generated so that you may use them in your implementation code.

Since the WSDL document governs the interface to the Web service, you may not add new operations to Web services that are created from WSDL, because these operations will not be reflected back in the WSDL.

Note that WSDLs that import other WSDLs are not supported by this facility.

Web Service Types

By default, NetBeans creates "document/literal" Web services. This refers to the way the SOAP message is sent over the wire and is expressed in the SOAP binding part of the WSDL. The document/literal nomenclature comes from the way SOAP messages are described in the SOAP binding of a WSDL document, namely its `style` and `use` attributes.

The `style` attribute refers to the formatting of the SOAP message. This basically refers to what the SOAP body will contain when it is sent over the wire.

There are two ways to format a SOAP message, “RPC” or document. When the style is RPC, the contents of the SOAP body are dictated by the rules of the SOAP specification. That is, the first child element is named after the operation, and its children are interpreted as the parameters of the method call. The endpoint will interpret this as an XML representation of a method call (that is, a remote procedure call). On the other hand, if the style attribute is `document`, the SOAP body consists of arbitrary XML, not constrained by any rules and able to contain whatever is agreed upon by the sender and receiver.

The `use` attribute describes how data is converted between XML and software objects, that is, how it is serialized to XML.

If the `use` attribute is “encoded”, it means that the rules to encode/decode the data is dictated by some rules for encoding, the most common of which is the SOAP encoding specified in the SOAP specification. Section 5 of the SOAP specification defines how data should be serialized to XML. In this case, Web services or clients see data in terms of objects.

If the `use` attribute is “literal”, the rules for encoding the data is dictated by an XML schema. There are no encoding rules, and the Web service or client see the data in terms of XML. Here, the developer does the work of parsing the XML to search for needed data.

Thus, document/literal Web services are typically used to exchange business documents while RPC/encoded Web services are typically used to invoke remote objects. For this reason, document/literal is preferred over RPC/encoded because in “document/literal”, you have full control of the messages that are being exchanged. The WS-I Basic Profile, which is a specification for Web services interoperability, does not support RPC/encoded Web services.

The following are the advantages of document/literal over RPC/encoded Web services:

- Document/literal formatting is more interoperable than RPC/encoded formatting because RPC formatting tends to bind the messages to programming language structures.
- Document/literal Web services scale better than RPC/encoded because of the overhead involved in marshalling and unmarshalling RPC data.
- Document-centric Web services lend themselves to validation of the documents being exchanged. This is cumbersome to do with RPC style services.

Implementing Web Services within an EJB Module

To implement a Web service in an EJB module, you will use a similar Web service wizard described for the Web application. Most of the artifacts that are created and the deployment descriptor entries that are added for the module are similar to those in a Web application. You will follow the same procedure for adding SOAP message handlers as well as configuring them in Web services.

One significant difference is the implementation bean of a Web service in an EJB module. JSR 109 requires that Web services be implemented in the EJB module as stateless session beans. Thus, the implementation of the Web service operations in an EJB module is contained in the session bean class. The module's deployment descriptor will have a stateless session bean entry, but this will declare an endpoint interface instead of a local or remote interface. Also, a Web service in an EJB module does not have a home interface.

Once you have created a Web service within an EJB module, you will see the Web service logical node in the Projects window. The source code you will manipulate is the stateless session bean implementation class. The developer experience is completely similar to the development of a Web service with a Web application.

