Pace University

Web Service Workshop Lab Manual

Dr. Lixin Tao

http://csis.pace.edu/~lixin
Computer Science Department
Pace University

April 9, 2010

Table of Contents

1 Lab objectives 1
2 Lab design 1
3 Lab environment 2
4 Development of a Java Web service for squaring an integer 3
5 Development of a Java client application for consuming the Java Web service 7
6 Development of a *Hello* Web service with C# in Microsoft Visual Studio 9
7 Development of a .NET Web client application to consume the .NET Web service 15
8 Development of a Java client application for consuming the .NET Web service 25

1 Lab objectives

This lab manual is designed for *Pace University's* workshop on "Enterprise System Integration with Web Services". The lab is intended for participants who may have limited experience with Java programming. The lab supplements the workshop presentation and its accompanying document "Essential Concepts of Web Services", which you should carefully read before starting to work on this lab.

After studying my document "Essential Concepts of Web Services" and completing the lab session described in this manual, you should have a good understanding of the motivations, major concepts, software architecture, and development in Java and Microsoft Visual Studio .NET for Web services.

2 Lab design

The lab session is divided into three parts. The expected duration of the lab session is one hour.

1. Part 1 includes Sections 4 and 5. In this part, the participants will use *Apache Axis toolkit v. 1.2 Alpha* and *Apache Jakarta Tomcat v. 5.5.9* to develop a simple Java Web service for squaring integers. A standalone Java program will then be developed as the client of

the Java Web service to demonstrate the interaction between client and server sides of the Web service.

2. Part 2 includes Sections 6 and 7. In this part, the participants will use *Microsoft Visual Studio .NET 2008* and *Microsoft IIS* Web server to develop a simple .NET Web service in C# for serving a personalized *Hello World* message to its clients. A C# .NET Web application will be developed to act as a client for the *Hello World* .NET Web service.

3. Part 3 includes Section 8. This part is used for demonstrating the interoperability of Java and .NET Web services. A simple Java program will be developed as a client for the *Hello World* .NET Web service that we developed in Part 2.

3 Lab environment

This lab depends on the following software tools that are freely available to *Pace University* students:

1. VMware virtual machine (VM) *Ubuntu10* (http://csis.pace.edu/lixin/ubuntu/ubuntu10.exe) or *WinXP2* (http://csis.pace.edu/lixin/download/vm.html). If you plan to work on lab components on .NET, you have to use WinXP2. You can watch my video tutorial http://csis.pace.edu/lixin/vm/ to get started in using the WinXP2 VM, and read Sections 2 and 3 of my Linux Tutorial http://csis.pace.edu/lixin/ubuntu/LinuxTutorial.pdf to learn how to install and use the Ubuntu10 VM.

2. If you use *Ubuntu10*, you first need to run "sudo chmod -R a+rwx ~/tomcat/webapps/axis" in a terminal window so you can modify files in folder "~/tomcat/webapps/axis". You can start a terminal window with menu item "Applications|Accessories|Terminal".

3. If you use *Ubuntu10*, you also need to download http://csis.pace.edu/lixin/download/webServiceLab.7z to *Ubuntu10* home folder "/home/user", right-click on it in file browser and choose "Extract Here" to extract folder "/home/user/webServiceLab". Later in this document, whenever we refer to folder "C:\webServiceLab" for Windows, you should use "~/webServiceLab" on the *Ubuntu10* VM.

4. If you use VM WinXP2, the lab folder "C:\webServiceLab" has already been created for you.

The following sections assume you are using one of my pre-configured VMware VM WinXP2. The VM for your course may be slightly different so you need to make necessary adjustments in following this lab manual.

Corresponding Terms on Windows and Linux

This document assumes that you work on a Windows VM. If you are using a Linux VM like Ubuntu10, you should adjust the terms as we list below:

Windows	Linux
C:\webServiceLab	~/webServiceLab
Windows Explorer	File browser
Internet Explorer	Firefox
Command Prompt window (DOS window)	Terminal window (Applications|Accessories|Terminal)
C:\tomcat6	~/tomcat
http://localhost:8090	http://localhost:8080
EditPad	gedit
..\..\..\..	../../../..

4 Development of a Java Web service for squaring an integer

In this section, you are going to create a new Java Web service that receives one integer and returns its squared value. For example, if input is 2, it will return 4.

The implementation is through a Java class source file. For your convenience, the file has been created in "C:\webServiceLab\java\server\SquareIntegerServer.java" ("~/webServiceLab/java/server/SquareIntegerServer.java" on Linux). Its contents are listed below:

```
public  class  SquareIntegerServer {
   public int square(int x) throws Exception {
      int  result = x*x;
      return  result;
   }
}
```

Program 1 SquareIntegerServer.java

Class "SquareIntegerServer" has only one method named "square", which accepts one integer-typed parameter "x", and returns an integer. The business logic for this method is to evaluate the square of the value in variable "x". The result is then returned to the method invoker. The second line of the source code contains clause "throws Exception" because we intend to use this class to implement a Web service, and there are many reasons for the network interactions between the method implementation and its remote client to go wrong. We need to explicitly declare that networking errors may happen.

After reviewing the source code, we are ready to deploy it as a Web service on our Tomcat Web server.

1. Open a *Command Prompt* window by clicking on "Start|All Programs|Accessories|Command Prompt" (or open a terminal window on Linux).

2. Change directory to "C:\webServiceLab\java\server" (using "cd ~/webServiceLab/server" on Linux). You can do so by typing the following in the *Command Prompt* window ("[Enter] means typing the Enter key):

   - C: [Enter]
   - cd C:\webServiceLab\java\server [Enter]

3. Verify that the source code is a valid Java class by typing the following command in the *Command Prompt* window to compile the source file:

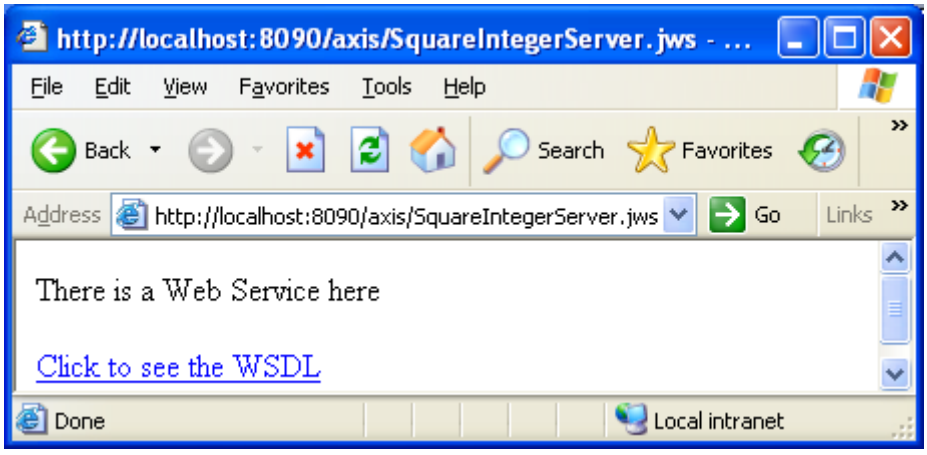
   - javac SquareIntegerServer.java [Enter]

   In this case you will notice that a new file "SquareIntegerServer.class" is successfully created. In general, if you see error messages during compilation, you need to revise your source code to make it error-free.

4. In *Windows Explorer* ("file browser" on Linux), rename file "SquareIntegerServer.java" to "**SquareIntegerServer.jws**". Here "jws" stands for "Java Web service".

5. In *Windows Explorer* (*File browser* on Linux; make sure you have run "sudo chmod -R a+rwx ~/tomcat/webapps/axis" beforehand), copy file "SquareIntegerServer.jws" to "**C:\tomcat6\webapps\axis**" ("~/tomcat/webapps/axis" in Linux), which is the directory for you to deploy any of your Java Web services.
6. Since we just deployed a new Web component, it is safer to restart Tomcat if it is already running. If Tomcat is running, double click on the "Shut down Tomcat" shortcut on desktop to shut it down. Double click on the "Start Tomcat" shortcut on the desktop to launch Tomcat. (On Linux, run in a terminal window "tomcat-stop" followed by "tomcat-start") Wait until the Tomcat startup is complete. You can minimize the DOS window, but don't close it.

7. Open a Web browser and visit URL http://localhost:8090/axis/SquareIntegerServer.jws (on Linux replace 8090 with 8080). You are going to see a window like the following one:



8. Your new Web service is working now! Click on the hyperlink "Click to see the WSDL" to review the WSDL file for this Web service. You will notice that the URL address box now contains value http://localhost:8090/axis/SquareIntegerServer.jws?wsdl. As a matter of fact, given any Web service URL, you can always add "?wsdl" to the end of the URL for retrieving the WSDL file for the Web service. When you started the Tomcat, the *AXIS* toolkit embedded in the Tomcat compiled our source file "SquareIntegerServer.jws" into a Java servelet with URL http://localhost:8090/axis/SquareIntegerServer.jws, and also generated the *Web service definition language (WSDL)* file for describing the method exposed by this Web service. Figure 1 shows the contents of the WSDL file after slight formatting. The lines in bold face specify the format of SOAP response message (for method call return value), SOAP request message (for method invocation information), and the URL for accepting the SOAP requests.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <wsdl:definitions
  targetNamespace="http://localhost:8090/axis/SquareIntegerServer.jws"
  xmlns:apachesoap="http://xml.apache.org/xml-soap"
  xmlns:impl="http://localhost:8090/axis/SquareIntegerServer.jws"
  xmlns:intf="http://localhost:8090/axis/SquareIntegerServer.jws"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 >

 <wsdl:message name="squareResponse">
  <wsdl:part name="squareReturn" type="xsd:int" />
 </wsdl:message>

 <wsdl:message name="squareRequest">
  <wsdl:part name="x" type="xsd:int" />
 </wsdl:message>

 <wsdl:portType name="SquareIntegerServer">
  <wsdl:operation name="square" parameterOrder="x">
   <wsdl:input message="impl:squareRequest" name="squareRequest" />
   <wsdl:output message="impl:squareResponse" name="squareResponse" />
  </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="SquareIntegerServerSoapBinding"
                type="impl:SquareIntegerServer">
  <wsdlsoap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="square">
   <wsdlsoap:operation soapAction="" />
   <wsdl:input name="squareRequest">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://DefaultNamespace" use="encoded" />
   </wsdl:input>
   <wsdl:output name="squareResponse">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://localhost:8080/axis/SquareIntegerServer.jws"
                use="encoded" />
   </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="SquareIntegerServerService">
  <wsdl:port  binding="impl:SquareIntegerServerSoapBinding"
              name="SquareIntegerServer">
     <wsdlsoap:address location="http://localhost:8090/axis/SquareIntegerServer.jws" />
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Figure 1 WSDL file for the SquareIntegerServer Web service

5 Development of a Java client application for consuming the Java Web service

Now it is time for us to develop a Java program to function as the client of the new Web service.

