Demo: Integrating a .NET-based user interface with a Java back end

Version 12.1

jnbridge.com

JNBridge, LLC

jnbridge.com

Feb 6, 2026

Note: The JDBC-ODBC bridge has been removed from Java 8. To work through this example, use Java 7. You can also migrate the JDBC-ODBC classes from Java 7 to Java 9 as described here: http://stackoverflow.com/questions/34345743/how-can-i-configure-an-excel-spreadsheet-as-a-javax-sql-datasource/34617075#34617075.

Introduction

This document shows how JNBridgePro can be used to construct a .NET Windows or Web-based application that calls Java classes. The reader will learn how to generate .NET proxies that call the Java classes, create .NET code that calls the proxies and, indirectly, the corresponding Java classes, and set up and run the code.

There are situations where the strategy in integrating a .NET-based user interface with Java back end software may be desirable. The developer may have a perfectly satisfactory Java library performing some functionality (for example, accessing a database), but deployment may require that the user interface run on a .NET platform. For example, a desktop application may require a .NET look-and-feel, or an otherwise .NET-based application may need to use the Java library's functionality. Alternatively, the Java library may be part of an ASP.NET-based Web application.

In this example, we assume an existing Java library to access a database of authors and books. A class Author contains methods to access all the authors in the database, and a class Book contains methods to access all the books in the database or to access all the books by a given author. Needless to say, the code is greatly simplified; it's sufficient to illustrate the concept. The example illustrates the creation of two .NET-based front ends that access the Java library: a rich GUI using Windows Forms, and a Web-based GUI using ASP.NET.

Generating the proxies

While this example uses the standalone proxy generation tool, you can also use the Visual Studio plug-in, and the example figures will look very much the same.

The first step in the process is to generate proxies for the Author and Book classes. Start by launching JNBProxy, the GUI-based proxy generator. When the "Launch JNBProxy" form is displayed (Figure 1), select "Create new .NET → Java project." Once this is done, JNBProxy's main form is displayed (Figure 2).

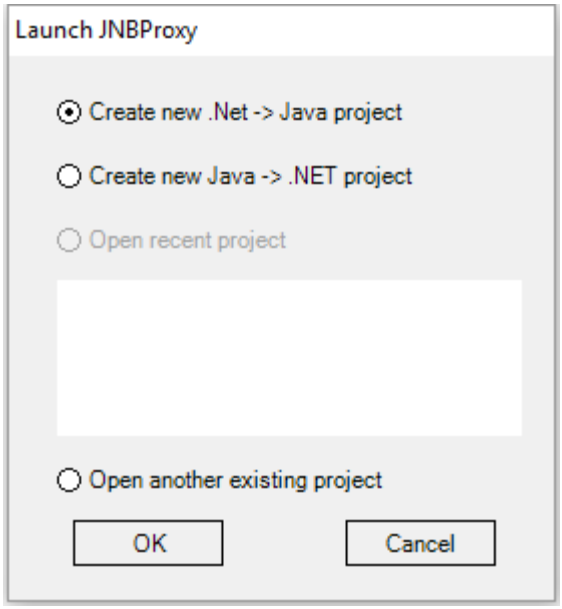

Figure 1. JNBProxy launch form

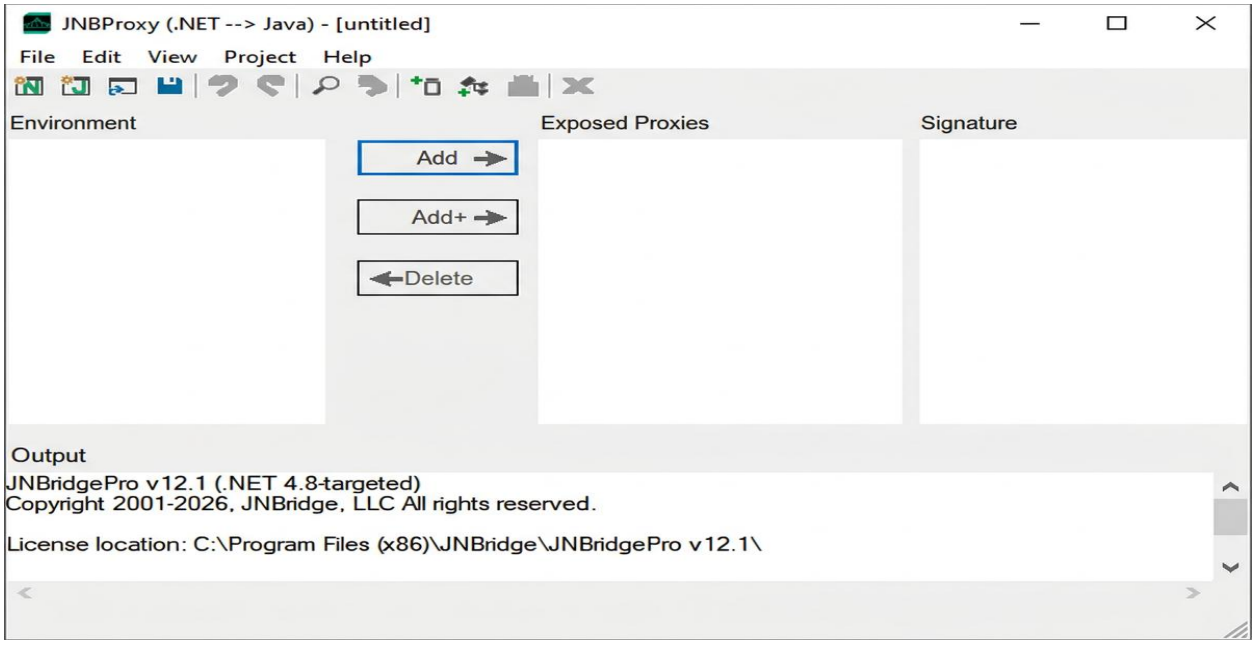

Figure 2. JNBProxy

Next, add the folder in which Demo2\Author.class and Demo2\Book.class are to be found. Use the menu command **Project→Edit Classpath...**. The **Edit Class Path** dialog box will come up, and clicking on the **Add...** button will bring up a dialog that will allow the user to indicate the path of the class files (Figure 3).

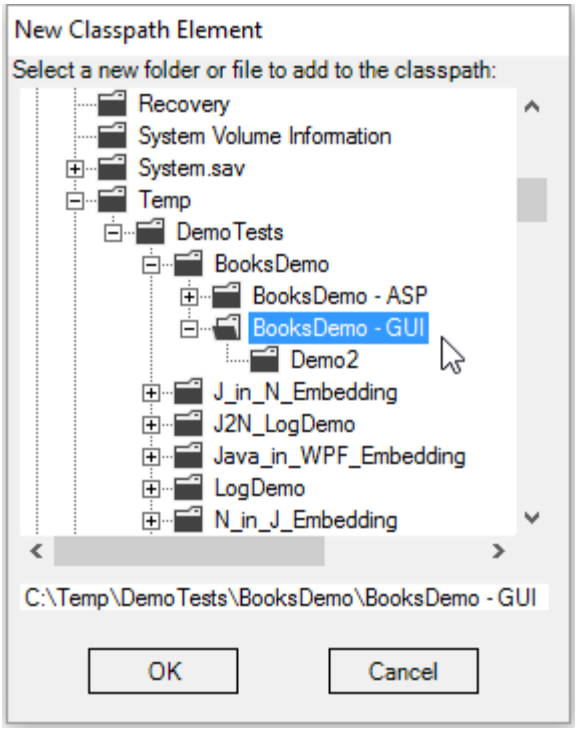

Figure 3. Adding a new classpath element

When all the necessary elements of the classpath are added, the **Edit Class Path** dialog should contain information similar to that shown in Figure 4.