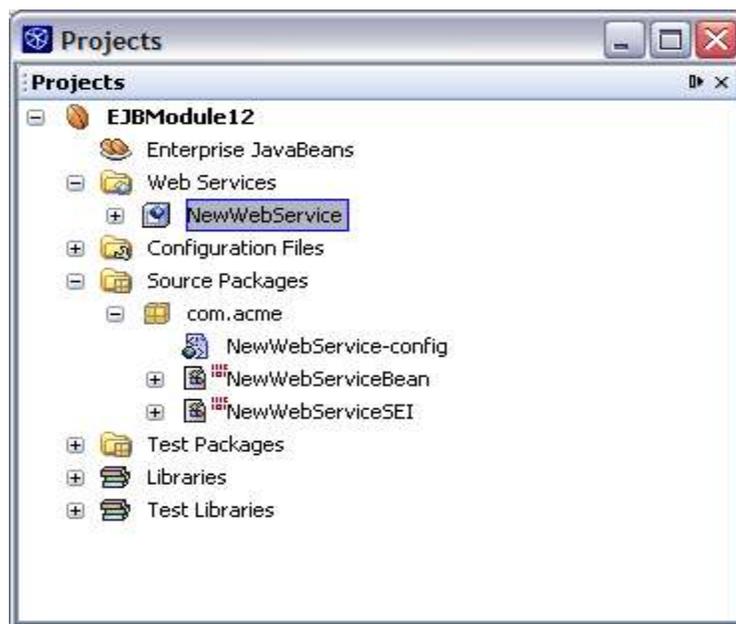


Figure 9-10
Web service within an EJB module

In the Source Packages node, you can see the service bean class as well as the Service Endpoint Interface (SEI) and the service XML config file.

Testing Web Services

NetBeans IDE has a built-in test environment for publishing Web services, either for those created by you and deployed within a Web application or a J2EE application, or those published externally. All you need is to access to the WSDL file for this Web service. You can use the Web Services Registry tool from the Web Services node in the IDE's Runtime window (as shown in Figure 9-11) to register the Web services in the IDE.

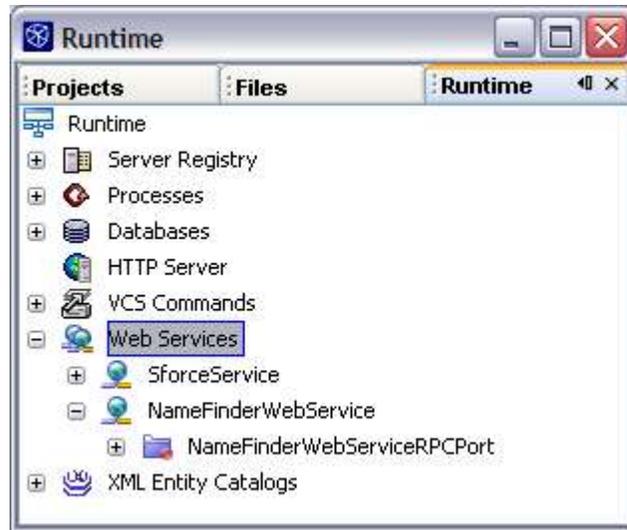


Figure 9-11
Web Services registry in the Runtime window

To add a Web service to the registry:

1. Right-click the Web Services node in the Runtime window and choose Add Web Service to activate the wizard.
2. In the wizard (as shown in Figure 9-12) enter the WSDL file (either as a URL or local file).

Once you specify the file and click the Add button, the service's operations are available as nodes in the registry and you can use the Test Operation command.

To test the Web service operation:

1. In the Runtime window, expand the Web Services node and navigate to the node for the operation you want to test (this node should have no subnodes), right-click that node and choose Test Operation (as shown in Figure 9-13).
2. In the wizard that appears (as shown in Figure 9-14), enter any input parameters for this operation and click Submit.

The Web service operation is called, and the output parameters are displayed in the Result area.