1. In the *Command Prompt* window, change directory to "C:\webServiceLab\java\client" by typing

   cd C:\webServiceLab\java\client [Enter]

   (On Linux, type "cd ~/webServiceLab/java/client")

2. Make sure that your Tomcat is still running. Run the following command in the *Command Prompt* window to generate source files for your proxy class for your new Web service:

   java org.apache.axis.wsdl.WSDL2Java http://localhost:8090/axis/SquareIntegerServer.jws?wsdl

   (on Linux, replace 8090 with 8080). This command will generate four Java source files supporting the Web service proxy class. These four Java files all belong to Java package "localhost.axis.SquareIntegerServer_jws", therefore they are located in directory, relative to the current directory "C:\webServiceLab\java\client", "localhost\axis\SquareIntegerServer_jws". The name and function of these four Java files are as follows:

   a) **SquareIntegerServer.java**
   Interface for the Web service proxy class to implement

   b) **SquareIntegerServerService.java**
   Interface for the factory class of the Web service proxy objects (proxy objects are not generated by operator **new**, but through method calls to a factory object)

   c) **SquareIntegerServerSoapBindingStub.java**
   Proxy class source, which implements interface **SquareIntegerServer**

   d) **SquareIntegerServerServiceLocator.java**
   Factory class of proxy objects; it implements interface **SquareIntegerServerService**

   The names of these files and the subdirectories depend on the URL and contents of the WSDL file. When you create proxy classes for a different Web service, you need to change the contents of the batch file "makeProxy.bat" so it refers to the URL of the WSDL file of the new Web service. The resulting proxy class files may have different names and packages, but they should follow the same pattern as our example here.

3. Now in directory "C:\webServiceLab\java\client", use a text editor, like *EditPad*, to create a Java source file named "SquareIntegerClient.java" with its contents specified

below (for your convenience, the file has been created for you; please review its contents):

```
import localhost.axis.SquareIntegerServer_jws.*;

public class SquareIntegerClient {
  public static void main(String[] args) throws Exception {
    int value = 0; // value to be squared
    // The program expects to receive an integer on command-line
    // Program quits if there is no such integer
    if (args.length == 1)  // there is one command-line argument
      value = Integer.parseInt(args[0]);  // parse the string form of integer to an int
    else {
      System.out.println("Usage: java SquareIntegerClient  [integer]");
      System.exit(-1);  // terminate the program
    }
    // Get the proxy factory
    SquareIntegerServerServiceLocator factory =
        new SquareIntegerServerServiceLocator();
    // Generate the Web service proxy object
    SquareIntegerServer proxy = factory.getSquareIntegerServer();
    // Access the Web service
    int result = proxy.square(value); // invoke server method to square value
    System.out.println("Square of " + value + " is " + result);
  }
}
```

Program 2 SquareIntegerClient.java

The file first imports all the Java classes generated by "makeProxy.bat" for supporting the Web service proxy class. Then it checks whether there is a command-line argument. If there is no command-line argument, the program prints a usage message and then terminates. Otherwise it parses the string form of the command-line integer into an *int* number, and saves the number in variable *value*. A factory object for generating a proxy object is then created with "new SquareIntegerServerServiceLocator()", and a new proxy object is obtained by calling the factory's method "getSquareIntegerServer()". The actual Web service invocation happens on the line with "proxy.square(value)". Even though this method call is to the local proxy object, the proxy's body for method "square" makes a TCP IP connection to the Tomcat servlet specified in the WSDL file, issues an HTTP request carrying a SOAP request message as its *entity body*, receives a response SOAP message from the Tomcat servlet (from the *entity body* of the HTTP response), parses the SOAP response message into a Java *int* value, and returns the value as its own method return value.

4. Now it's time to compile the client program. In the *Command Prompt* window, type

C:\webServiceLab\java\client> **javac SquareIntegerClient.java -source 1.4**

You only need to type the bold face part of the above line. The *javac* command-line switch "-source 1.4" is for informing the Java compiler that this class should be compiled with Java SDK 1.4's API. This is because in Java SDK 1.5 (or 5), "enum" is defined as a Java key word, but they are not in Java SDK 1.4 or earlier. The *AXIS* toolkit uses "enum" extensively as a package name, which is not allowed in Java SDK 1.5 (5). You may see two warnings related to this fact, which are harmless. After executing this command, you will notice a new file "SquareIntegerClient.class".

5. Now you can test your Web service with your new Java client program. Type the bold face part of the following line, and the second line is the response of your client program. The computation is performed by your Web service. You can try to use different integers to replace integer 2.

   C:\webServiceLab\java\client> **java SquareIntegerClient 2**
   Square of 2 is 4

Congratulations! You have successfully implemented a Web service client application in Java. The class name of your client application is not important and you can choose any name meaningful to you.

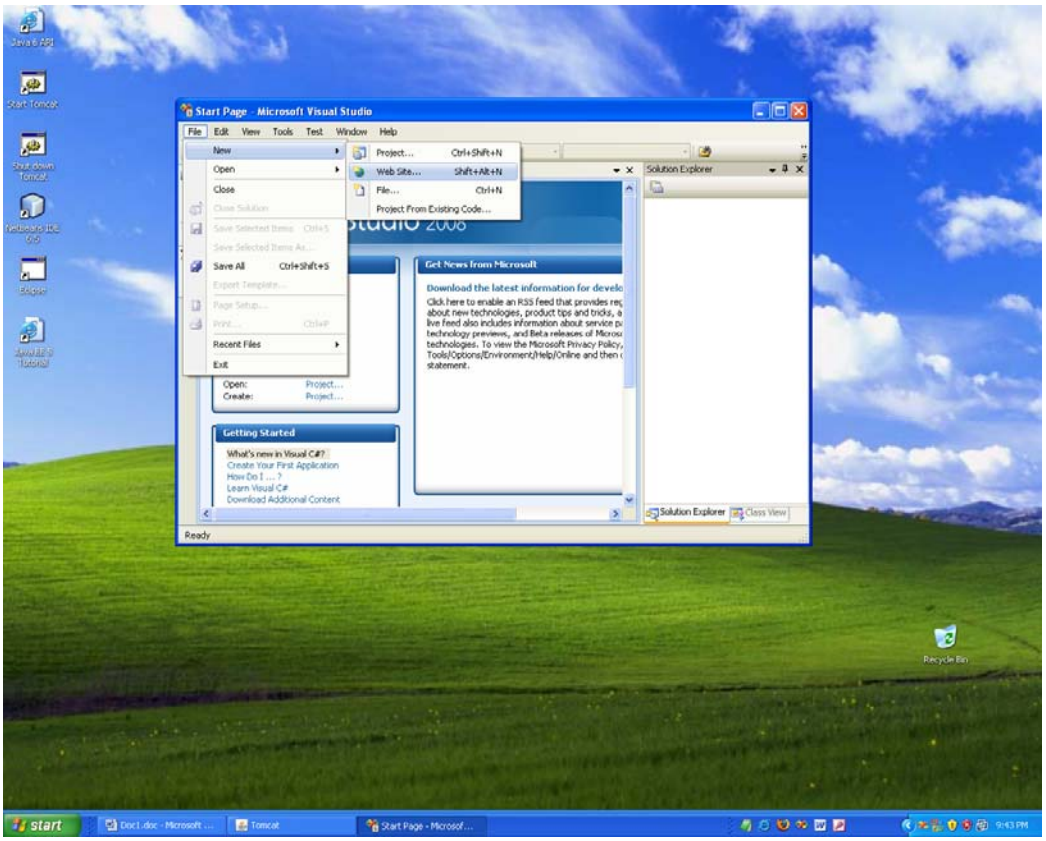
To consume a Web service implemented on another platform, say on Microsoft .NET with C# or Visual Basic .NET, you just follow the same procedure. As long as you provide the right URL for the WSDL file of the Web service, the implementation language or platform of the Web service is of no relevance to you. This is the real power of Web services!

6 Development of a *Hello* Web service with C# in Microsoft Visual Studio

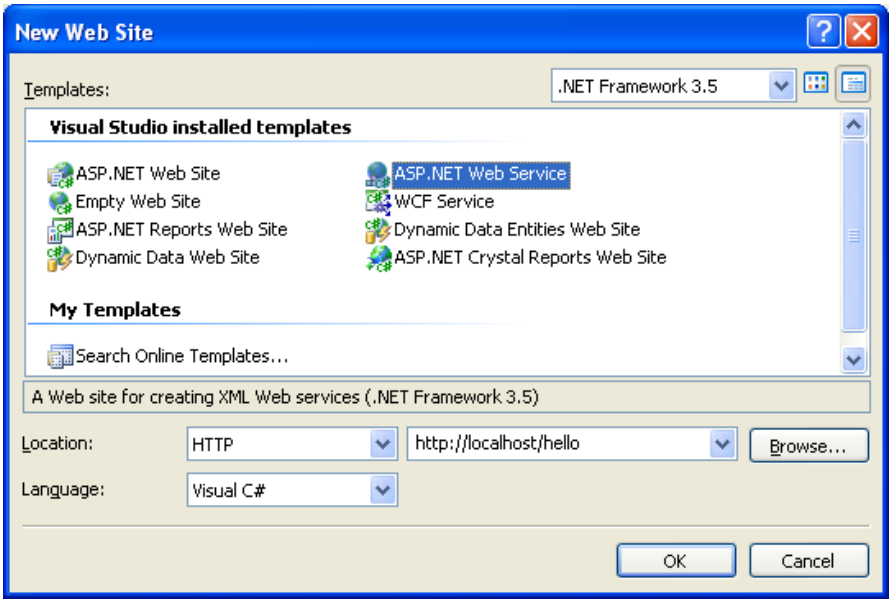
(You need VM WinXP2 to work on this section.) Now we are going to use *Microsoft Visual Studio 2008* to create a new Web service. While the underlying process is very similar to that with Java, *Visual Studio* hides away much more implementation details from the developers.