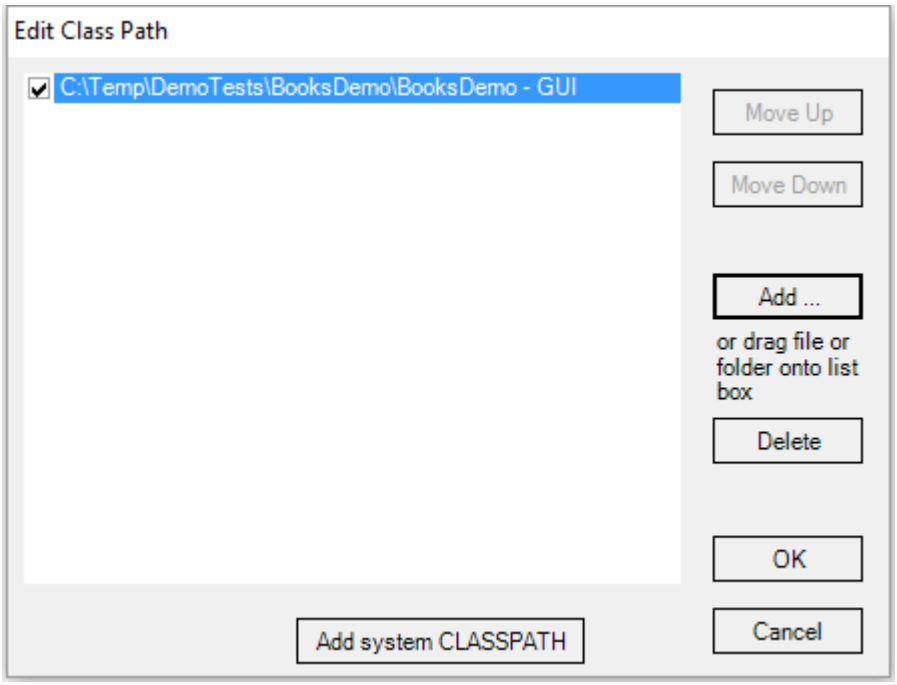

Figure 4. After creating classpath

The next step is to load the Author and Book Java classes. Use the menu command **Project→Add Classes from Classpath...** and enter the fully qualified class names Demo2.Author and Demo2.Book (Figure 5). It is not necessary to add supporting classes, since the only supporting classes needed to use Author and Book are arrays and strings, which are automatically converted to native .NET arrays and strings.

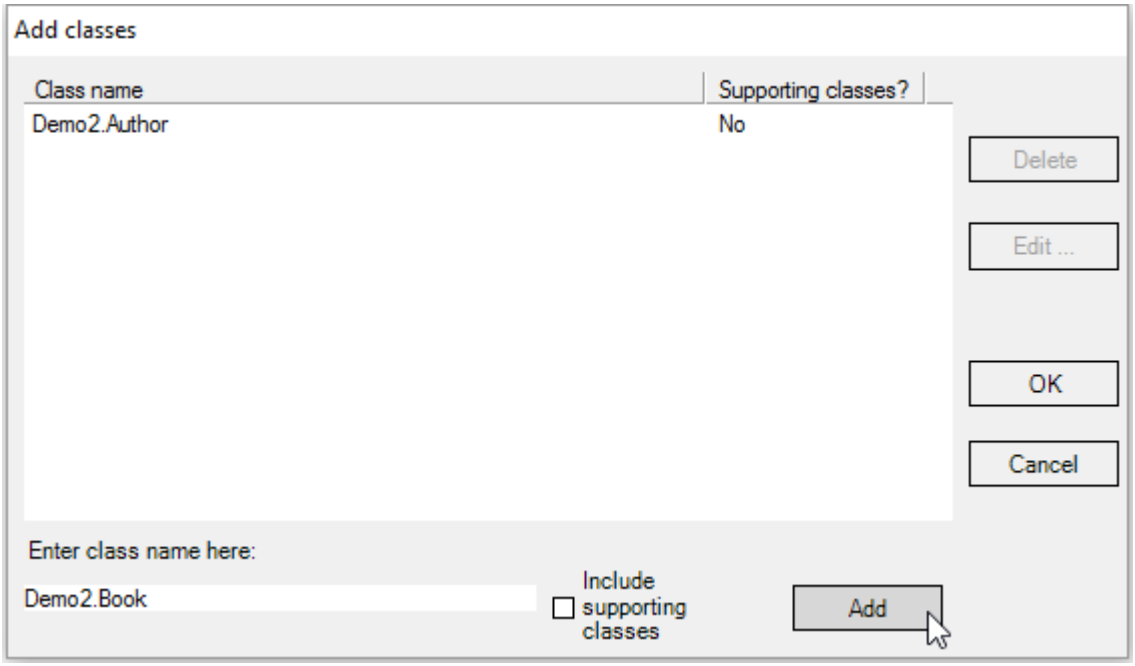

Figure 5. Adding a class from the classpath

Loading the classes may take a few seconds. Progress will be shown in the output pane in the bottom of the window, and in the progress bar. When completed, Demo2.Author and Demo2.Book will be displayed in the Environment pane on the upper left of the JNBProxy window (Figure 6).

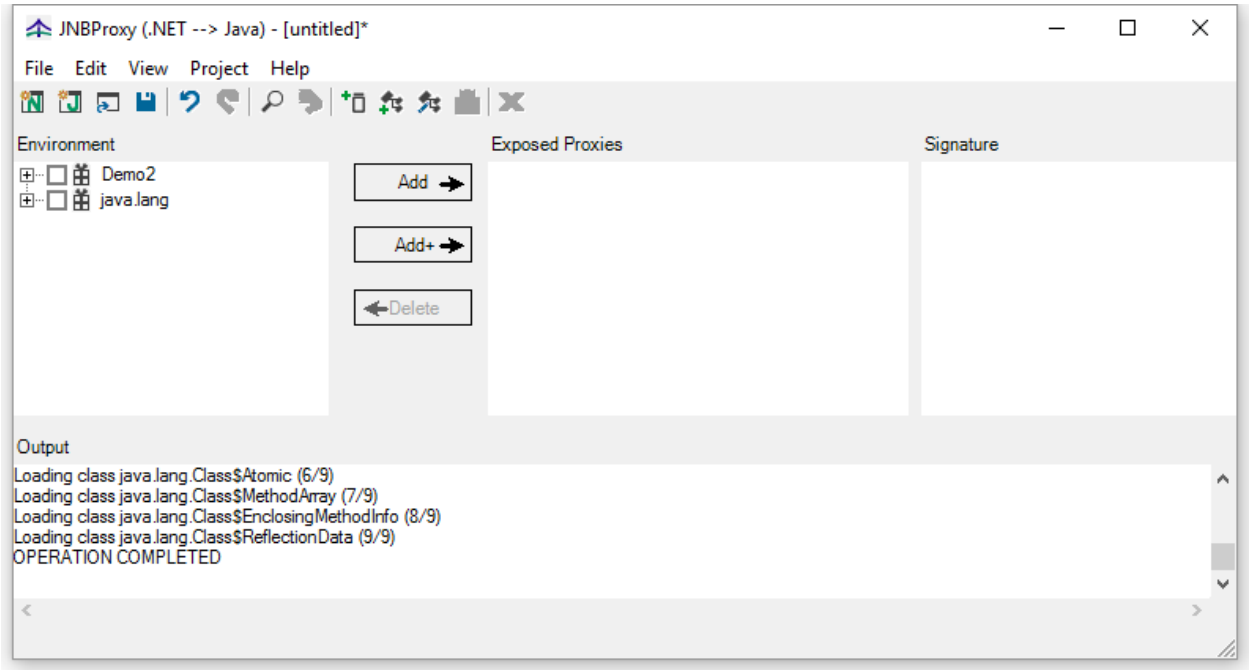

Figure 6. After adding classes

We wish to generate proxies for both of these classes, so when all the classes have been loaded into the environment, make sure that each class in the tree view has a check mark next to it. Quick ways to do this include clicking on the check box next to each package name, or simply by selecting the menu command **Edit→Check All in Environment**. Once each class has been checked, click on the **Add** button to add each checked class to the list of proxies to be exposed. These will be shown in the Exposed Proxies pane (Figure 7).