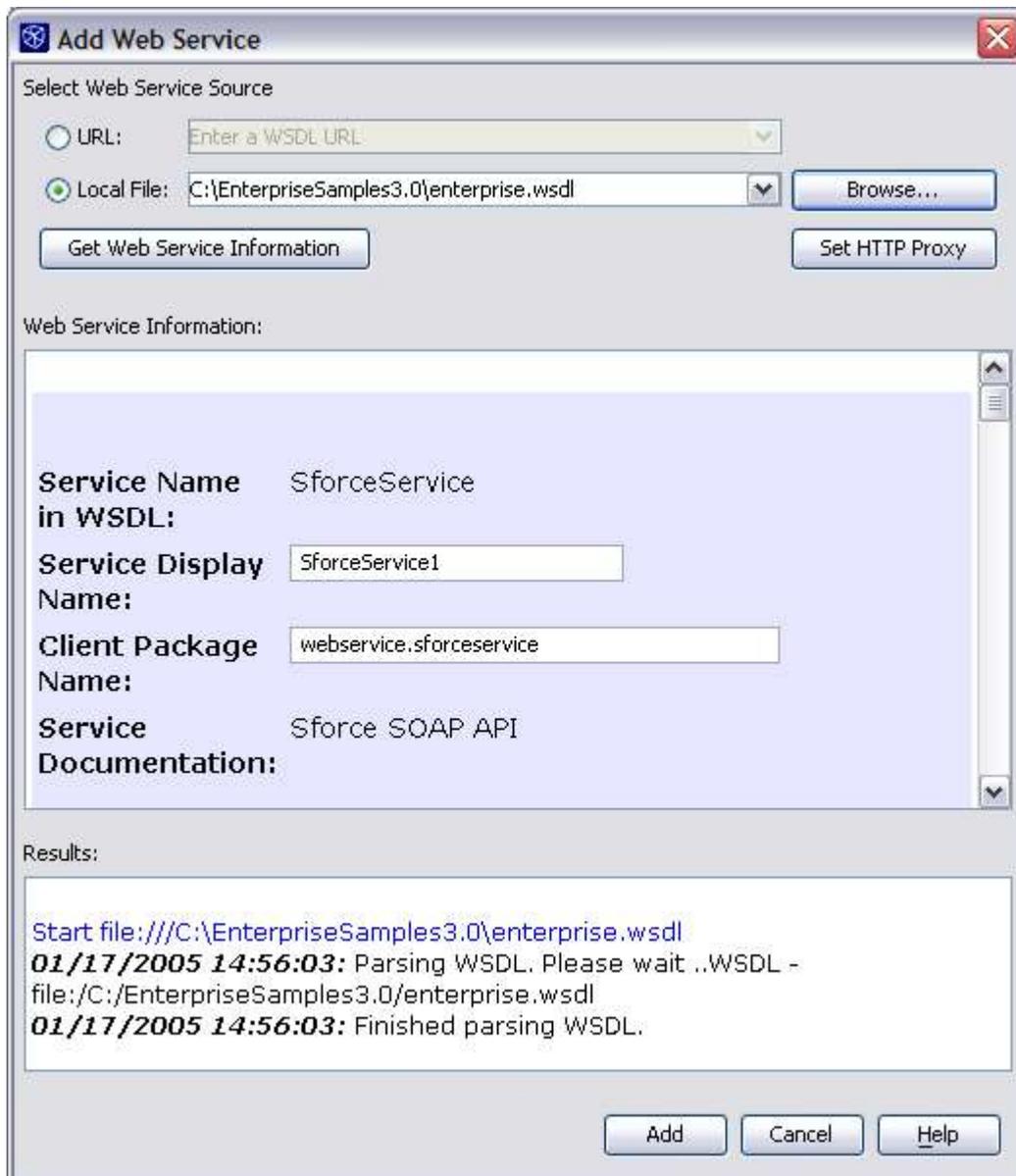


Figure 9-12
Add Web Service wizard

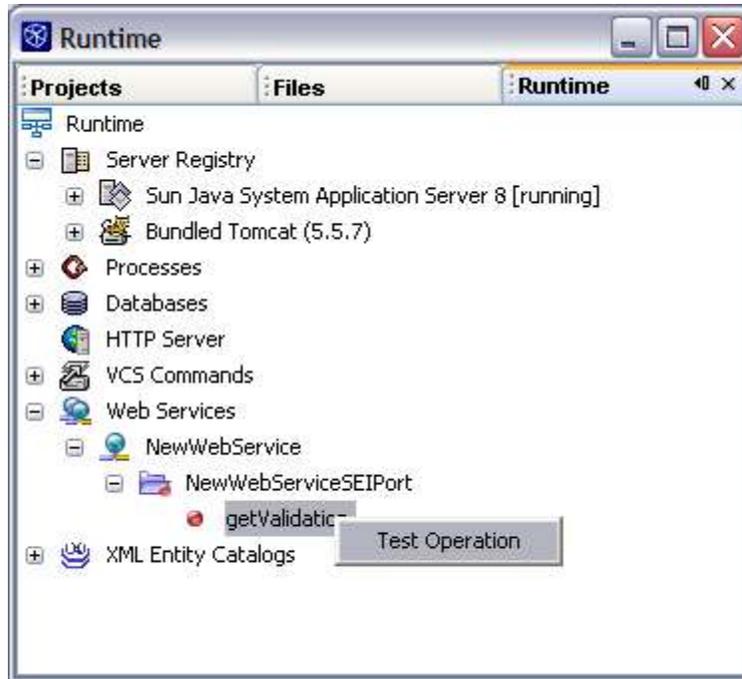
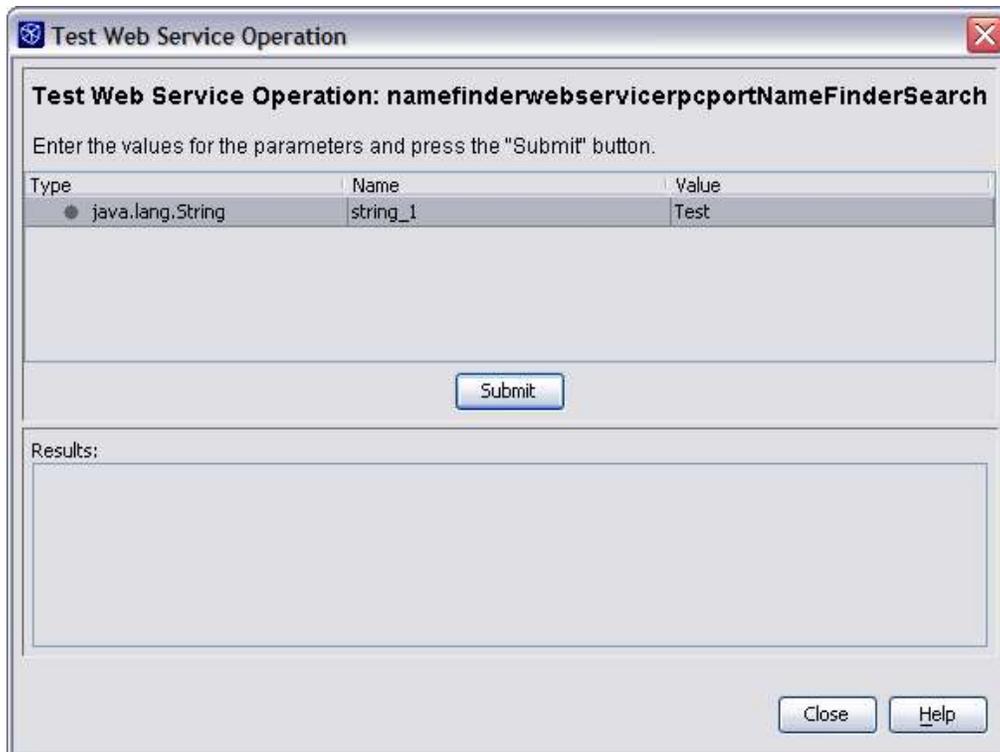


Figure 9-13

Choosing the Test Operation command on a Web Service in the Runtime window

Figure 9-14

The Test Web Service Operation dialog box



Adding Message Handlers to a Web Service

NetBeans IDE makes it easy to develop J2EE Web services and clients because it shields application developers from the underlying SOAP messages. Instead of writing code to build and parse SOAP messages, application developers merely implement the service methods and invoke them from remote clients.

However, there are times when you want to add functionality to Web service applications without having to change the Web service or client code. For example, you might want to encrypt remote calls at the SOAP message level. SOAP message handlers provide the mechanism for adding this functionality without having to change the business logic. Handlers accomplish this by intercepting the SOAP message as it makes its way between the client and service.

A SOAP message handler is a stateless instance that accesses SOAP messages representing RPC requests, responses, or faults. Tied to service endpoints, handlers enable you to process SOAP messages and to extend the functionality of the service. For a given service endpoint, one or more handlers may reside on the server and client.

A SOAP request is handled as follows:

- The client handler is invoked before the SOAP request is sent to the server.
- The service handler is invoked before the SOAP request is dispatched to the service endpoint.