1. First, make sure your IIS Web server is running. You can open a Web browser and direct it to http://localhost/. If the IIS Welcome page is not displayed, you need to start your IIS server by following my instructions in the document "Software Installation for .NET Web Services".

2. Click on "Start|All Programs|Microsoft Visual Studio 2008| Microsoft Visual Studio 2008" to launch the *Microsoft Visual Studio 2008* IDE.
3. Click on "File|New|Web Site ...".

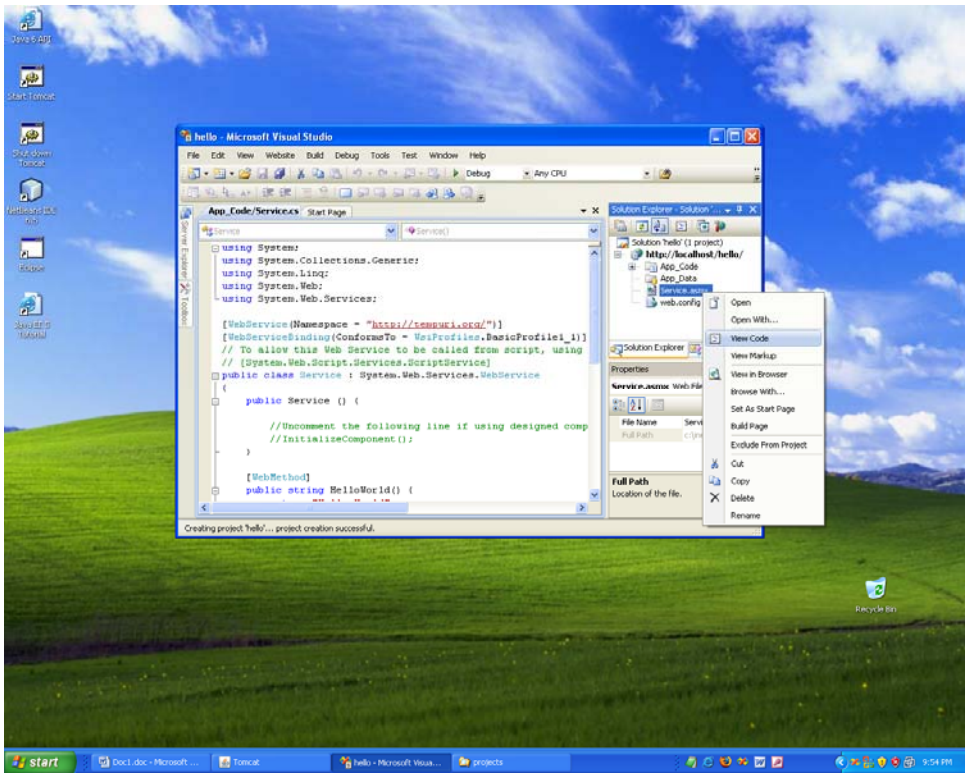


4. The following "New Web Site" window will pop up. Click on the "ASP .NET Web Service" template. In the bottom "Location" combo box, choose "HTTP". In its right URL textbox, enter URL *http://localhost/hello*, as shown below:



Then click on the right-bottom *OK* button. Now *Visual Studio* should show you the contents of file "Service.cs" in the left editor pane, as below. If it does not, right-click on

file “Service.asmx” in the right *Solution Explorer*, and then click on “*View Code*” on the pop-up menu, as shown below:



5. Notice that in C#, “using System” corresponding to Java’s “import System.*”. Here we have a class named “Service”. If necessary, you can rename the component “Service.asmx” by right-clicking on it in the *Solution Explorer* and then clicking on *Rename* in the pop-up menu (we choose not to do so here).

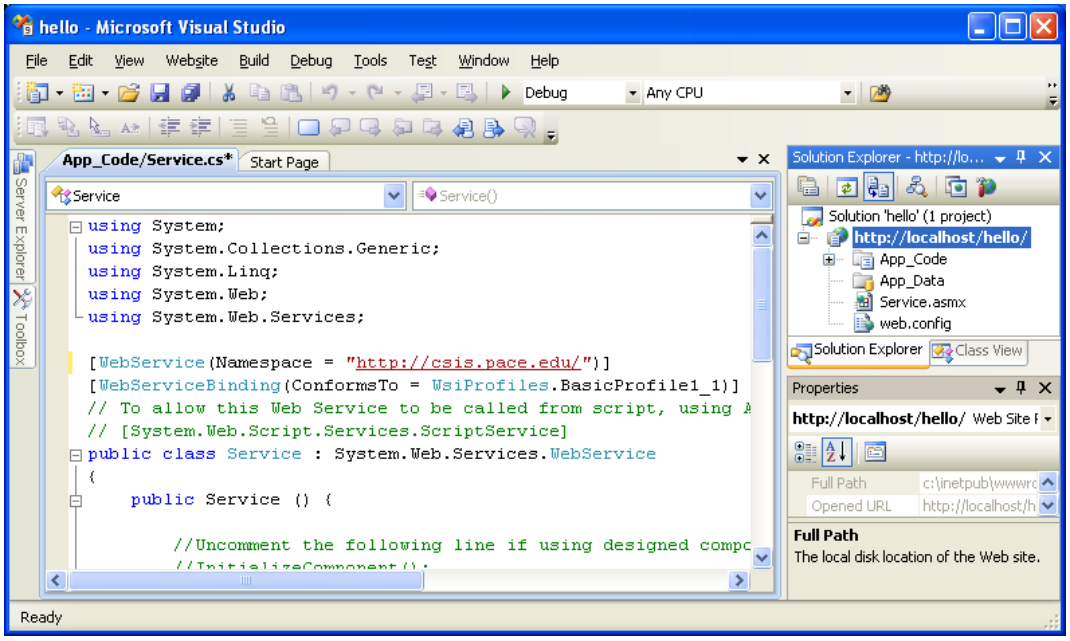
6. Now you change namespace value before the declaration of class “Service”. In the *Editor* pane, replace the line

   [WebService(Namespace="http://tempuri.org/")]

   with

   [WebService(Namespace="http://csis.pace.edu/")]

   so *Visual Studio* now looks like



For your convenience, you can also copy-and-paste the text line before the dashed line in file "C:\webServiceLab\dotNet\HelloWorldServer.txt". This new line is a *class attribute* declaring a namespace for the WSDL file to be created for our new Web service. If we don't declare a namespace with a class attribute, or use the default one, the project still works, but we will receive a warning. The value of the "Namespace" attribute could be any string, hopefully uniquely identifying the authors. Its purpose is for avoiding name collision for Web services developed by different companies.

7. In the *Editor* pane, scroll down to the end of file "Service.cs". Replace the sample code below

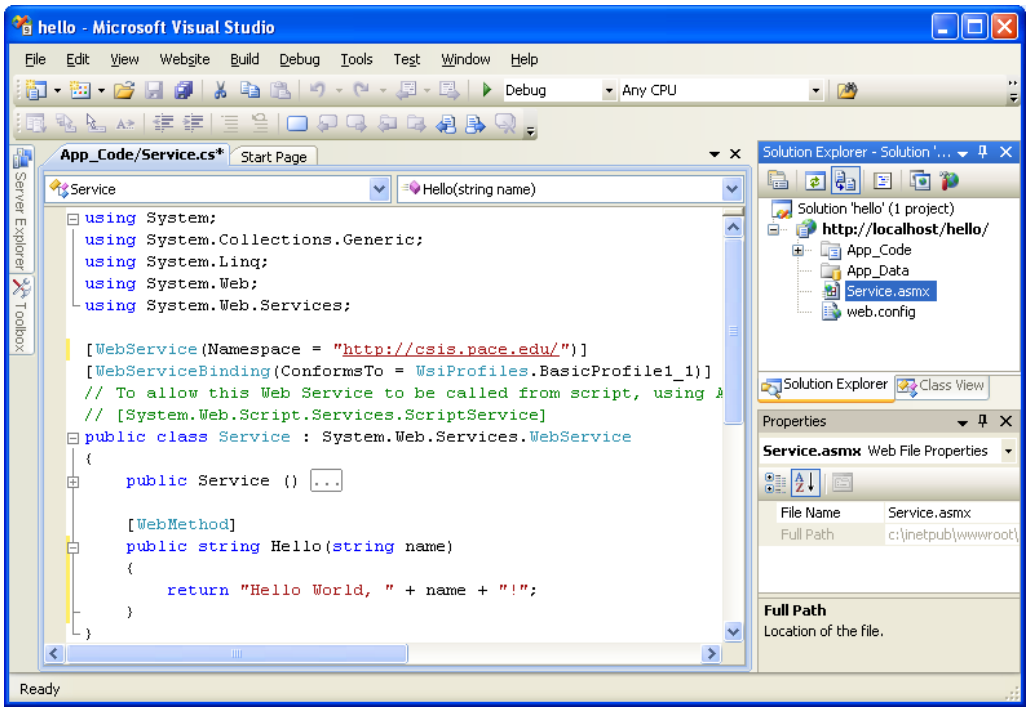
```
[WebMethod]
public string HelloWorld()
{
    return "Hello World";
}
```

with the following new code

```
[WebMethod]
public string Hello(string name)
{
    return "Hello World, " + name + "!";
}
```