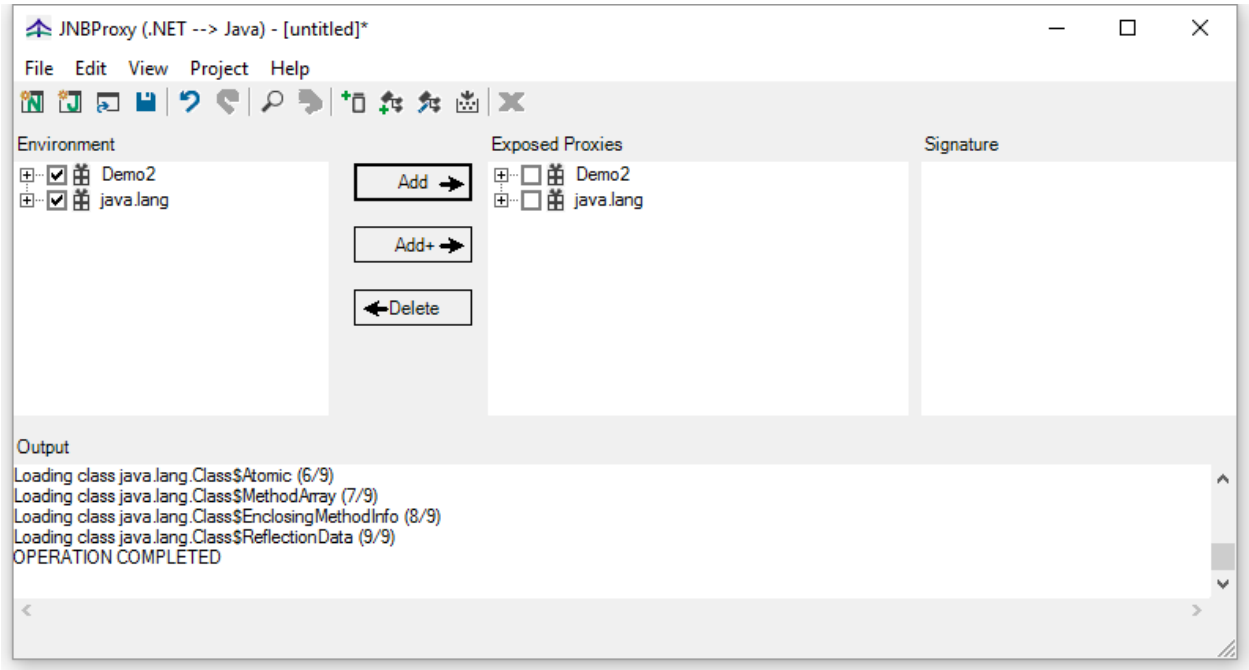

Figure 7. After adding classes to Exposed Proxies pane

We are now ready to generate the proxies. Select the **Project→Build...** menu command, and choose a name and location for the assembly (.dll) file that will contain the generated proxies. The proxy generation process may take a few minutes, and progress and other information will be indicated in the Output pane. In this example, we will call the generated proxy assembly bookAccess.dll.

Using the proxies - WinForms

Now that the proxies have been generated, we can use them to access Java classes from .NET. Launch Visual Studio .NET (note that this development can also be done using the .NET SDK), and create a new C# Windows application project. Add references to the assemblies bookAccess.dll (the one just generated) and jnbshare.dll (distributed with JNBridgePro). Add to your project the file jnbauth_x86.dll or jnbauth_x64.dll or both (depending on whether the application will run as a 32-bit process, a 64-bit process, or either one). Add the file app.config to the project. It is an application configuration file that configures the .NET side of JNBridgePro. You may want to examine the settings (it is set to communicate with the Java side using tcp/binary communications, where the Java side is on the same machine as the .NET side and is listening on port 8085). Make sure that, depending on whether you are using .NET Framework 2.0 or 4.0, you have commented and uncommented the appropriate sections of the configuration file. Using the Windows Forms designer, create a Windows form with two ListViews, as shown in Figure 8, below. The procedure for constructing this form is beyond the scope of this document, but we provide sample code in the file Form1.cs.