A SOAP response is processed in this order:

1. The service handler is invoked before the SOAP response is sent back to the client.

2. The client handler is invoked before the SOAP response is transformed into a Java method return and passed back to the client program.

To create a message handler in a Web application in the IDE:

1. Right-click the Web application's node in the Projects window and choose New | Message Handler.
2. On the New Message Handler page of the New File wizard (as shown in Figure 9-15), enter a name and package for the handler and click Finish.

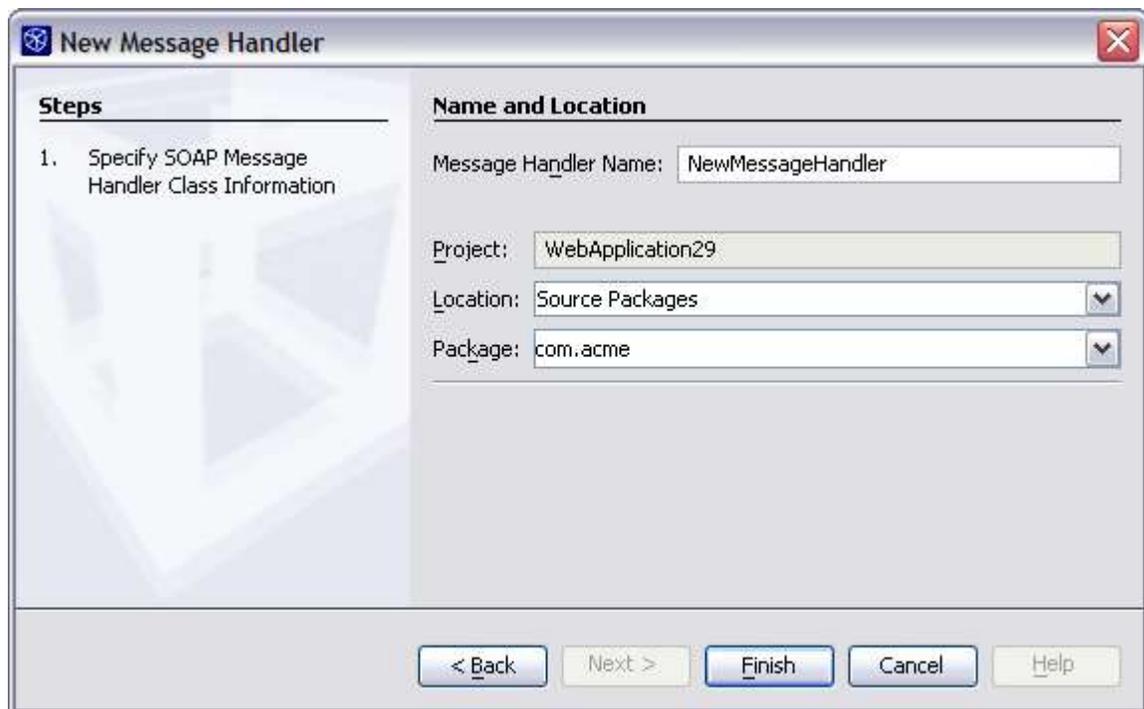


Figure 9-15

New Message Handler page of the New File wizard

The new Java file created contains the core code for the message handler as shown in the code sample below.

One interesting method here is `handleRequest(MessageContext context)`, which is called before the SOAP message is dispatched to the endpoint. The generated handler class provides a default implementation of this method (as an example) which prints out the contents of the SOAP body plus some date information. Note that the `MessageContext` parameter provides a context for obtaining the transmitted SOAP message. You may then use the SAAJ API (SOAP with Attachments API for Java) to access and manipulate the SOAP message.

