1 Accessing .NET Web Services

This document tells you how to develop a **.NET Web Service consumer** (a program that accesses a Microsoft .NET Web Service) with SilverStream eXtend Workbench 2.0. You'll learn about using **jBroker Web** 1.1 and **Workbench tools** to create the client files needed for this kind of Web Service.

The techniques presented here can also be used more generally to develop consumers for any kind of Web Service that uses **complex data types** for input or output.

📖 For more on Web Services, jBroker Web, and Workbench tools, see the Workbench help.

Basics

You can use jBroker Web and Workbench tools to generate the code needed for a **Java-based consumer program** to access any **standard (SOAP-based) Web Service**. The generated code handles all HTTP SOAP processing under the covers, enabling the consumer program to call the Web Service as a **Java remote object** (using RMI) and invoke its methods.

In this architecture, **type mappings** are necessary to convert parameters and return values back and forth between XML (in the SOAP messages) and Java (in the client code). These mappings are generated automatically whenever possible. However, if your target Web Service passes complex types, you'll usually need to write some additional mapping code yourself. This applies for .NET and other **document-style** Web Services (as well as some **RPC-style** Web Services).

jBroker Web provides the underlying support for type mapping through its **compilers** and **API**. You should read the jBroker Web documentation to get familiar with these before you begin development.

Steps

The process of developing your .NET consumer program involves:

1. Preparing for development by setting up your project
2. Providing a WSDL file that describes the .NET Web Service you want to access
3. Writing the type-mapping files that handle the Web Service data and its conversion on the client

4. Generating the consumer files by using the jBroker Web compilers
5. Examining the generated files (which include a remote interface and stub class that facilitate the Web Service access)
6. Writing your client code to call Web Service methods via the generated files
7. Building the project to compile all of the classes you've written and generated
8. Running the consumer program to test how it works

Preparing for development

To prepare for developing a .NET Web Service consumer, you:

1. Set up an appropriate **project** in Workbench.

   The type of project you should create depends on how you ultimately plan to use the consumer code that the jBroker Web compilers will generate. For instance:

If you plan to use the consumer code in	You should create
A standard Java application	A JAR project
A J2EE application client	A CAR project
A JSP page or servlet	A WAR project
An Enterprise JavaBean	An EJB JAR project

2. Add **jbroker-web.jar** to your project.

   The jbroker-web.jar file contains the SilverStream Web Service API classes needed by the generated consumer code at runtime. You'll find it in the Workbench **compilelib** directory.

   In a WAR project, for example, you'd add jbroker-web.jar to the WEB-INF/lib directory.

3. Edit the **classpath** of your project so you can compile your consumer classes once they're generated and edited. You'll need to include:

   - jbroker-web.jar
   - Any application-specific entries

   For J2EE projects, you'll also need j2ee_api_1_*n*.jar (it's included automatically when you create a J2EE project in Workbench).

Providing a WSDL file

To generate consumer code, you'll need to provide the jBroker Web compilers with a WSDL file that describes the target .NET Web Service. It's a good idea to obtain the file location or URL of this WSDL file early on.

These are common scenarios:

- **For a .NET Web Service developed in your organization**, you might have the WSDL file on your file system or even in your project.
- **For an external .NET Web Service**, you should be able to get the WSDL file's URL from the appropriate Web site or registry.

Example: WSDL file for Autoloan .NET Web Service

Suppose you want to generate consumer code to use the **Autoloan** .NET Web Service, which is listed on the XMethods public registry under the name **Equated Monthly Instalment (EMI) Calculator**. That Web Service calculates and returns the monthly loan payment for a given term (number of months), interest rate, and loan amount.