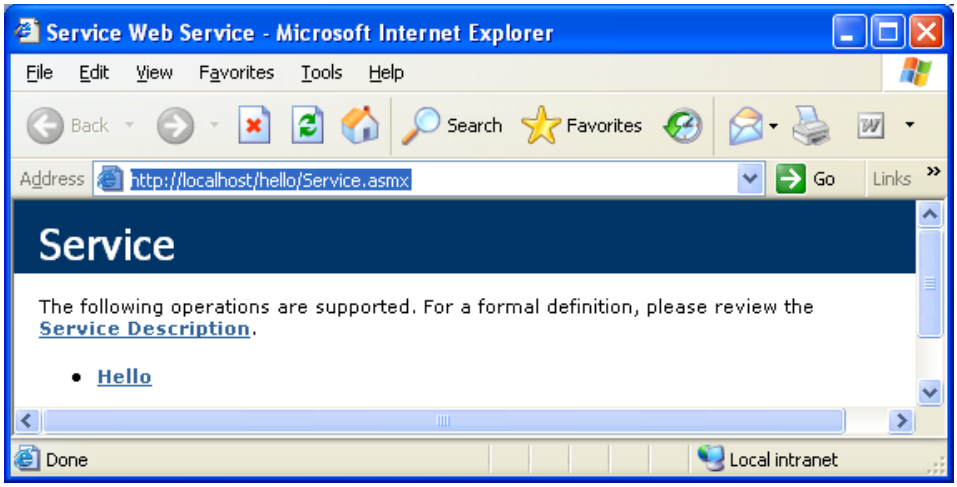
For your convenience, you can also copy-and-paste the text after the dashed line in file "C:\webServiceLab\dotNet\HelloWorldServer.txt". "[WebMethod]" is a *method attribute* declaring that the following method is intended to be exposed as a Web service. In *Visual Studio*, we need to add attribute "[WebMethod]" before each public method that we plan to expose as a Web service. The business logic here is very simple. Input is a name string.

The return value is an extended string containing the value of parameter *name*. Now *Visual Studio* looks like

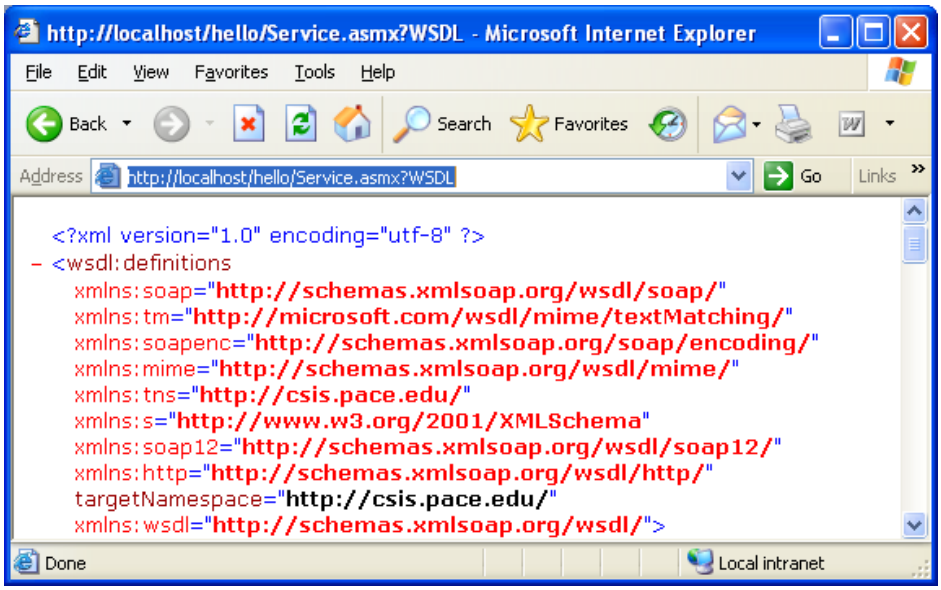


8. Click on “File|Save” to save the file.

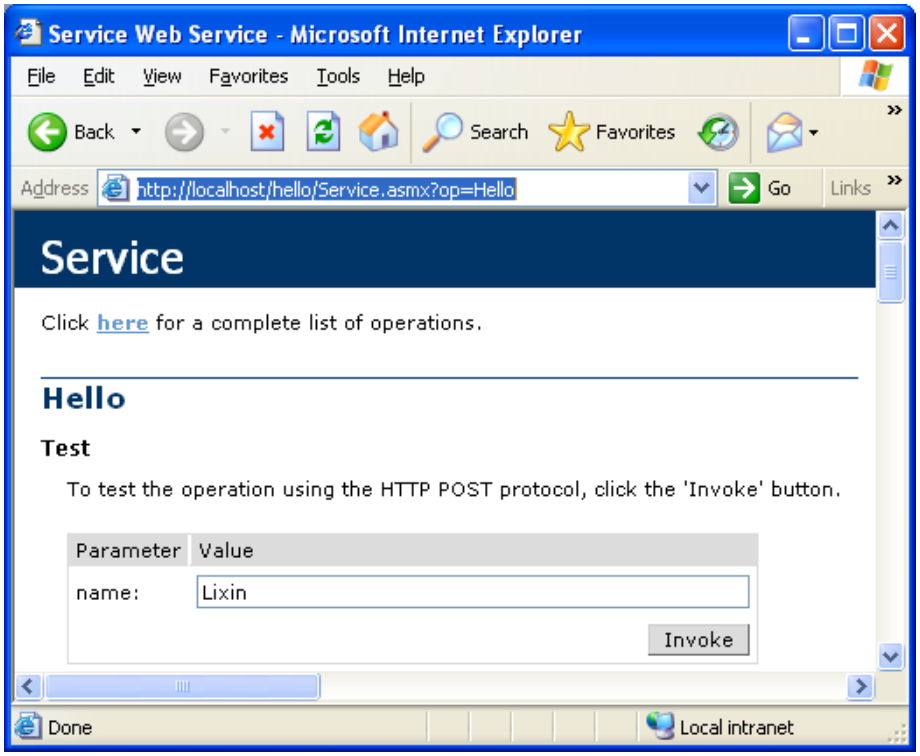
9. Click on menu item “Debug|Start Without Debugging” or type key combination Ctrl+F5 to compile and run the project. The new C# project is now compiled into a Web service, its WSDL file is generated, a new ASP component is generated to function as the entry point for this Web service, and the ASP component and the WSDL file are both deployed to your IIS Web server. A new *Internet Explorer* window, like the one below, will pop up automatically to allow us to test the new Web service. Notice that the URL for the Web service page is *http://localhost/hello/Service.asmx*.



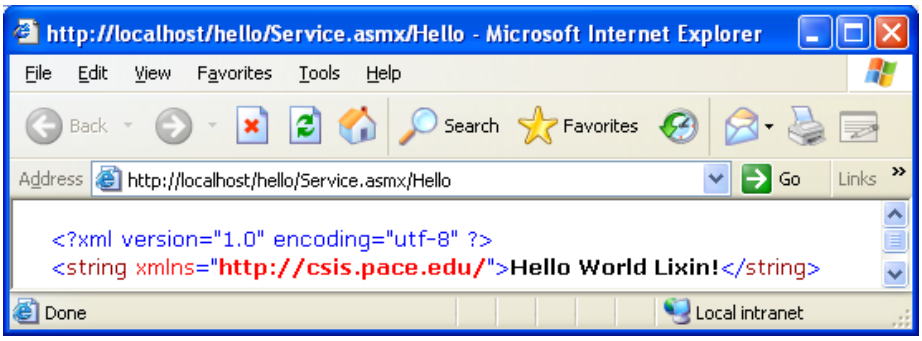
10. Click on the hyperlink “Service Description”, and the WSDL file for the Web service will be displayed in the Web browser, as shown below. Notice that the URL for the WSDL file is *http://localhost/hello/Service.asmx?WSDL*.



11. Click on the *Back* arrow of *Internet Explorer* and go back to the previous page. Click on the hyperlink “Hello”. The following window will be presented. This page allows you to test your new Web service.



12. Type your name in the *name* textbox. Click on the *Invoke* button. A window similar to the following one will be presented to you.



So far you have successfully developed and deployed a new .NET Web service.

13. Shut down this *Internet Explorer* so the run session terminates. Before we start a new Web client project for our Web service, click on "File|Close Solution" to close the current project.

7 Development of a .NET Web client application to consume the .NET Web service

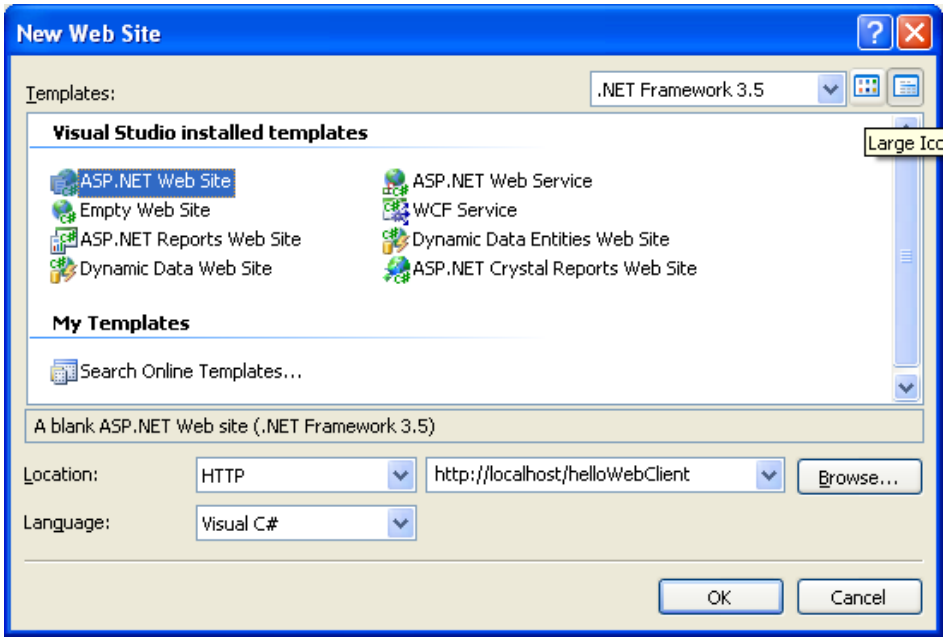
Now you will develop a C# Web application to function as the client of your new .NET Web service.