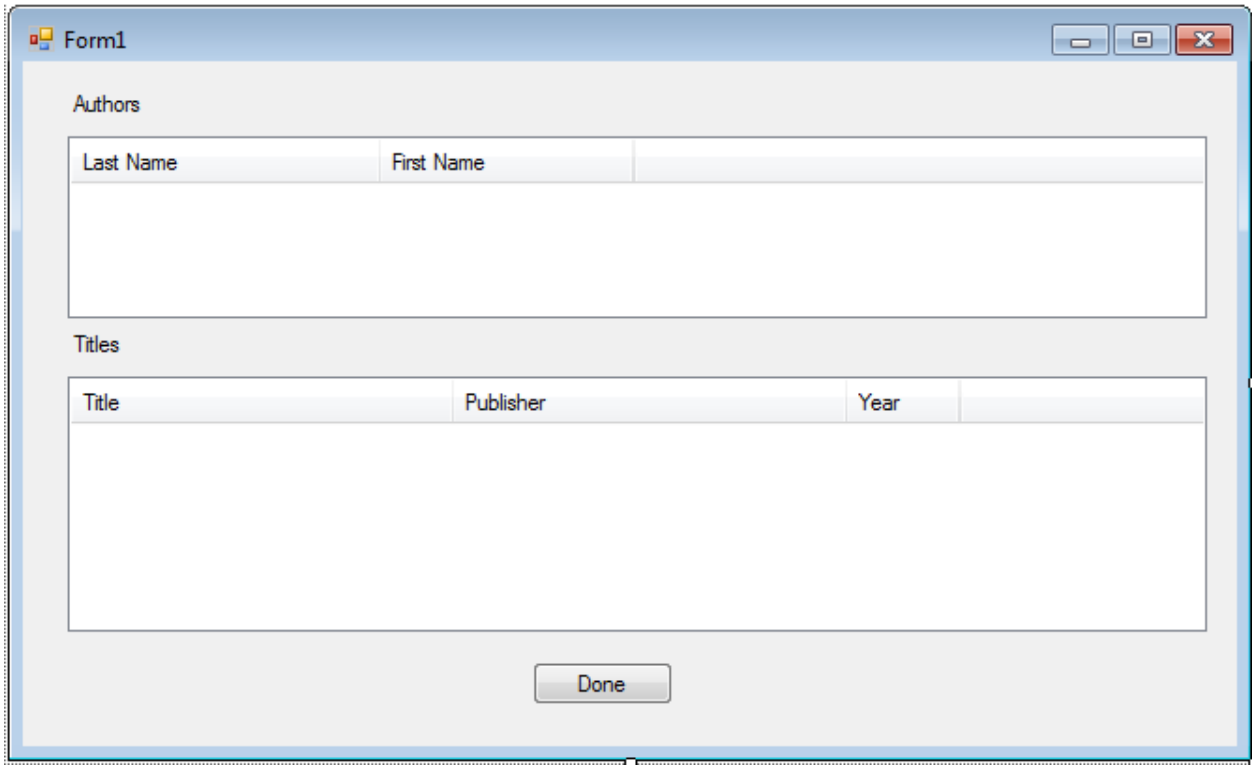

Figure 8. Windows form for example program

Assuming that the top ListView is named **authors**, the bottom ListView is named **titles**, and the button is named **button1**, add the following event handlers :

```
using Demo2;

. . .

// Click event handler for button1
private void button1_Click(object sender, System.EventArgs e)
{
    Application.Exit();
}

// Load event handler for Form1
private void Form1_Load(object sender, System.EventArgs e)
{
    Author[] authorList = Author.getAuthors();
    for (int i = 0; i < authorList.Length; i++)
    {
           ListViewItem li = new ListViewItem();
           li.Text = authorList[i].lastName;
           li.SubItems.Add(authorList[i].firstName);
           authors.Items.Add(li);
    }
}

// SelectedIndexChanged event handler for authors
private void authors_SelectedIndexChanged(object sender, System.EventArgs e)
{
    // clear the titles list
```


```
    titles.Items.Clear();
    if (authors.SelectedItems.Count == 0)
    {
           return;
    }

    // get the selected item
    ListViewItem li = authors.SelectedItems[0];

    // get the first and last names
    string lastName = li.Text;
    string firstName = li.SubItems[1].Text;


    // get the Books array
    Book[] books = Book.getBooks(firstName, lastName);

    // load the books information
    for (int i = 0; i < books.Length; i++)
    {
           ListViewItem li2 = new ListViewItem();
           li2.Text = books[i].title;
           li2.SubItems.Add(books[i].publisher);
           li2.SubItems.Add(books[i].year);
           titles.Items.Add(li2);
    }
}
```

The proxies for the Java objects of class Author and Book are used exactly as the original objects would be used in Java. Note the following items of interest:

Proxies for the Java classes have namespaces identical to the package names of the original Java classes. Thus, we simply import the namespace Demo2, and afterwards can use the names of the Java classes.

Proxies for the Java classes Author and Book are used in exactly the same way as the original Java classes would have been used.

When typing in the calls to the Java objects, Visual Studio's IntelliSense facility will offer to complete the names of methods and fields in the same way that it would for calls to .NET objects (Figure 9), and will provide information on number and types of parameters.

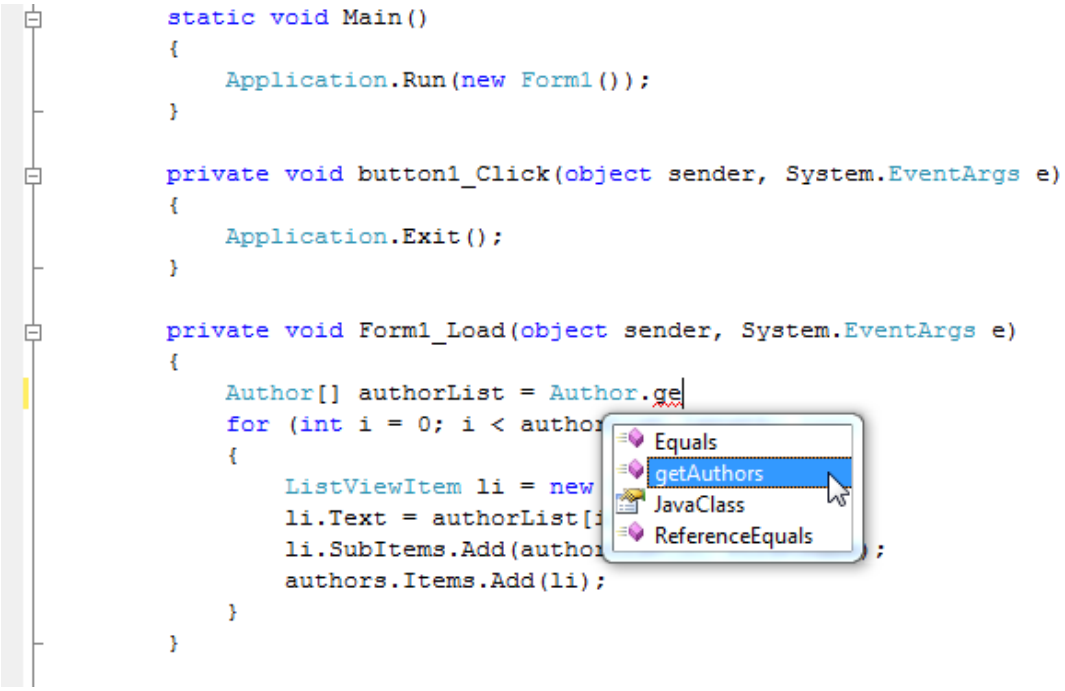

Figure 9. IntelliSense method completion for Java calls

After entering the code, build the project to obtain the executable.

Running the program - WinForms

Running the program is simple. First, make sure that there is an ODBC data source called 'BookDemo' referencing the supplied Microsoft Access database books.mdb. Make sure that JNBridgePro is properly configured on the .NET side (i.e., app.config has been added to the project – upon building the solution, app.config will be copied to your project's build folder and renamed *projectName*.exe.config, assuming your exe file is *projectName*.exe) and on the Java side (i.e., that there is a copy of the properties file jnbcore.properties in the same folder as jnbcore.jar), and that the .NET and Java side configurations agree on the protocol and port to be used. Then, start up a JVM. Assuming that the folder Demo2 containing the Java class files is in the same folder, we can start up the Java-side in a console window as follows:

```
java -cp ".;jnbcore.jar" com.jnbridge.jnbcore.JNBMain
```

Double-click on the icon representing the compiled .NET program. The program's form will be displayed with the author information obtained from the database through the Java classes (Figure 10). Clicking on one of the author names will cause books written by that author to be displayed (Figure 11), again, where the title information is supplied by the Java classes. Clicking on the "Done" button will cause the application to exit.