In this case, you can go to the Web site www.xmethods.net to discover the URL for the corresponding WSDL file:

```
http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
```

When you provide this URL to the jBroker Web compilers, they will read the WSDL file to learn what they need to know about the Autoloan Web Service:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
             xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
             xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:s0="http://circle24.com/webservices/"
             targetNamespace="http://circle24.com/webservices/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema attributeFormDefault="qualified" elementFormDefault="qualified"
              targetNamespace="http://circle24.com/webservices/">
      <s:element name="Calculate">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="Months" type="s:double" />
            <s:element minOccurs="1" maxOccurs="1" name="RateOfInterest" type="s:double"
/>
```

```
          <s:element minOccurs="1" maxOccurs="1" name="Amount" type="s:double" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="CalculateResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="CalculateResult" nillable="true"
                     type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="string" nillable="true" type="s:string" />
  </s:schema>
</types>
<message name="CalculateSoapIn">
  <part name="parameters" element="s0:Calculate" />
</message>
<message name="CalculateSoapOut">
  <part name="parameters" element="s0:CalculateResponse" />
</message>
<message name="CalculateHttpGetIn">
  <part name="Months" type="s:string" />
  <part name="RateOfInterest" type="s:string" />
  <part name="Amount" type="s:string" />
</message>
<message name="CalculateHttpGetOut">
  <part name="Body" element="s0:string" />
</message>
<message name="CalculateHttpPostIn">
  <part name="Months" type="s:string" />
  <part name="RateOfInterest" type="s:string" />
  <part name="Amount" type="s:string" />
</message>
<message name="CalculateHttpPostOut">
  <part name="Body" element="s0:string" />
</message>
<portType name="AutoloanSoap">
  <operation name="Calculate">
    <input message="s0:CalculateSoapIn" />
    <output message="s0:CalculateSoapOut" />
  </operation>
</portType>
<portType name="AutoloanHttpGet">
  <operation name="Calculate">
    <input message="s0:CalculateHttpGetIn" />
    <output message="s0:CalculateHttpGetOut" />
  </operation>
</portType>
<portType name="AutoloanHttpPost">
```

```
  <operation name="Calculate">
    <input message="s0:CalculateHttpPostIn" />
    <output message="s0:CalculateHttpPostOut" />
  </operation>
</portType>
<binding name="AutoloanSoap" type="s0:AutoloanSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                style="document" />
  <operation name="Calculate">
    <soap:operation soapAction="http://circle24.com/webservices/Calculate"
                    style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<binding name="AutoloanHttpGet" type="s0:AutoloanHttpGet">
  <http:binding verb="GET" />
  <operation name="Calculate">
    <http:operation location="/Calculate" />
    <input>
      <http:urlEncoded />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<binding name="AutoloanHttpPost" type="s0:AutoloanHttpPost">
  <http:binding verb="POST" />
  <operation name="Calculate">
    <http:operation location="/Calculate" />
    <input>
      <mime:content type="application/x-www-form-urlencoded" />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<service name="Autoloan">
  <documentation>This Web Service mimics a Simple Autoloan calculator.</documentation>
  <port name="AutoloanSoap" binding="s0:AutoloanSoap">
    <soap:address location="http://upload.eraserver.net/circle24/autoloan.asmx" />
  </port>
  <port name="AutoloanHttpGet" binding="s0:AutoloanHttpGet">
```