1. First, make sure your IIS Web server is running. You can open a Web browser and direct it to http://localhost/. If the IIS Welcome page is not displayed, you need to start your IIS server by following my instructions in the document "Software Installation for .NET Web Services".

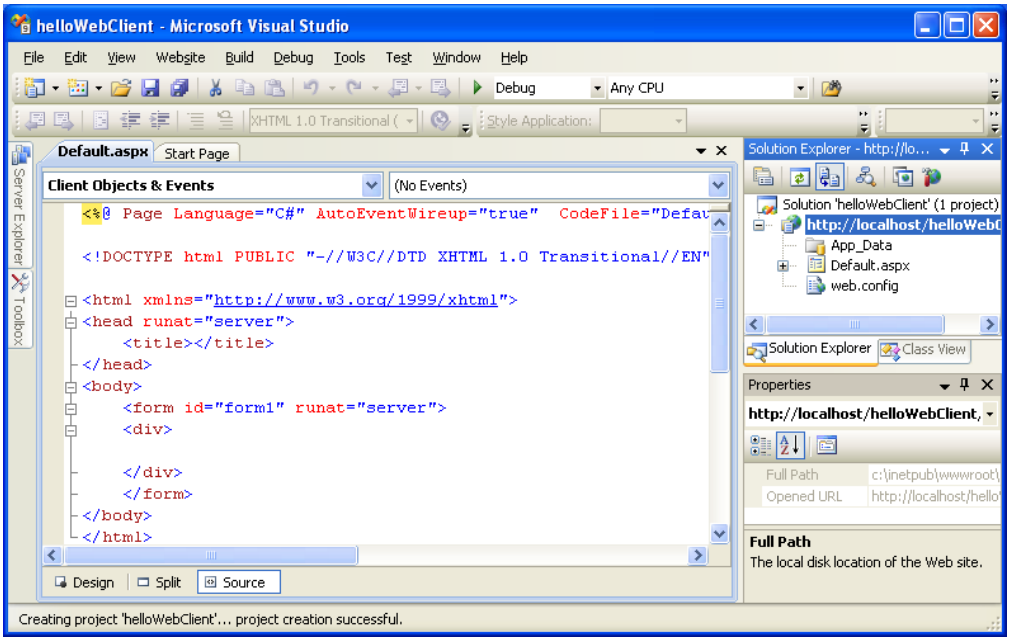
2. If Microsoft Visual Studio is not running, click on "Start|All Programs|Microsoft Visual Studio 2008| Microsoft Visual Studio 2008" to launch *Microsoft Visual Studio 2008* IDE.

3. Click on "File|New|Web Site ..." to launch the "New Web Site" window.

4. In the pop-up "New Web Site" window, click on the icon for "ASP .NET Web Site" to highlight it. In the "Location" combo box, choose "HTTP"; in the corresponding textbox, enter value "http://localhost/helloWebClient". The resulting "New Web Site" window will now look as below:

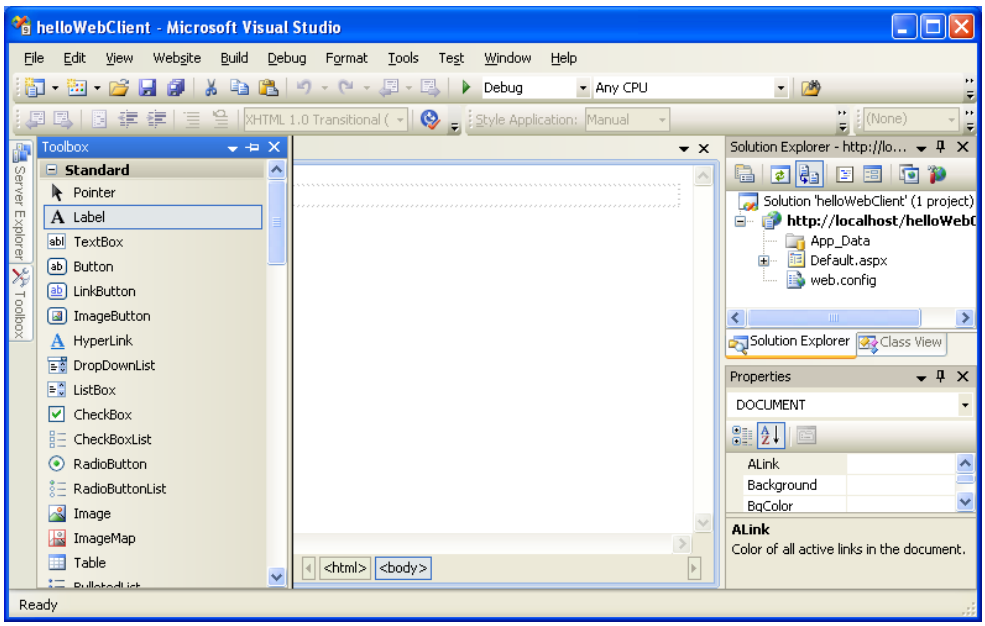


5. Click on the *OK* button to create the project and show the contents of file "Default.aspx" in the editor pane, as shown below. "Default.aspx" is an HTML file, and you can view it in either *Design* mode or in *Source* (HTML) mode, by clicking on the corresponding button below the *Editor/Design* pane. To view the C# code behind file "Default.aspx", you can right-click the file in the *Solution Explorer* (the right-upper corner pane) and then click on "View Code" in the popup menu. But we don't need to open this C# file yet. These two files are very similar to a JSP page and the JSP page's supporting Java classes.



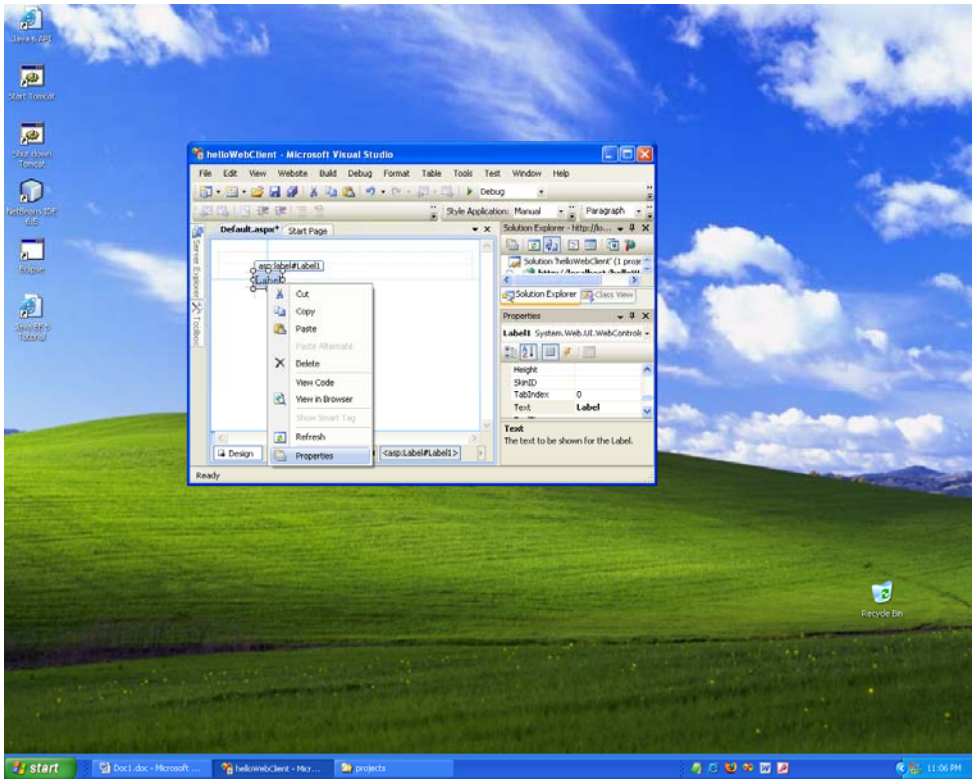
6. Click on the "Design" mode button at left-bottom of the editor pane to see file "Default.aspx' in design mode.

7. We now create a simple graphic user interface for the Web application. Putting the mouse cursor on the *Toolbox* button (or clicking on it) on the left margin of the editor pane, you will see the pop-up Toolbox menu as below:

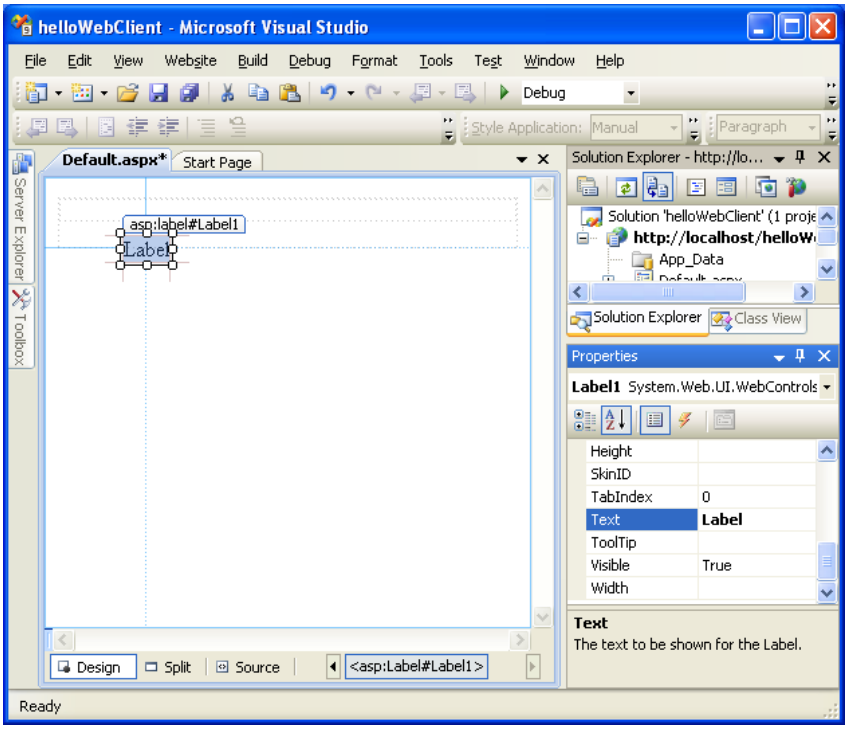


The *Toolbox* menu has categories or sub-menus including *Standard*, *Data*, *Validation*, *Login*, *WebParts*, and *HTML*; and each of the categories contains many controls as sub-menu items. We will only use the controls under category *Standard*. To add a *control* to your current Web form, just double-click on its corresponding sub-menu item in the proper *Toolkit* sub-menu, and a new instance of the control will be added to your current form. You can then visually drag its center or header to move it to your desired position, and drag one of its boundary square handles to reshape it. To customize a control's properties, you first click on the control in the *Design* pane to highlight it, and then change its properties through the *Properties* editor pane below the *Solution Explorer* pane of the *Visual Studio* IDE. If you cannot find the *Properties* pane, right-click on the current control and choose "Properties" in the pop-up menu. The most important properties are *Text* for what value will be displayed on the control's surface and *ID* for the program variable name representing this control. After you enter a new value for a property, make sure you follow the value with the *Enter* key.