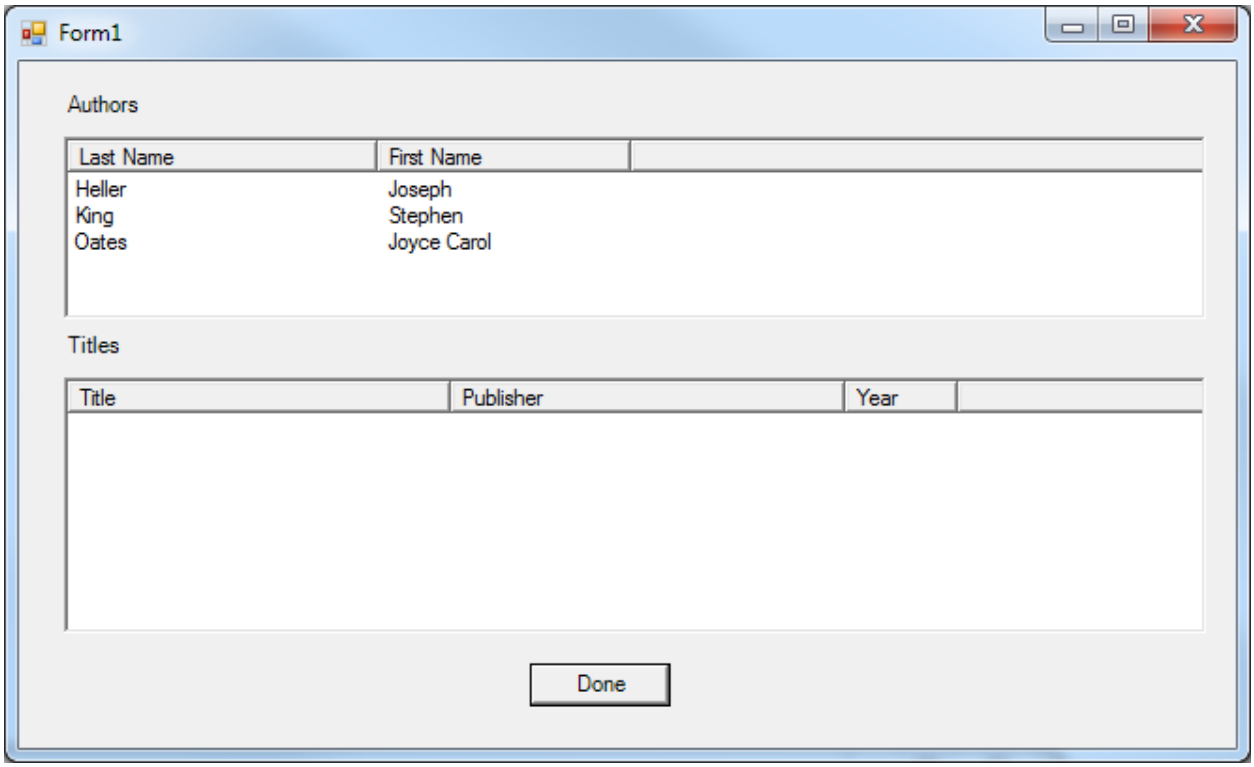

Figure 10. .NET application when first activated.

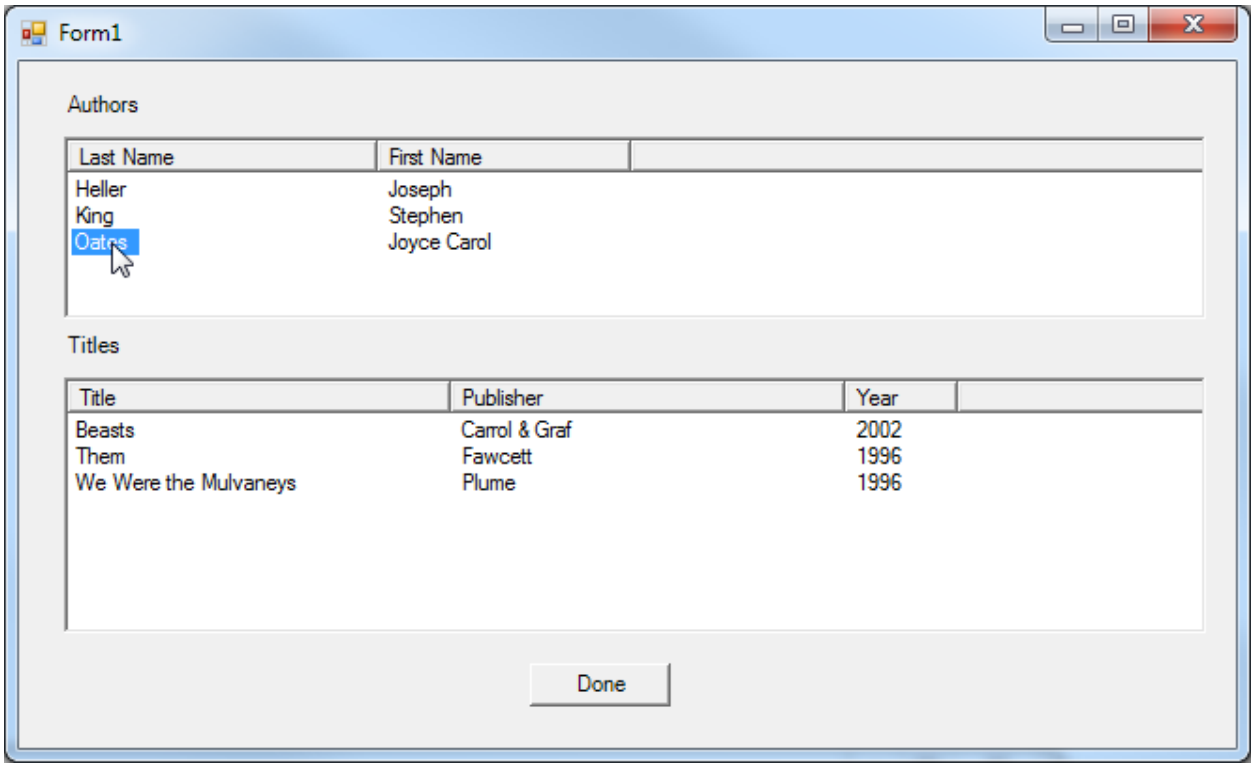

Figure 11. After clicking on author name.

Using shared-memory communication. It is possible to run the Java side in the same process as the .NET side, using a shared-memory communication mechanism. This has several advantages: it's much faster than the socket-based tcp/binary and http/soap mechanisms, and it's not necessary to explicitly start up the Java side – it's automatically done before the first call to a proxy. To use shared memory, stop the .NET and Java sides (if they're still running), then open the app.config application configuration file. Comment out the <dotNetToJavaConfig> element whose "scheme" value is "jtcp," and uncomment the <dotNetToJavaConfig> element whose "scheme" value is "sharedmem." You will need to edit the "jvm," "jnbcore," "bcel," and "classpath" values to reflect the locations on your machine of jvm.dll, jnbcore.jar, bcel-6.10.0.jar, and the classpath classes Demo2.Author and Demo2.Book. Once you have made the changes, build the project and start it. It will run as before, even though the Java side has not been explicitly started, since the Java side is now running inside the .NET process.

Class whitelisting. When using TCP/binary communications, the Java side can be configured to only allow requests from the .NET side that reference specific Java classes. This prevents the possibility of malicious clients accessing sensitive APIs (which need not even have been proxied). The file jnbcore_tcp_no_security.properties contains the following properties to activate the class whitelist feature:

```
javaSide.useClassWhiteList=true
```

This is the default value and may be omitted. To turn off class whitelisting, the property must be explicitly set to false.

```
javaSide.classWhiteListFile=./classWhiteList.txt
```

The property above is the path to a text file each of whose lines is a class that can be accessed from the .NET side. If the .NET side client attempts to access a class not in the whitelist (or one of the short list of classes that is always whitelisted), an exception will be thrown. The supplied whitelist file contains the following classes that are directly accessed from the .NET side:

```
org.apache.log4j.Category
org.apache.log4j.BasicConfigurator
loggerDemo.JavaClass
```

The class whitelist can be easily derived by examining the .NET side code that calls the proxies. For each proxy class that is called, add that class or interface name to the whitelist.

For more information on class whitelisting, see the *Users' Guide*.