```
      <http:address location="http://upload.eraserver.net/circle24/autoloan.asmx" />
    </port>
    <port name="AutoloanHttpPost" binding="s0:AutoloanHttpPost">
      <http:address location="http://upload.eraserver.net/circle24/autoloan.asmx" />
    </port>
  </service>
</definitions>
```

Understanding the WSDL

In the Autoloan WSDL, you can ignore the definitions for **HttpGet** and **HttpPost** (including message, portType, binding, and service port). Only the **Soap** definitions apply to the Web Service consumer program you're developing.

Notice that this Web Service exposes one method named **Calculate()**. It takes a **Calculate** object containing three doubles (Months, RateOfInterest, and Amount) and returns a **CalculateResponse** object containing one string (CalculateResult).

Pay particular attention to the data definitions in the **types** section (for Calculate and CalculateResponse). You'll need to set up mappings for these types later on.

If you look in the **binding** section for AutoloanSoap, you'll see where this Web Service is defined as document-style (as opposed to RPC-style). This is true of all .NET Web Services. The significance of this style attribute is as follows:

- **Document-style** specifies that the SOAP request and response messages pass XML documents.
- **RPC-style** specifies that SOAP request messages pass input parameters and SOAP response messages pass return values.

Writing the type-mapping files

Once you've set up a project and located the appropriate WSDL file, you need to write type-mapping files for your .NET Web Service consumer. These files represent the Web Service's data on the client, including both input and output objects. They also determine how that data is to be converted back and forth between XML and Java.

There are three kinds of type-mapping files you need:

File	Description
Type class	For each complex data type defined in the **types** section of the WSDL, you need to write a corresponding Java class. It can be either of the following: • A **Java class with public fields** • A **JavaBean** with getter and setter methods for the fields
Marshaler class	Each type class requires a corresponding marshaler class that performs the Java-to-XML data conversion (**serialization**) and the XML-to-Java data conversion (**deserialization**). Typically, you can use one of the general-purpose marshaler classes provided by the jBroker Web API (in com.sssw.jbroker.web.mapping): • **PublicFieldsMarshaler** for Java classes with public fields • **BeanMarshaler** for JavaBeans Alternatively, you can write your own custom marshaler classes if necessary.
Mapping properties file	You provide a text file named **xmlrpc.type.mappings** that specifies the details of the type mappings. For each type to be mapped, you include a line of information in the following format: `name = class serializer deserializer namespace local-name schema-location` This information tells the jBroker Web compilers and runtime about your type mappings and how they should be handled.

Add all of the type-mapping files you write to your project. Then compile the classes.

Example: type-mapping files for Autoloan .NET Web Service

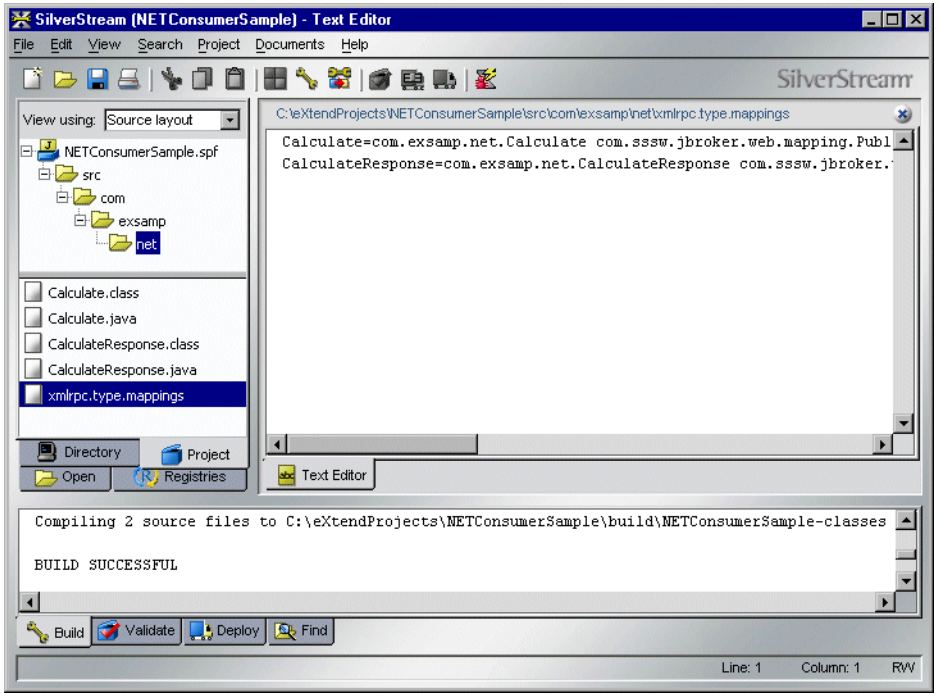
The consumer program for the Autoloan .NET Web Service requires the following type-mapping files:

- Calculate.java
- CalculateResponse.java

- xmlrpc.type.mappings

In this case, Calculate and CalculateResponse are coded as Java classes with public fields. That means the jBroker Web **PublicFieldsMarshaler** class can be used with each. There's no need to write custom marshaler classes.

These files are added to the **src** directory of the **NETConsumerSample** project, in the package **com.exsamp.net**:



NOTE In this example, the compiled Calculate and CalculateResponse **.class files** have been manually copied from the project's build directory tree to the src directory tree (where the xmlrpc.type.mappings file is located). This will enable the jBroker Web compilers to find them later on.

Calculate.java

This is the Java class that represents the Web Service's Calculate type:

```
package com.exsamp.net;

public class Calculate
```

```
{
  public double Months;
  public double RateOfInterest;
  public double Amount;
}
```

CalculateResponse.java

This is the Java class that represents the Web Service's CalculateResponse type:

```
package com.exsamp.net;

public class CalculateResponse
{
  public String CalculateResult;
  public String toString() { return CalculateResult; }
}
```

xmlrpc.type.mappings

Here's the properties file that specifies the mappings for the Calculate and CalculateResponse types (there are two lines; the snippet below shows them with line-wrapping for readability):

```
Calculate=com.exsamp.net.Calculate com.sssw.jbroker.web.mapping.PublicFieldsMarshaler
  com.sssw.jbroker.web.mapping.PublicFieldsMarshaler http://circle24.com/webservices/
  Calculate none
CalculateResponse=com.exsamp.net.CalculateResponse
com.sssw.jbroker.web.mapping.PublicFieldsMarshaler
  com.sssw.jbroker.web.mapping.PublicFieldsMarshaler http://circle24.com/webservices/
  CalculateResponse none
```

Notice that PublicFieldsMarshaler is specified as both the serializer and deserializer in each case.

Generating the consumer files

Now you're ready to use the jBroker Web compilers to generate the main Java files for your .NET Web Service consumer. You'll invoke the **wsdl2java** compiler to read the Web Service's WSDL file and create a corresponding remote interface. The **rmi2soap** compiler is then invoked automatically to create a stub class that handles the SOAP access to the Web Service.

To start generating:

1. Go to the **command prompt** in your operating system.

2. Type the **wsdl2java command line** using the syntax specified in the jBroker Web documentation.

   Make sure you include your type-mapping files on the **classpath** so the compilers will use them.

   TIP The jBroker Web compilers are located in the Workbench **bin\win32** directory.

Example: invoking wsdl2java for Autoloan .NET Web Service

Here's the command line used to invoke the wsdl2java compiler for the Autoloan consumer example (in the snippet below, the wsdl2java command line is shown with line-wrapping for readability):

```
cd c:\eXtendProjects\NETConsumerSample\src

wsdl2java -noskel -notie -keep -ds . -package com.exsamp.net -d .
  -classpath com\exsamp\net;. http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
```

Notice that:

- The **-noskel** and **-notie** options are specified to prevent the generation of Web Service implementation files (which don't apply when you're developing a consumer).
- The **-keep** option is specified to retain the generated stub source file (in this case, _AutoloanSoap_ServiceStub.java) after compilation so you can study it.
- The **-ds**, **-package**, and **-d** options specify the directories and paths for the generated source and class files.
- The **-classpath** option specifies the location of the type-mapping files (in this case, it's where xmlrpc.type.mappings, Calculate.class, and CalculateResponse.class are stored).
- The **URL of the WSDL file** is specified at the end of the command line.

Examining the generated files

The jBroker Web compilers generate the following files for your .NET Web Service consumer. Add these files to your project in Workbench (but don't edit them).

What the jBroker Web compilers generate	Details
Remote interface	*xxx*.**java** An interface that extends java.rmi.Remote and declares the methods exposed by the target Web Service (as determined from the WSDL file). The generated stub class _*xxx*_ServiceStub implements this interface to support method calls for the Web Service. **NOTE** When generating file names, the compilers fill in the *xxx* portion based on the portType name specified in the WSDL file. (For instance, if the portType name is **TemperaturePortType**, then *xxx* will be **Temperature**.)
Stub class	**_*xxx*_ServiceStub.java** Facilitates method calls from a Java-based consumer to the target Web Service. _*xxx*_ServiceStub implements the generated remote interface by sending an appropriate HTTP SOAP request for each method call. When you write your client code, it will need to instantiate _*xxx*_ServiceStub (via a JNDI lookup) and call Web Service methods on the resulting object.

In some cases, the jBroker Web compilers may generate additional files to support requirements specific to your application, such as:

- Faults
- Multiple portType definitions

 For more information, see the jBroker Web documentation.

Example: generated consumer files for Autoloan .NET Web Service

Here are the files generated by the jBroker Web compilers for the Autoloan consumer example:

- AutoloanSoap.java (and AutoloanSoap.class)
- _AutoloanSoap_ServiceStub.java (and _AutoloanSoap_ServiceStub.class)

- AutoloanHttpGet.java (and AutoloanHttpGet.class)

  You can ignore these files. They are generated from the AutoloanHttpGet portType in the WSDL file, which doesn't apply to your SOAP consumer.

- AutoloanHttpPost.java (and AutoloanHttpPost.class)

  You can ignore these files. They are generated from the AutoloanHttpPost portType in the WSDL file, which doesn't apply to your SOAP consumer.

NOTE Once you're done running the jBroker Web compilers and return to Workbench, you can remove all of the **.class files** from your project's src directory tree. When you build the .java files under src, Workbench will generate the .class files in a separate build directory tree.

AutoloanSoap.java

This is the remote interface used by the stub class to support method calls for the Autoloan .NET Web Service.