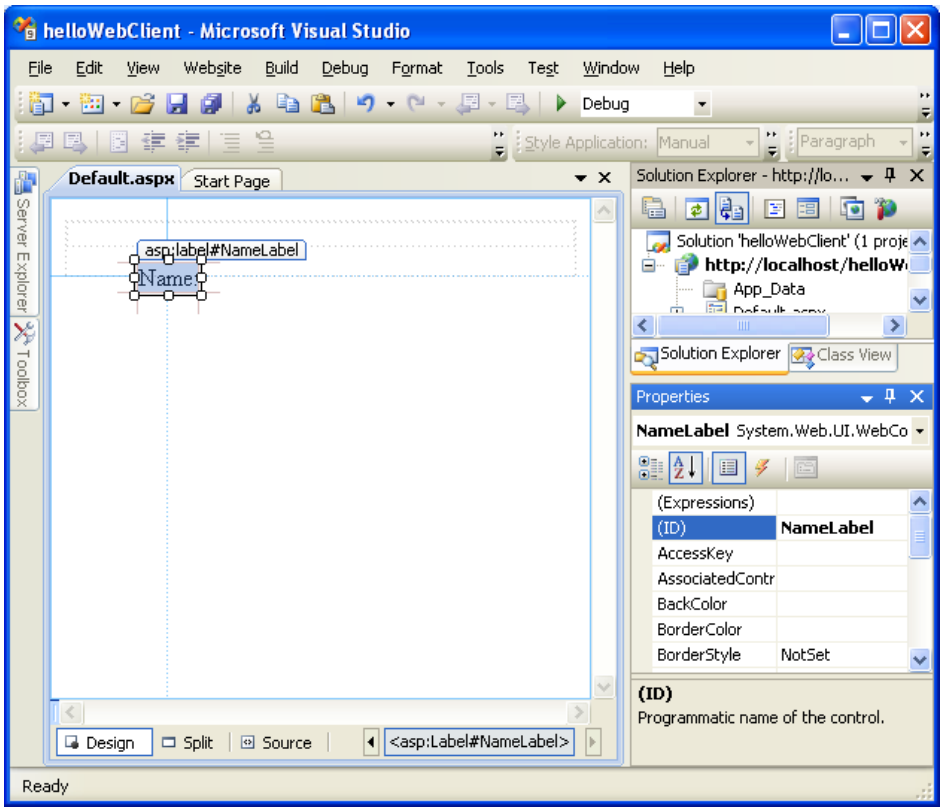
8. Double click on "Label" under *Standard* to add a label in the design pane. Use mouse to drag it to where you want. To see its properties, right-click on the label and choose "Properties" on the pop-up menu, as below:



Now you can see the properties of the current label control in the *Properties* editor under the *Solution Explorer* pane, as below.



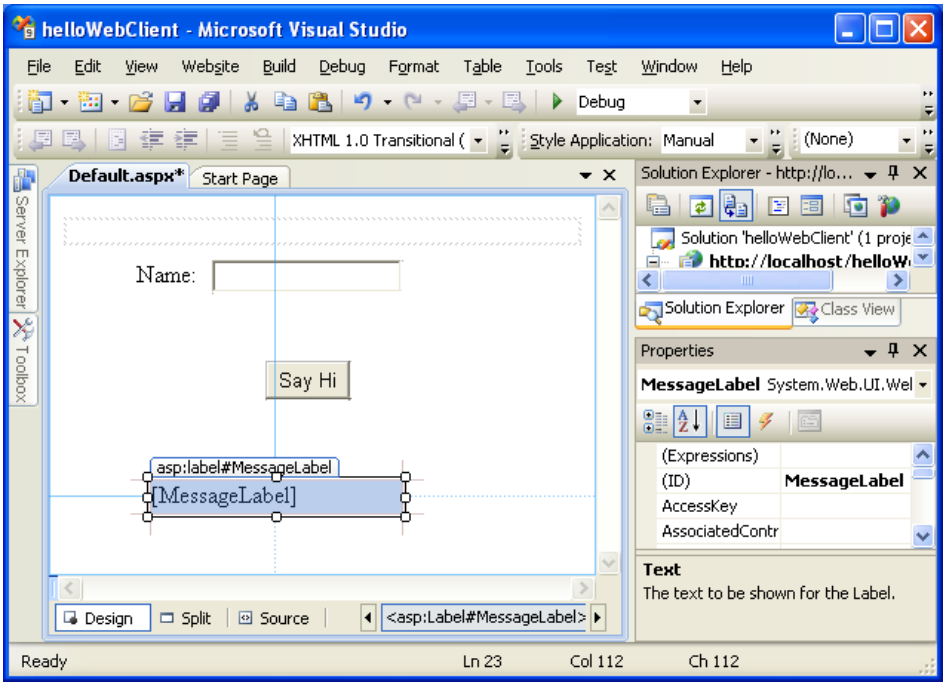
9. The right-bottom *Properties* pane is used to edit the properties of the current active control. For this label, we change its *ID* value to “NameLabel”, and its *Text* value to “Name:” as below:NameLabel



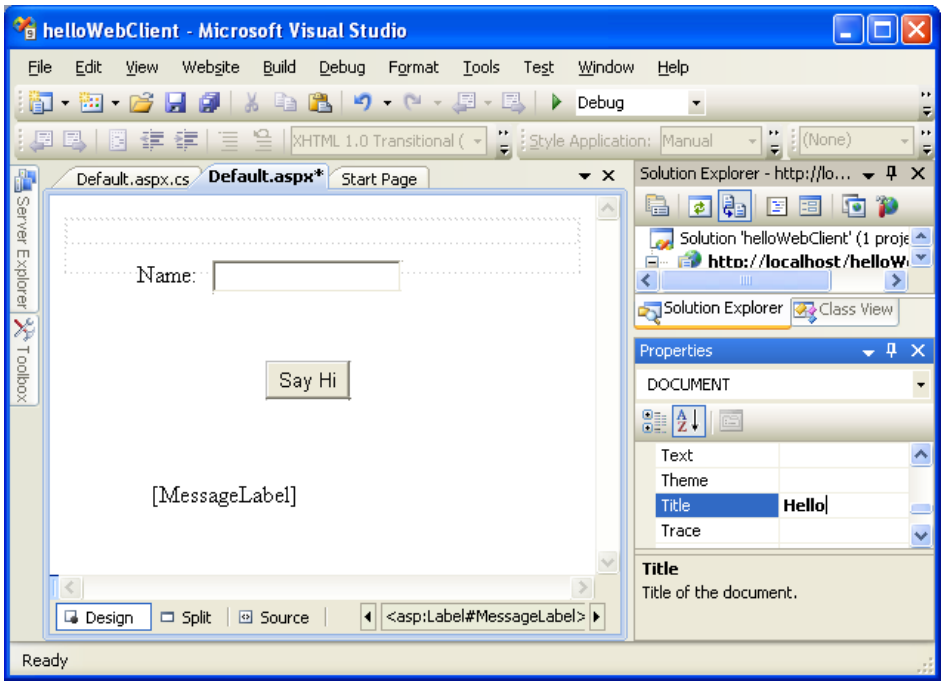
10. Repeat the above steps to add to the Web form "Default.aspx" one *TextBox* for entering a name, one *Button* for submitting the name in the TextBox, and one more *Label* for displaying the server messages. You customize their *ID* and *Text* values in the *Properties* editor as specified in the following table:

Component	ID	Text
Label 1	NameLabel	Name:
TextBox	NameTextBox	
Button	SayButton	Say Hi
Label 2	MessageLabel	

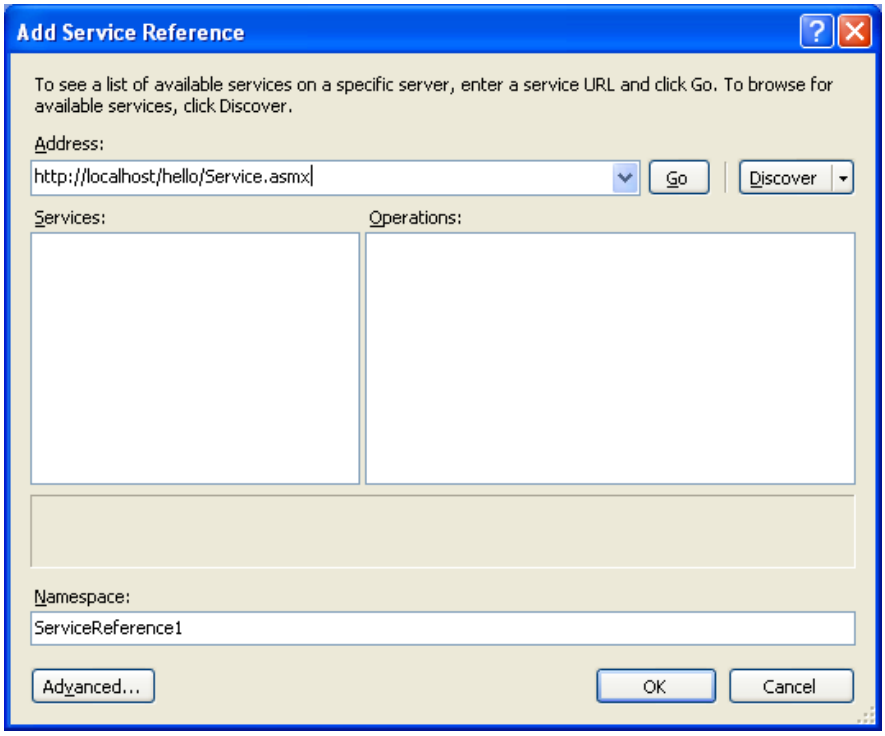
Notice that the *TextBox* and the bottom *Label* have blank value for their *Text* property. The *Visual Studio* IDE will now look like the following window:



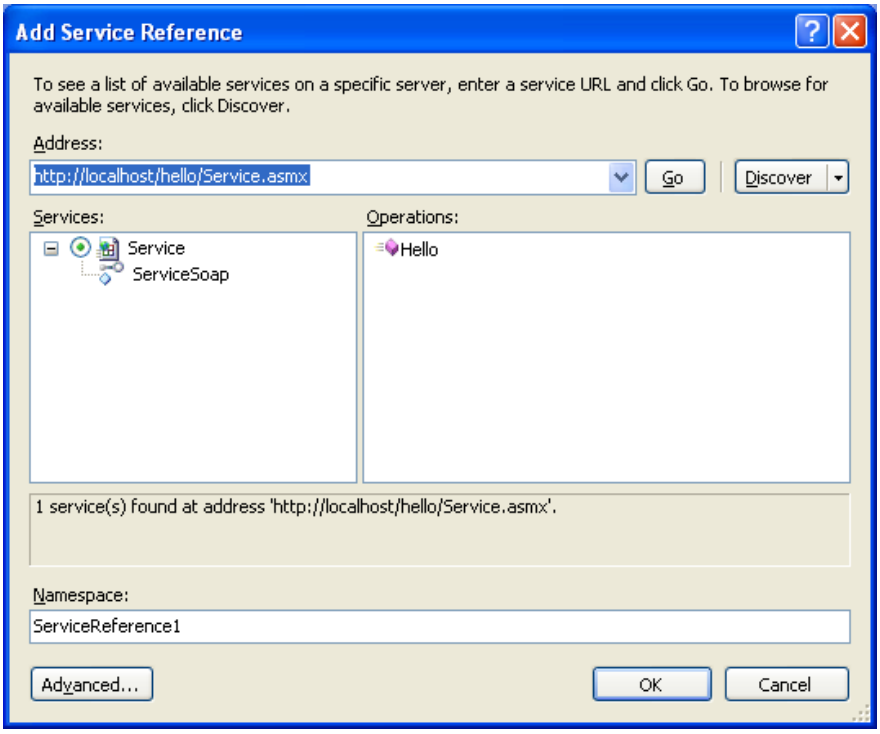
11. Click anywhere in the *Design* pane where there are no controls. Now you can use the *Properties* pane to set the Web page container's property. Here we change its *Title* value to "Hello".



12. Now you need to download the *WSDL* file for your *Hello* Web service, and generate the proxy class for it. Bouble-click on "Website|Add Service Reference…" to show the "Add Service Reference" window. In the URL textbox of this pop-up window, enter our *Hello* Web service's URL: *http://localhost/hello/Service.asmx*, as below:

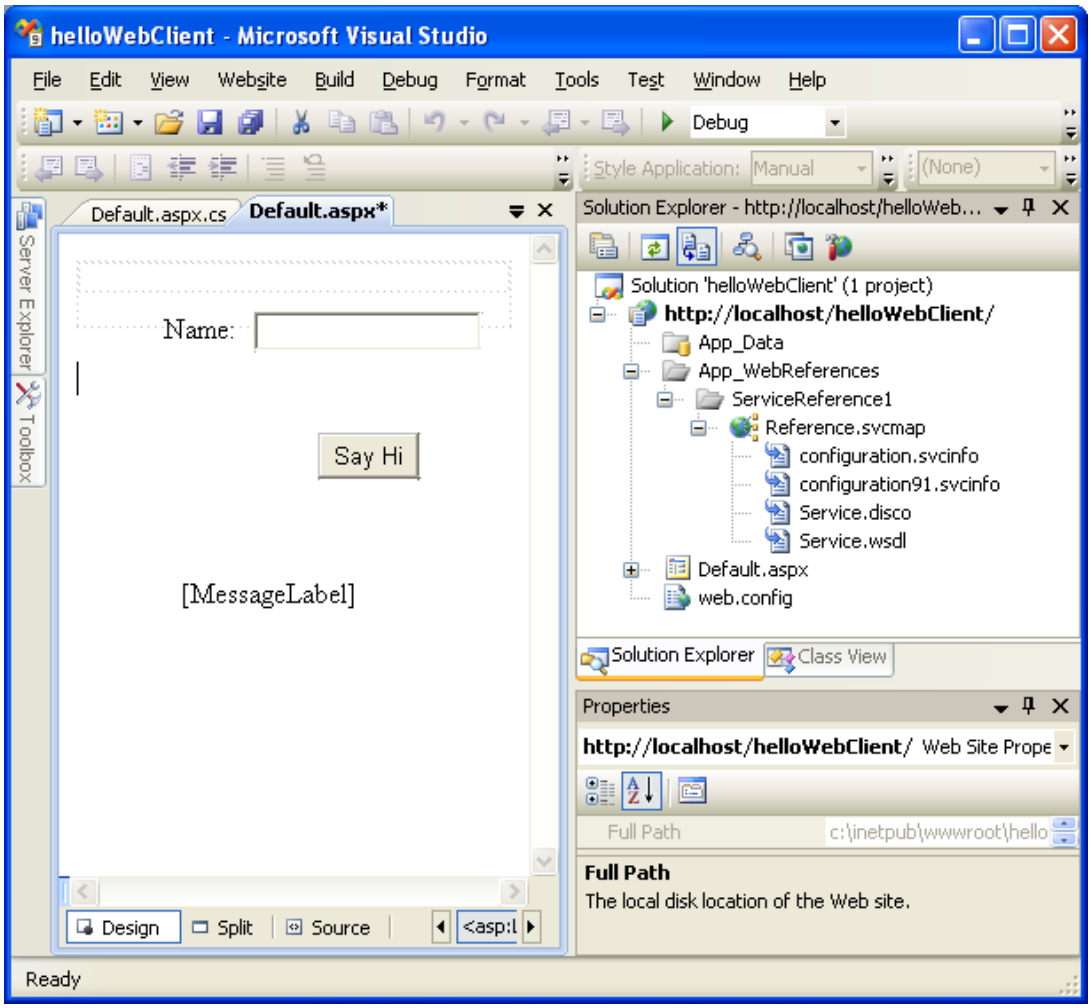


Click on the *Go* button. After a short delay you will see that the *Hello* Web service is found, as shown below:

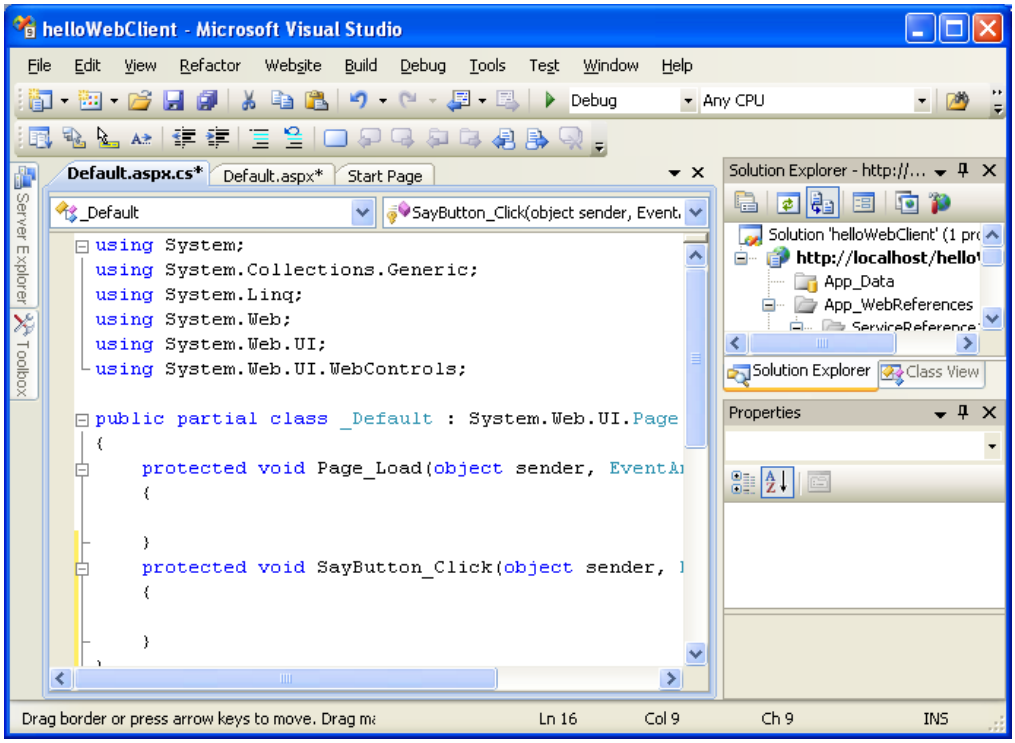


It shows that we have found a Web service named “Service”, and this service supports an operation named “Hello”. By default the namespace for the Web service reference is “ServiceReference1”. The Web service namespace will also be used as the namespace for

the corresponding proxy class. Now you click on the "OK" button to add the Web service reference to the project and generate the proxy class in the project. Now you can see in *Solution Explorer* the new Web service reference "ServiceReference1".