Secure communication using SSL. It is possible to configure secure communications between the .NET and Java sides through SSL (secure sockets library). SSL in JNBridgePro provides data encryption, message integrity, and server communications. It is only available when using tcp/binary communications (shared memory is inherently secure). For more information on secure communications, see the *Users' Guide*.

Please note that the following instructions use certificates that we supply. These certificates are for instructional use only; you should NOT use them in production scenarios. For production scenarios, you should supply your own certificates.

To use SSL, first make sure that the example is configured to use tcp/binary communications without SSL (that is, the appropriate useSSL properties are set to false), and that this is working.

Once it is established that the application works with regular tcp/binary communications, we configure for SSL. First, add the attribute *useSSL="true"* to the <dotNetToJavaConfig> element in

the app.config file. Also, add the *javaSide.useSSL=true* property to the jnbcore.properties file that you will be using when you run the Java side. To turn SSL off, these properties may be omitted (the default is *false*, or explicitly set to *false*.

On the .NET side, in the app.config file, comment out the version of <dotNetToJavaConfig> without the security features, and uncomment the version of <dotNetToJavaConfig> with the security features. Note the following elements:

- *useSSL* – this indicates that SSL is being used, and should be set to true
- *clientCertificateLocation* – this is the path to the .NET side's client certificate, and is used to authenticate itself to the server side, and also for encryption. This version of the client certificate must contain the public/private key pair, and should be password protected.
- *clientCertificatePassword* – this is the password of the client certificate.
- *sslAlternateServerNames* – this is a semicolon-separated list of server names that may be accepted when the server authenticates itself to the client. For example, in this case the .NET client is accessing the Java server on the same machine, so it is attempting to contact "localhost". However, the server certificate is for a server named "myServer" (the value in the CN/Common Name field of the certificate). Unless myServer appears in the sslAlternateServerNames list, the connection will fail.

On the Java side, we have the following additional properties:

- *javaSide.keyStore* – this is the path to a Java keystore (.jks) file containing the public/private key pair for the Java-side server's certificate.
- *javaSide.keyStorePassword* -- this is the password of the keystore file.
- *javaSide.trustStore* – this is another .jks file containing a list of trusted certificates. You should place the authorized .NET sides' certificates in this folder.
- *javaSide.trustStorePassword* – this is the password of the truststore file.

Since the Java-side server certificate (in this case, myserver.cer) is a self-signed certificate, we have to explicitly instruct the .NET side to trust it. To do so, copy the certificate to the .NET-side machine and install it into the certificate store by right-clicking on the .cer file and selecting Install…. In the resulting wizard, choose to install the certificate in either the machine store or the user store. In the next step, when asked where the certificate should be stored, select either "Trusted Root Certification Authorities" or "Third-Party Root Certification Authorities." After that selection, follow all remaining instructions.

At this point, rebuild the .NET side, and start the Java side with the command-line:

```
java -cp ".;log4j.jar;log4j-core.jar;jnbcore.jar" com.jnbridge.jnbcore.JNBMain
/props ".\jnbcore_tcp_with_security.properties"
```

Run the .NET side. It should work as previously, where security features were not used, except in this case the .NET-side client and the Java-side server are both authenticated, and communications are encrypted.

Using the proxies – ASP.NET

It is also possible to access Java classes behind a Web interface running on ASP.NET. The process is similar to that used with WinForms. First, create a new C# Web Application project. (Make sure that you have Internet Information Server installed and properly configured before you do this.) Replace

the file web.config in the project with the web.config file that is supplied with this example. It is an application configuration file that configures the .NET side of JNBridgePro. You may want to examine the settings (it is set to communicate with the Java side using tcp/binary communications, where the Java side is on the same machine as the .NET side and is listening on port 8085). Make sure that, depending on whether you are using .NET Framework 2.0 or 4.0, you have commented and uncommented the appropriate sections of the configuration file. Use the designer to create a simple WebForm as shown in Figure 12. The two boxes in the WebForm are ListBoxes, not ListViews, which are not available in WebForms. Make sure that the AutoPostBack property of the top ListBox is set to **true**.

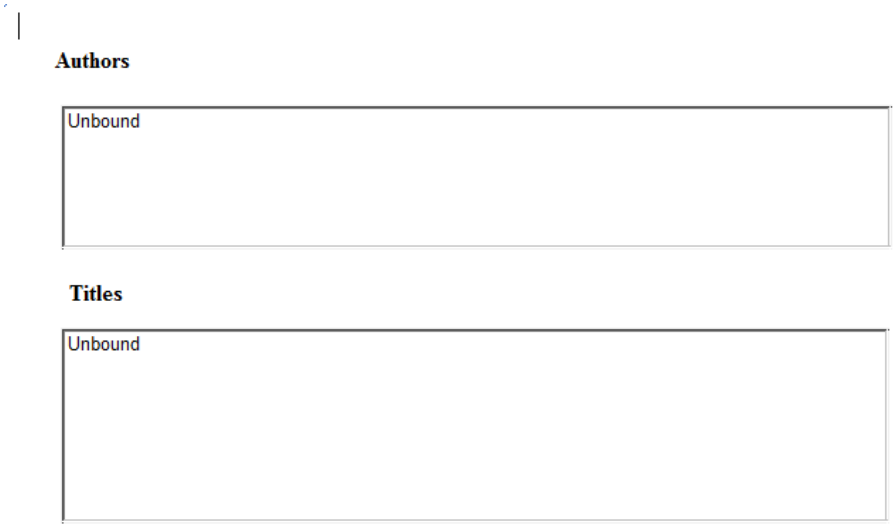

Figure 12. Web form for example program

Assuming that the top ListBox is named **ListBox1**, and the bottom ListBox is named **ListBox2**, add the following event handlers to the WebForm's codebehind :

```
using Demo2;
. . .
private void ListBox1_Load(object sender, System.EventArgs e)
{
    if (ListBox1.Items.Count > 0)
    {
           return;       // don't add anything more
    }

    // load the authors
    Author[] authors = Author.getAuthors();
    for (int i = 0; i < authors.Length; i++)
    {
           ListItem li = new ListItem();
           li.Text = authors[i].lastName + ", " + authors[i].firstName;
           ListBox1.Items.Add(li);
    }
}

private void ListBox1_SelectedIndexChanged(object sender, System.EventArgs e)
{
```

```
    // clear listBox2
    ListBox2.Items.Clear();

    // if nothing selected, return
    ListItem li = ListBox1.SelectedItem;

    if (li == null)
    {
           return;
    }

    // last, first names
    string name = li.Text;
    int separator = name.IndexOf(",");
    string lastName = name.Substring(0, separator);
    string firstName = name.Substring(separator+2);

    // look up books
    Book[] titles = Book.getBooks(firstName, lastName);

    // write out books
    for (int i = 0; i < titles.Length; i++)
    {
           ListItem li2 = new ListItem();
           string theBook = titles[i].title + "   (" + titles[i].publisher + ", " +
                              titles[i].year + ")";
           li2.Text = theBook;
           ListBox2.Items.Add(li2);
    }
}
```

Again, note that the proxies for the Java objects of class Author and Book are used exactly as the original objects would be used in Java.

After entering the code, build the project to obtain the Web application.