```
// Generated from http://upload.eraserver.net/circle24/autoloan.asmx?wsdl
// On Tue Mar 05 09:15:02 EST 2002

package com.exsamp.net;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface AutoloanSoap extends Remote
{
    com.exsamp.net.CalculateResponse Calculate(com.exsamp.net.Calculate parameters)
      throws RemoteException;
}
```

_AutoloanSoap_ServiceStub.java

This is the stub class. It passes method calls to the Autoloan .NET Web Service as HTTP SOAP requests.

```
// Tue Mar 05 09:15:02 EST 2002

package com.exsamp.net;

public class _AutoloanSoap_ServiceStub extends com.sssw.jbroker.web.portable.Stub
    implements com.exsamp.net.AutoloanSoap
{
    private static com.sssw.jbroker.web.QName _portType =
        new com.sssw.jbroker.web.QName("http://circle24.com/webservices/",
```

```
                                    "AutoloanSoap");

private static final com.sssw.jbroker.web.Binding[] _bindings =
    new com.sssw.jbroker.web.Binding[] {
        new com.sssw.jbroker.web.Binding("soap",
          "http://upload.eraserver.net/circle24/autoloan.asmx"),
};

public _AutoloanSoap_ServiceStub()
{
    super(_portType, _bindings);
    _setProperty("xmlrpc.schema.uri", "http://www.w3.org/2001/XMLSchema".intern());
    _setProperty("version", "1.1");
}

private static final com.sssw.jbroker.web.portable.RequestProperty[]
    Calculate_props = new com.sssw.jbroker.web.portable.RequestProperty[] {
    new com.sssw.jbroker.web.portable.RequestProperty("SOAPAction",
      "\"http://circle24.com/webservices/Calculate\""),
};

public com.exsamp.net.CalculateResponse Calculate(com.exsamp.net.Calculate _arg0)
    throws java.rmi.RemoteException
{
    com.sssw.jbroker.web.portable.ClientResponse in = null;

    try {
        // create an output stream
        com.sssw.jbroker.web.portable.ClientRequest out =
            _request("Calculate", true, "literal", true, "null");
        out.setProperties(Calculate_props);
        Object arg = null;

        // marshal the parameters
        arg = _arg0;
        out.writeObject(arg, "http://circle24.com/webservices/", "Calculate");

        // do the invocation
        in = _invoke(out);

        // return
        com.exsamp.net.CalculateResponse ret = null;
        try {
            ret = (com.exsamp.net.CalculateResponse)
            in.readObject(com.exsamp.net.CalculateResponse.class,
              "http://circle24.com/webservices/", "CalculateResponse");
        } catch(java.io.EOFException eofExc) {
            ret = null;
        }
```

```
            return ret;