Another method, `handleResponse(MessageContext context)` is called before the response message is sent back to the caller. This method, together with `handleFault`, provides only the default implementation and it is left to you to provide your own.

```
package com.acme;

import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPBody;
import java.util.Date;

public class NewMessageHandler extends
    javax.xml.rpc.handler.GenericHandler {
    // TODO Change and enhance the handle methods to suit individual
    needs.
    private QName[] headers;
    public void init(HandlerInfo config) {
        headers = config.getHeaders();
    }
    public javax.xml.namespace.QName[] getHeaders() {
        return headers;
    }
    // Currently prints out the contents of the SOAP body plus some
    date information.
    public boolean handleRequest(MessageContext context) {
        try{
            SOAPMessageContext smc = (SOAPMessageContext) context;
            SOAPMessage msg = smc.getMessage();
            SOAPPart sp = msg.getSOAPPart();
            SOAPEnvelope se = sp.getEnvelope();
            SOAPHeader shd = se.getHeader();

            SOAPBody sb = se.getBody();
            java.util.Iterator childElems = sb.getChildElements();
            SOAPElement child;
            StringBuffer message = new StringBuffer();
            while (childElems.hasNext()) {
                child = (SOAPElement) childElems.next();
                message.append(new Date().toString() + "--");
                formLogMessage(child, message);
            }

            System.out.println("Log message: " + message.toString());
        } catch(Exception e){
            e.printStackTrace();
        }
        return true;
    }

    public boolean handleResponse(MessageContext context) {
```

```

        return true;
    }

    public boolean handleFault(MessageContext context) {
        return true;
    }

    public void destroy() {
    }

    private void formLogMessage(SOAPElement child, StringBuffer
message) {
        message.append(child.getElementName().getLocalName());
        message.append(child.getValue() != null ? ":" + child.getValue
() + " " : " ");

        try{
            java.util.Iterator childElems = child.getChildElements();
            while (childElems.hasNext()) {
                Object c = childElems.next();
                if(c instanceof SOAPElement)
                    formLogMessage((SOAPElement)c, message);
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

Once this message handler is created, you need to associate it with your Web service. To associate a message handler with a Web service:

1. Right-click the the Web service in the Projects window and choose Configure Handlers.
2. In the Configure SOAP Message Handlers dialog box (as shown in Figure 9-16), click Add and navigate to and select the message handler.

Figure 9-16
Configure SOAP Message Handlers dialog box



The IDE automatically updates the `webservices.xml` file under the `WEB-INF` directory of your Web application by adding the `<handler>` element.

Below is an example of `webservices.xml` file that the IDE has updated for you. (In general, you do not have to worry about this file at all. The IDE keeps it up to date for you.)