Running the program – ASP.NET

Running the program is simple. First, make sure that there is an ODBC data source called 'BookDemo' referencing the supplied Microsoft Access database books.mdb. Make sure that JNBridgePro is properly configured on the .NET side (i.e., web.config has been added to the project) and on the Java side (i.e., that there is a copy of the properties file jnbcore.properties in the same folder as jnbcore.jar), and that the .NET and Java side configurations agree on the protocol and port to be used. Then, start up a JVM. Assuming that the folder Demo2 containing the Java class files is in the same folder, we can start up the Java-side in a console window as follows:

```
java -cp ".;jnbcore.jar" com.jnbridge.jnbcore.JNBMain
```

Start the Web application by launching a browser and entering the URL of the Web application (for example, http://localhost/Demo2/WebForm1.aspx). The WebForm will be displayed with author names, obtained through the Java class Author, displayed in the top list box (Figure 13). Clicking on any of the authors will cause books written by that author to be displayed (Figure 14).

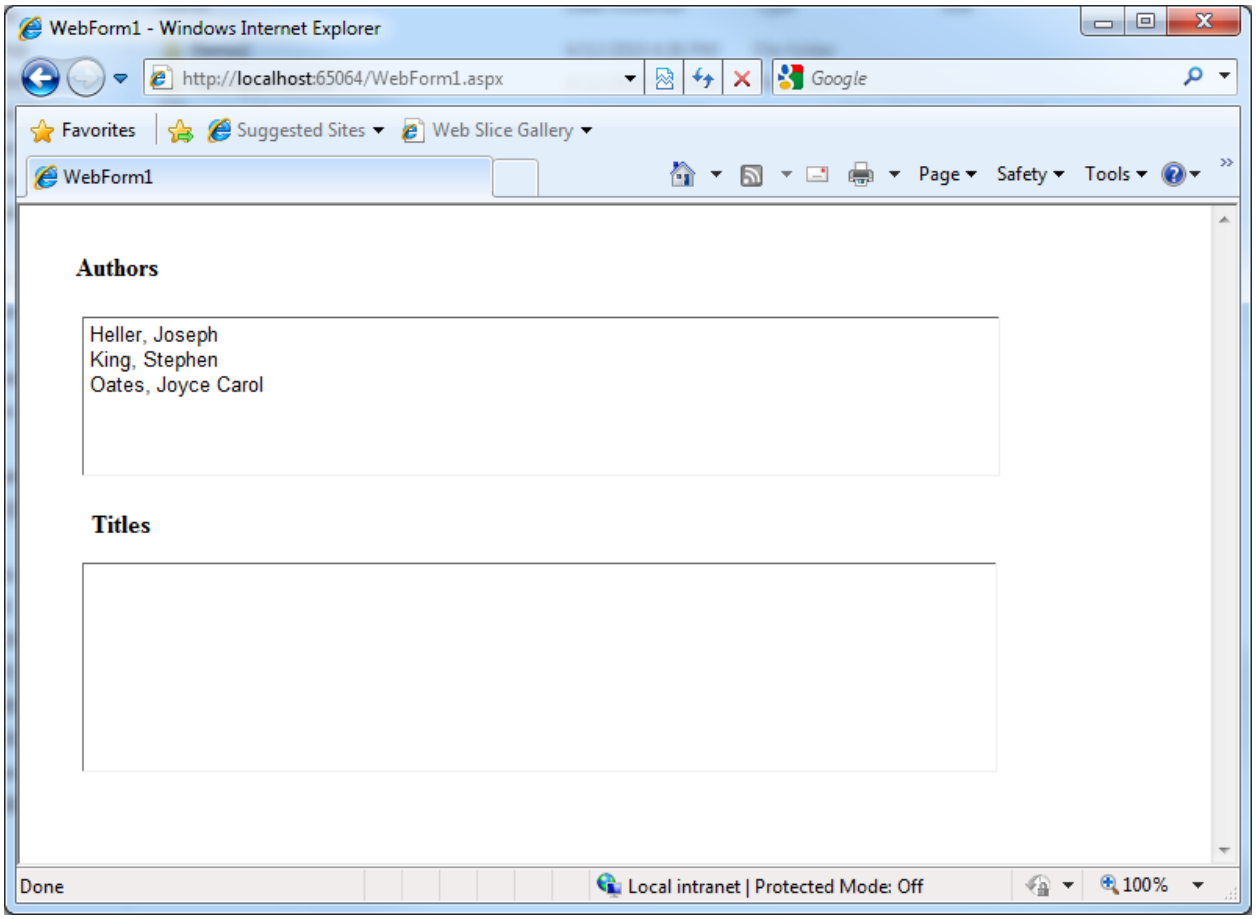

Figure 13. Initial WebForm.

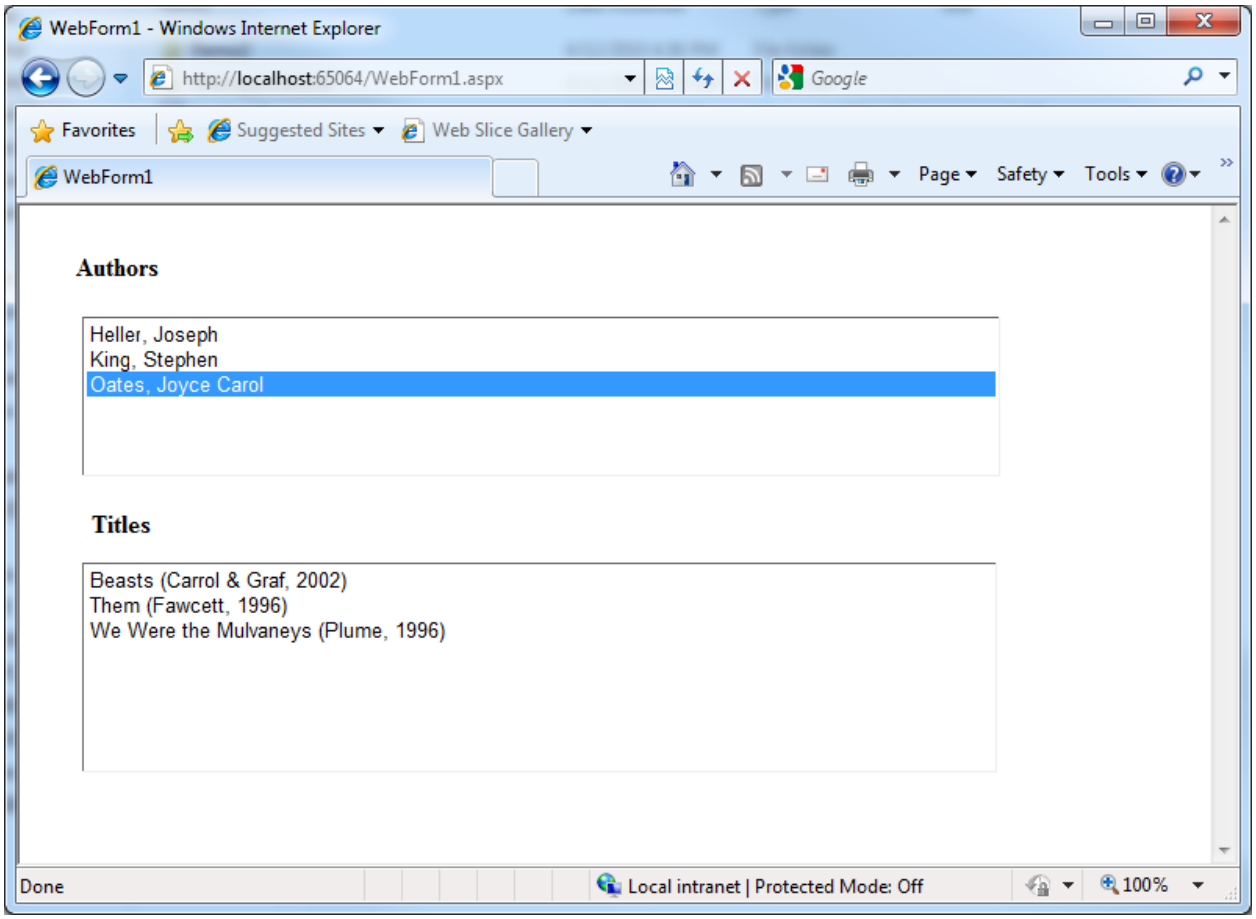

Figure 14. WebForm after clicking on author.