        } catch (java.lang.Throwable t) {

            if (t instanceof com.sssw.jbroker.web.ServiceException) {
                com.sssw.jbroker.web.ServiceException sex =
                    (com.sssw.jbroker.web.ServiceException) t;
                if (sex.getTargetException() != null)
                    t = sex.getTargetException();
            }

            // map to remote exception
            throw com.sssw.jbroker.web.ServiceException.mapToRemote(t);
        }
    }
}
```

Writing your client code

No matter what kind of consumer you are developing for a .NET Web Service, your client program needs to do the following:

1. **Instantiate the stub class** (*_xxx*_ServiceStub) via a JNDI lookup
2. **Enable your type mappings** for use by the stub object instance
3. **Call Web Service methods** on the stub object instance

When you write the class(es) for your client program, include them in your project along with the type-mapping files, generated remote interface, and generated stub class.

Example: client program for Autoloan .NET Web Service

Here's the client code written for the Autoloan consumer example. You can base your own client program on what you see here.

```
package com.exsamp.net;

import javax.naming.InitialContext;
import com.sssw.jbroker.web.ServiceObject;
import com.sssw.jbroker.web.mapping.TypeMapper;

public class Client
{
    public static void main(String[] args) throws Exception
    {
    // Instantiate the generated stub class via JNDI lookup.
```

```
    // Optionally let user override the binding in the stub by
    // passing in a binding (Web Service address) at runtime.
    InitialContext ctx = new InitialContext();
    String lookup = "xmlrpc:soap:com.exsamp.net.AutoloanSoap";
    if (args.length > 0)
        lookup += "@" + args[0];
    AutoloanSoap loan = (AutoloanSoap) ctx.lookup(lookup);

    // Create and set a type mapper to be used by the stub
    // object instance. Get the mappings to use from your
    // xmlrpc.type.mappings file.
    TypeMapper mapper = new TypeMapper();
    mapper.importMappings("xmlrpc.type.mappings");
    ((ServiceObject)loan)._setTypeMapper(mapper);

    // Instantiate and populate the type class Calculate used
    // for input to the Web Service method.
    Calculate calc = new Calculate();
    calc.Months = 24;
    calc.RateOfInterest = 8;
    calc.Amount = 15000;