13. Now we are ready to implement the business logic of this client application. While file "Default.aspx" is open in *Design* mode, double-click on the "Say Hi" button and your mouse cursor will be put in the body of the event handler of this button, as shown below:



14. Now insert code in the body of method "SayButton_Click()" so the method reads as the following listing (you can also copy-and-paste the text after the dashed line in file "C:\webServiceLab\dotNet\HelloWorldClient.txt"):

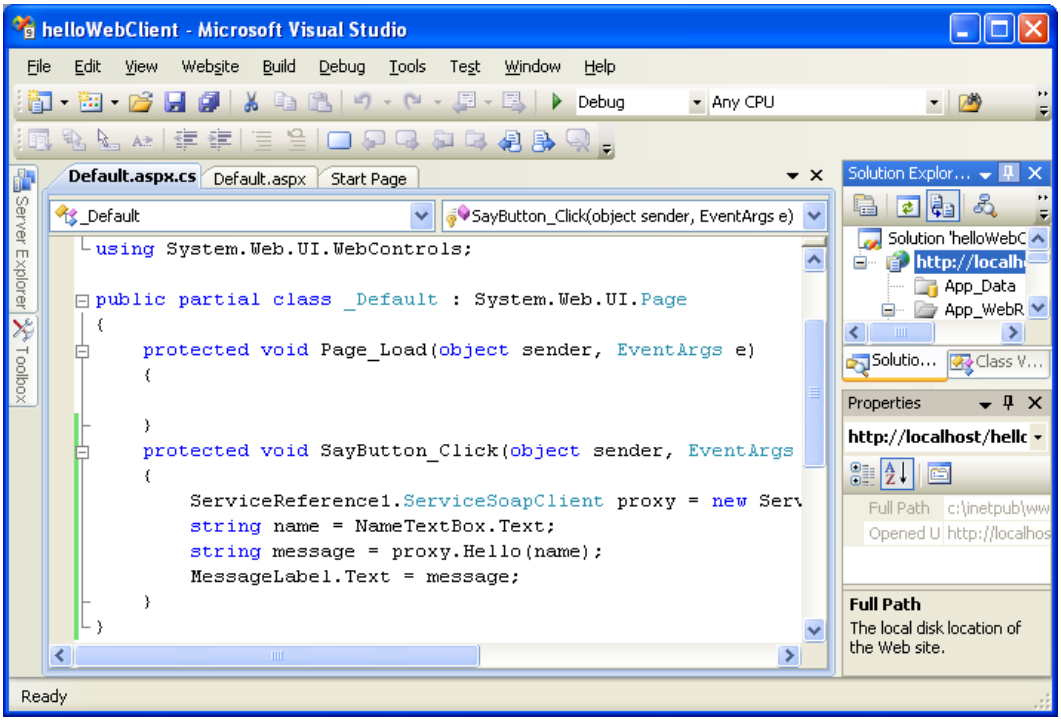
```
protected void SayButton Click(object sender, System.EventArgs e)
{
    ServiceReference1.ServiceSoapClient proxy =
      new ServiceReference1.ServiceSoapClient();
    string name = NameTextBox.Text;
    string message = proxy.Hello(name);
    MessageLabel.Text = message;
}
```

[To see how we can find the proxy class name, in the method body, first type the namespace "ServiceReference1" for the Web service reference. Type period after the namespace and wait. Choose "ServiceSoapClient" in the popup menu.]

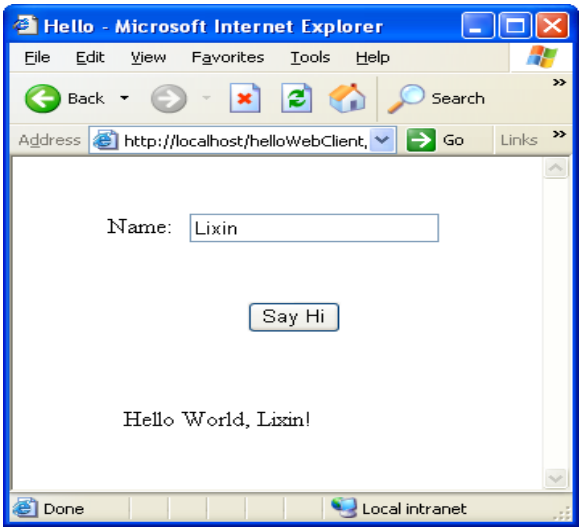
In this method body, line 1 creates a proxy object for our *Hello* .NET Web service; line 2 copies the string that a user has typed in the TextBox into variable *name*; line 3 invokes the proxy method "Hello()" with argument *name*, and saves the return string in variable *message*; and line 4 copies the string in variable *message* into MessageLabel for display. Even though the method call on line 3 is to the local proxy object, the proxy's body for method "Hello()" makes a TCP/IP connection to the ASP .NET component specified in the WSDL file, issues an HTTP request carrying a SOAP request message as its *entity body*, receives a response SOAP message from the ASP .NET component (from the *entity body* of the HTTP response), parses the SOAP response message into a C# string value, and returns the value as its own method return value.

As we can see from this example, .NET hides away more implementation details from software developers.

15. Save all files by clicking on "File|Save All". The *Visual Studio* IDE now looks like the following:



16. Now we can test our client Web application. Click on menu item "Debug|Start Without Debugging" or type key combination Ctrl+F5. If any error message is reported, you need to redo some of the steps and remove the bugs in the code. *Visual Studio* will then compile your project and deploy it in your IIS Web server, and a new *Internet Explorer* window will be launched at http://localhost/helloWebClient/ for you to test the new Web client application. The following screen capture is for my test session. Just type your name in the textbox, and click on the "Say Hi" button. Your name will be transmitted to the (potentially remote) Web service implementation, which will then return a personalized message for your Web client to display.



8 Development of a Java client application for consuming the .NET Web service

(You need VM WinXP2 to work on this section.) As the last step of the Lab, we are going to develop a Java standalone client application for our new .NET *Hello World* Web service. This will show you that .NET Web services and Java Web services are fundamentally based on the same standards, and Web services and clients based on these two technologies can interact with each other.

1. First, make sure your IIS Web server is running. You can open a Web browser and direct it to http://localhost/. If the IIS Welcome page is not displayed, you need to start your IIS server by following my instructions in the document "Software Installation for .NET Web Services".

2. Open a *Command Prompt* window by clicking on "Start|All Programs|Accessories|Command Prompt".

3. Change directory to "C:\webServiceLab\dotNet\javaClient". You can do so by typing the following in the *Command Prompt* window ("[Enter] means typing the Enter key):

   - C: [Enter]
   - cd C:\webServiceLab\dotNet\javaClient [Enter]

6. Type the following command in the *Command Prompt* window to generate source files for your proxy class for your new .NET *Hello* Web service:

   makeProxy [Enter]

   makeProxy.bat is a batch file that I created for saving your typing. Run this batch file is the same as running the contents of this file, which is

   ```
   java org.apache.axis.wsdl.WSDL2Java
        http://localhost/hello/Service.asmx?WSDL
   ```

   (there should be no line break in the file contents). This command will generate five Java source files supporting the Web service proxy class: *Service.java*, *ServiceLocator.java*, *ServiceSoap12Stub.java*, *ServiceSoap.java*, and *ServiceSoapStub.java*. These five Java files all belong to Java package "edu.pace.csis" (since we used *class attribute* "[WebService(Namespace="http://csis.pace.edu/")]" for our *Hello* Web service implementation class), therefore they are located in directory, relative to the current directory "C:\webServiceLab\dotNet\javaClient", "edu\pace\csis".

4. Now in directory "C:\webServiceLab\dotNet\javaClient", use a text editor, like *EditPad*, to create a Java source file named "HelloWorldClient.java" with its contents specified below (for your convenience, the file has been created for you; please review its contents):

```
import edu.pace.csis.*;

public class HelloWorldClient {
  public static void main(String[] args) throws Exception {
    String name = ""; // name to be sent to the Web service
    // The program expects to receive a string name on command-line
    // If you have spaces inside the name, enclose the entire name inside double quotes
    // Program quits if there is no such name string
    if (args.length == 1)  // there is one command-line argument
      name = args[0];
    else {
      System.out.println("Usage: java HelloWorldClient  \"[your name]\"");
      System.exit(-1);  // terminate the program
    }
    // Get the proxy factory
    ServiceLocator factory = new ServiceLocator();
    // Generate the Web service proxy object
    ServiceSoap proxy = factory.getServiceSoap();
    // Invoke Web service method to retrieve a personalized message
    String message = proxy.hello(name);
    System.out.println(message);
  }
}
```

Program 3 HelloWorldClient.java

The file first imports all the Java classes generated by "makeProxy.bat" for supporting the Web service proxy class. Then it checks whether there is a command-line argument. If the number of command-line arguments is not one, the program prints a usage message and then terminates. Otherwise it saves the argument string in variable *name*. A factory object for generating a proxy object is then created with "new ServiceLocator()", and a new proxy object is obtained by calling the factory's method "getServiceSoap()". The actual Web service invocation happens on the line with "proxy.hello(name)". Even though this method call is to the local proxy object, the proxy's body for method "hello" makes a TCP/IP connection to the ASP .NET component specified in the WSDL file, issues an HTTP request carrying a SOAP request message as its *entity body*, receives a response SOAP message from the ASP .NET component (from the *entity body* of the HTTP response), parses the SOAP response message into a Java String value, and returns the value as its own method return value.

7. Now it's time to compile the client program. In the *Command Prompt* window, type

C:\webServiceLab\dotNet\javaClient> **javac HelloWorldClient.java -source 1.4**

You only need to type the bold face part of the above line. The *javac* command-line switch "-source 1.4" is for informing the Java compiler that this class should be compiled with Java SDK 1.4's API. This is because in Java SDK 1.6, "enum" is defined as a Java key word, but they are not in Java SDK 1.4 or earlier. The *AXIS* toolkit uses "enum" extensively as a package name, which is not allowed in Java SDK version after 1.4. You may see warnings related to this fact, which are harmless. After executing this command, you will notice a new file "HelloWorldClient.class".

8. Now you can test your *Hello* .NET Web service with your new Java client program. Type the bold face part of the following line, and the second line is the response of your client program. The message is generated by your .NET Web service. You can try to use different strings as the command-line name string.

    C:\webServiceLab\dotNet\javaClient> **java HelloWorldClient "Lixin Tao"**
    Hello World, Lixin Tao!

Congratulations! You have successfully implemented a Java Web service client application for receiving service from a .NET Web service. The class name of your client application is not important and you can choose any name meaningful to you.

So far you have successfully completed all hands-on lab components for our workshop. This is a good jump-start for your in-depth study of Web services and other related technologies.