Using shared-memory communication. It is possible to run the Java side in the same process as the .NET side, using a shared-memory communication mechanism. This has several advantages: it's much faster than the socket-based tcp/binary and http/soap mechanisms, and it's not necessary to explicitly start up the Java side – it's automatically done before the first call to a proxy. To use shared memory, stop the .NET and Java sides (if they're still running), then open the web.config application configuration file. Comment out the <dotNetToJavaConfig> element whose "scheme" value is "jtcp," and uncomment the <dotNetToJavaConfig> element whose "scheme" value is "sharedmem." You will need to edit the "jvm," "jnbcore," "bcel," and "classpath" values to reflect the locations on your machine of jvm.dll, jnbcore.jar, bcel-6.10.0.jar, and the classpath classes Demo2.Author and Demo2.Book. Once you have made the changes, build the project. Restart ASP.NET by issuing the command "iisreset" from a command-line prompt, then start the Web application. It will run as before, even though the Java side has not been explicitly started, since the Java side is now running inside the .NET process.

Note that if you see a problem, particularly a ClassNotFoundException, you will need to make sure that the account running the ASP.NET process (usually called ASPNET) has "Read & Execute," "List Folder Contents," and "Read" permissions to the folders containing the jar and class files. (See http://jnbridge.com/support/knowledge-base/classnotfoundexceptionnoclassdeffounderror for more information.)

Class whitelisting. When using TCP/binary communications, the Java side can be configured to only allow requests from the .NET side that reference specific Java classes. This prevents the possibility of malicious clients accessing sensitive APIs (which need not even have been proxied). The file jnbcore_tcp_no_security.properties contains the following properties to activate the class whitelist feature:

javaSide.useClassWhiteList=true

This is the default value and may be omitted. To turn off class whitelisting, the property must be explicitly set to false.

javaSide.classWhiteListFile=./classWhiteList.txt

The property above is the path to a text file each of whose lines is a class that can be accessed from the .NET side. If the .NET side client attempts to access a class not in the whitelist (or one of the short list of classes that is always whitelisted), an exception will be thrown. The supplied whitelist file contains the following classes that are directly accessed from the .NET side:

Demo2.Author
Demo2.Book

The class whitelist can be easily derived by examining the .NET side code that calls the proxies. For each proxy class that is called, add that class or interface name to the whitelist.

For more information on class whitelisting, see the *Users' Guide*.

Secure communication using SSL. It is possible to configure secure communications between the .NET and Java sides through SSL (secure sockets library). SSL in JNBridgePro provides data encryption, message integrity, and server communications. It is only available when using tcp/binary communications (shared memory is inherently secure). For more information on secure communications, see the *Users' Guide*.

Please note that the following instructions use certificates that we supply. These certificates are for instructional use only; you should NOT use them in production scenarios. For production scenarios, you should supply your own certificates.

To use SSL, first make sure that the example is configured to use tcp/binary communications without SSL (that is, the appropriate useSSL properties are set to false), and that this is working.

Once it is established that the application works with regular tcp/binary communications, we configure for SSL. First, add the attribute *useSSL="true"* to the <dotNetToJavaConfig> element in the app.config file. Also, add the *javaSide.useSSL=true* property to the jnbcore.properties file that you will be using when you run the Java side. To turn SSL off, these properties may be omitted (the default is *false*, or explicitly set to *false*.

On the .NET side, in the app.config file, comment out the version of <dotNetToJavaConfig> without the security features, and uncomment the version of <dotNetToJavaConfig> with the security features. Note the following elements:

- *useSSL* – this indicates that SSL is being used, and should be set to true
- *clientCertificateLocation* – this is the path to the .NET side's client certificate, and is used to authenticate itself to the server side, and also for encryption. This version of the client certificate must contain the public/private key pair, and should be password protected.
- *clientCertificatePassword* – this is the password of the client certificate.

- *sslAlternateServerNames* – this is a semicolon-separated list of server names that may be accepted when the server authenticates itself to the client. For example, in this case the .NET client is accessing the Java server on the same machine, so it is attempting to contact "localhost". However, the server certificate is for a server named "myServer" (the value in the CN/Common Name field of the certificate). Unless myServer appears in the sslAlternateServerNames list, the connection will fail.

On the Java side, we have the following additional properties:

- *javaSide.keyStore* – this is the path to a Java keystore (.jks) file containing the public/private key pair for the Java-side server's certificate.
- *javaSide.keyStorePassword* -- this is the password of the keystore file.
- *javaSide.trustStore* – this is another .jks file containing a list of trusted certificates. You should place the authorized .NET sides' certificates in this folder.
- *javaSide.trustStorePassword* – this is the password of the truststore file.

Since the Java-side server certificate (in this case, myserver.cer) is a self-signed certificate, we have to explicitly instruct the .NET side to trust it. To do so, copy the certificate to the .NET-side machine and install it into the certificate store by right-clicking on the .cer file and selecting Install…. In the resulting wizard, choose to install the certificate in either the machine store or the user store. In the next step, when asked where the certificate should be stored, select either "Trusted Root Certification Authorities" or "Third-Party Root Certification Authorities." After that selection, follow all remaining instructions.

At this point, rebuild the .NET side, and start the Java side with the command-line:

```
java -cp ".;log4j.jar;log4j-core.jar;jnbcore.jar" com.jnbridge.jnbcore.JNBMain
/props ".\jnbcore_tcp_with_security.properties"
```

Run the .NET side. It should work as previously, where security features were not used, except in this case the .NET-side client and the Java-side server are both authenticated, and communications are encrypted.

Summary

The above example shows how simple it is to integrate Java and .NET code and to run the resulting program. The example above shows how a program can integrate a .NET-based user interface with a Java-based back-end library. Two versions of the example were shown, one employing a rich user interface using Windows Forms, and one employing a Web-based interface using ASP.NET.

Creating this program was accomplished in three stages:

In the first stage, proxies were generated allowing access by .NET classes to the Java classes. The proxies were generated using JNBProxy, a visual tool that allows developers a wide variety of strategies for determining which Java classes are to be exposed to access by .NET.

In the second stage, the .NET assembly containing the proxies was linked to the .NET development project and .NET code accessing the Java classes was developed. .NET classes can access Java classes transparently, as if the Java classes were themselves .NET classes. Nothing special or additional needs to be done to manage Java-.NET communications or object lifecycles. Benefits provided by Visual Studio .NET, such as IntelliSense, are also available when writing .NET code that accesses Java classes.

In the third stage, the integrated .NET and Java code is run. All that is required is to start a Java-side containing the Java code to be accessed and an additional support module (jnbcore.jar). Once the Java-side is started, the user simply runs the .NET program that will access the Java objects. We have also shown how to use shared-memory communication to run the Java side in the same process as the .NET side so that the Java side need not be explicitly started.

By allowing Java and .NET code to interoperate, JNBridgePro helps developers derive full value from their existing Java code, even as they take advantage of Microsoft's .NET platform.