```
<?xml version='1.0' encoding='UTF-8' ?>
<webservices xmlns='http://java.sun.com/xml/ns/j2ee' version='1.1'>
  <webservice-description>
    <webservice-description-name>NewWebService</webservice-
description-name>
    <wsdl-file>WEB-INF/wsdl/NewWebService.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/wsdl/NewWebService-
mapping.xml</jaxrpc-mapping-file>
    <port-component xmlns:wsdl-port_ns='urn:NewWebService/wsdl'>
      <port-component-name>NewWebService</port-component-
name>
      <wsdl-port>wsdl-port_ns:NewWebServiceSEIPort</wsdl-
port>
      <service-endpoint-
interface>com.acme.NewWebServiceSEI</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>WSServlet_NewWebService</servlet-
link>
      </service-impl-bean>
      <handler>
        <handler-name></handler-name>
        <handler-
class>com.acme.NewMessageHandler</handler-class>
      </handler>
    </port-component>
  </webservice-description>
</webservices>
```

To see the effect of the message handler on the Web service, perform the following steps:

1. Run the Web application by right-clicking its node in the Projects window and choosing Run Project.
2. Add the Web service to the IDE registry by right-clicking the Web service's node in the Projects window and choosing Add to Registry.
3. Switch to the Runtime window of the IDE. Then navigate through the hierarchy of Web service nodes, right-click an operation node, and choose Test Operation (as shown in Figure 9-17).
4. In the Test Web Service Operation (as shown in Figure 9-18), enter the input parameters and click the Submit button.

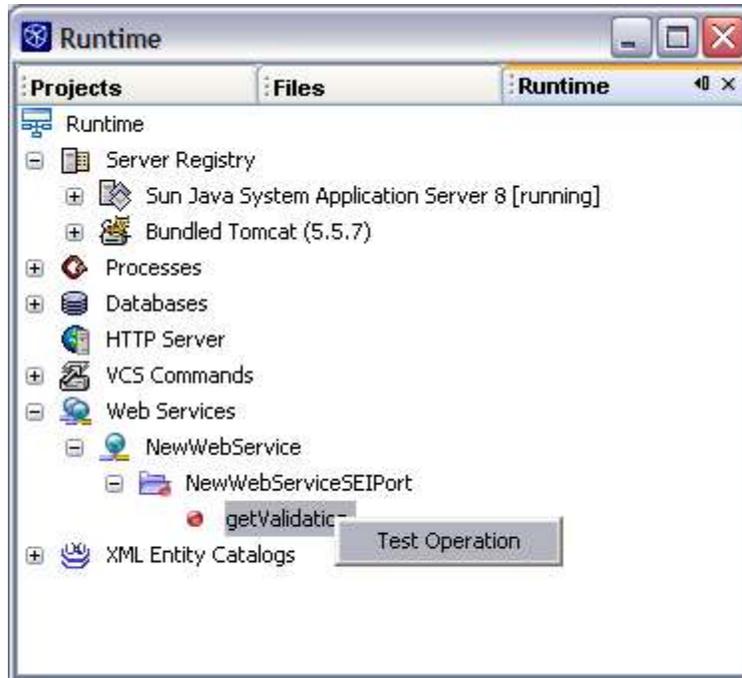


Figure 9-17

Running the Test Operation command on an operation in the Runtime window

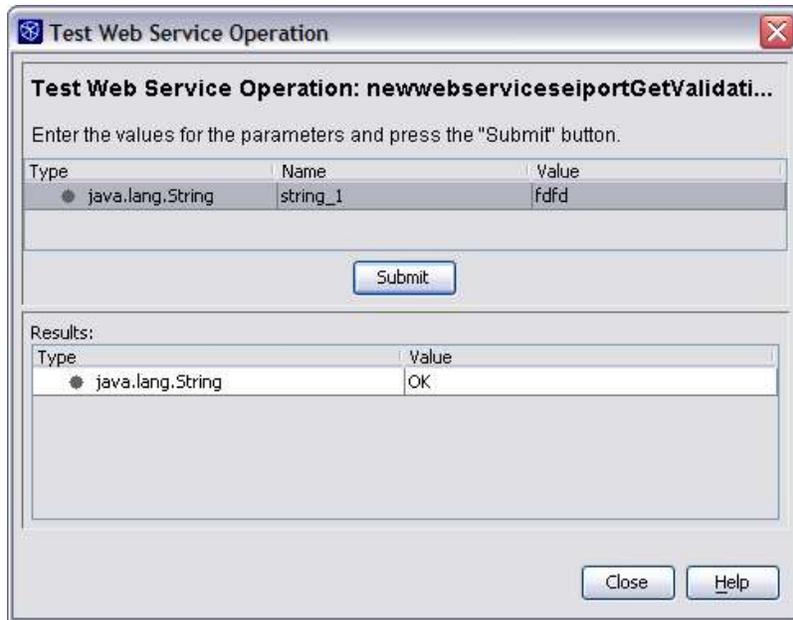


Figure 9-18

*Test Web
Operation dialog box.*

Service

You can then see the handler trace in the application server log file by following these steps. Open the Runtime window, expand the Servers node, right-click the Sun Java System Application Server node, and choose View Server Log.

A trace looking something like the following should be displayed.

```
[#|2005-02-20T16:59:01.293-0800|DPL5306:Servlet Web service Endpoint
[NewWebService] listening at address
[http://129.145.133.80:8080/WebApplication29/NewWebService]|#]
[#|2005-02-20T16:59:02.084-0800|
javax.enterprise.system.tools.deployment|DPL5306:Servlet Web service
Endpoint [NewWebService] listening at address
[http://129.145.133.80:8080/WebApplication29/NewWebService]|#]
[#|2005-02-20T16:59:14.292-0800||javax.enterprise.system.stream.out|Log
message: Sun Feb 20 16:59:14 PST 2005--getValidation String_1:klkl|#]
[#|2005-02-20T17:05:40.297-0800|javax.enterprise.system.stream.out|
Log message: Sun Feb 20 17:05:40 PST 2005--getValidation
String_1:fdfd|#]
```