    // Call the Web Service method Calculate() via the stub
    // object instance. Then print the result (from the output
    // object CalculateResponse).
    System.out.println(loan.Calculate(calc));
    }
}
```

 For more detailed information on writing code to use stubs and type mappers, see the jBroker Web documentation.

Building the project

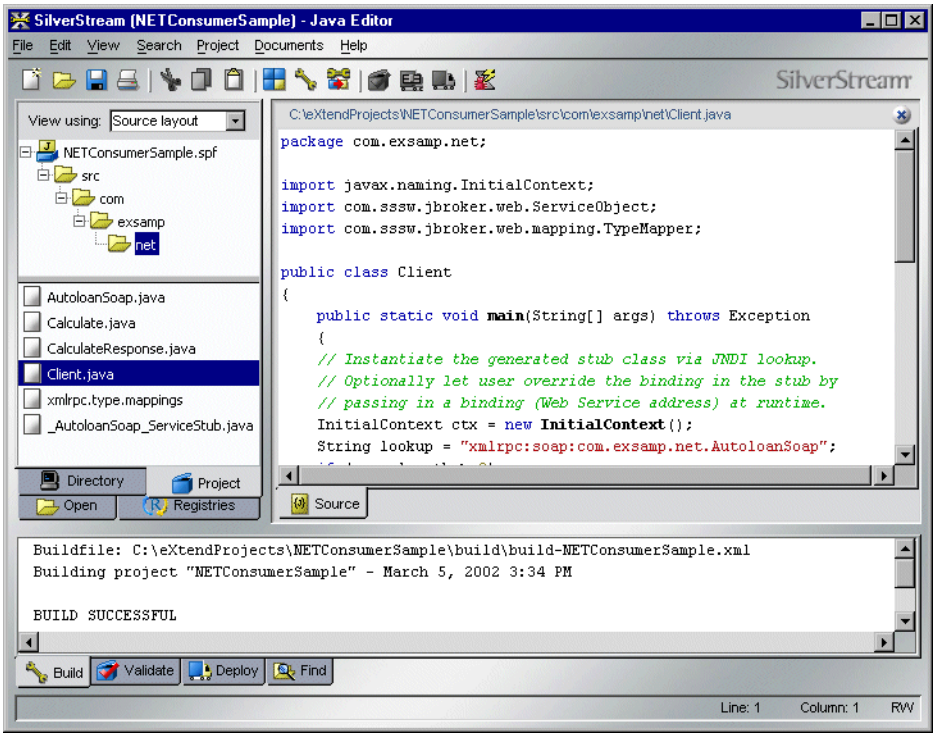
At this point, you should be ready to build the project for your .NET Web Service consumer. Building this project in Workbench compiles all of the following:

- Your type-mapping classes
- The generated remote interface and stub class
- The class(es) for your client program

Workbench puts the compiled .class files in your project's build directory tree.

Example: completed consumer project for Autoloan .NET Web Service

Here's the finished state of the Autoloan consumer example:



Because this consumer involves a simple Java client program, it is now ready for testing. You don't need to deploy anything first. (For other kinds of consumers, you may need to deploy prior to testing.)

Running the consumer program

If your Web Service consumer is a standard Java client program, you can test it in Workbench by using the **Web Service Wizard Client Runner**. This facility lists the client programs in your current project and lets you select one to execute. For each run, it automatically sets the classpath to include all required files and lets you supply command-line arguments.

NOTE Before you use the Client Runner to test your .NET Web Service consumer, update the project's classpath (via **Project**>**Project Settings**) to include the directory that contains your **xmlrpc.type.mappings** file. (For the NETConsumerSample example, you'd add the project's src\com\exsamp\net directory.) This enables the jBroker Web runtime to find that file.

➢ To use the Client Runner:

1. Open the **project** that contains the compiled client class you want to run.
2. Select **Project**>**Run Web Service Client Class** to display the Client Runner window.
3. Select a client from the **Client class to run** dropdown.

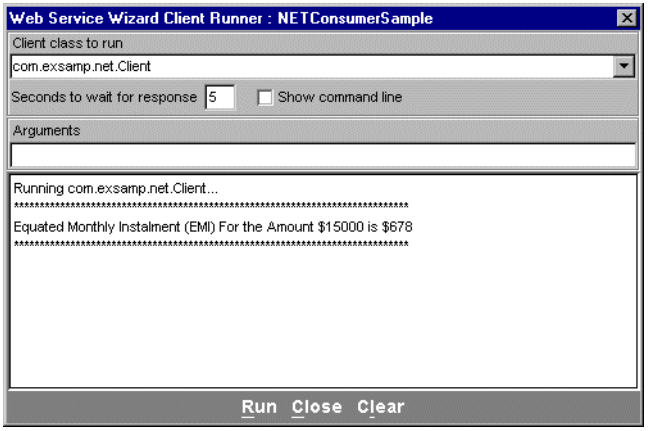
   This dropdown lists every compiled class in your project that has a main() method.
4. Type the number of **Seconds to wait for response** from the Web Service.

   You might need to increase this number if Web Service requests from your client time out before getting a response.
5. Check **Show command line** if you want to:
   - See the complete command line that the Client Runner uses to execute your client (it will appear in the display console portion of the window after you click Run)
   - Optionally copy that command line to the system clipboard by clicking **Copy command line** (after a run)
6. Type any command-line **Arguments** required by your client (use a space to separate each argument).

   NOTE In Workbench 2.0, there's a known issue that prevents the Client Runner from parsing multiple command-line arguments. It passes them as a single String to your client's args[0].
7. Click **Run** to execute your client and see its output in the display console portion of the window.

Example: testing the client for Autoloan .NET Web Service

Here's the result of calling the Autoloan .NET Web Service by executing the project's Client class in the Client Runner:


Web Service Wizard Client Runner : NETConsumerSample
Client class to run
com.exsamp.net.Client
Seconds to wait for response
5
Show command line
Arguments
Running com.exsamp.net.Client...
****
Equated Monthly Instalment (EMI) For the Amount $15000 is $678
****
Run
Close
Clear