Demo: Calling a Java Logging Package from .NET

Version 12.0

jnbridge.com

JNBridge, LLC
jnbridge.com

April 13, 2021

Introduction

This document shows how JNBridgePro can be used to construct a .NET console 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.

In the example, JNBridgePro is used to allow .NET code to call log4j, a Java-based logging package developed as part of the Apache project. There are a number of reasons one might want to do such a thing. The developer may feel that this package is the best one for the job. Another, more compelling reason, might be that the developer is integrating .NET classes with existing Java classes that already use log4j to log events, and it would be desirable to have the Java- and .NET-originated logging messages go to the same output. Additionally, use of log4j by both Java and .NET code would allow logging to be controlled from a single configuration file, rather than requiring Java and .NET logging to be controlled from separate configuration files.

In this example, we assume an existing Java class, loggerDemo.JavaClass, that includes an instance method doIt() that sends a log message to log4j. We will create a .NET-based class, com.jnbridge.demos.logging.DotNetClass that includes its own instance method f() that also sends a log message to log4j. A .NET-based driver method calls both JavaClass and DotNetClass, and we will see how both Java- and .NET-originated logging messages are displayed on the same console output.

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 classes in the log4j package, and for loggerDemo.JavaClass. Start by launching JNBProxy, the GUI-based proxy generator, then selecting "Create new .NET → Java project" when the "Launch JNBProxy" form is displayed (Figure 1). After doing this, the main form of JNBProxy is displayed (Figure 2).

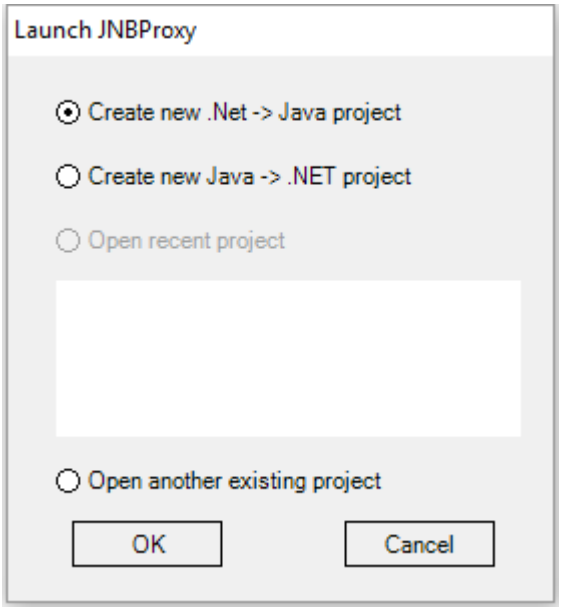

Figure 1. JNBProxy launch form

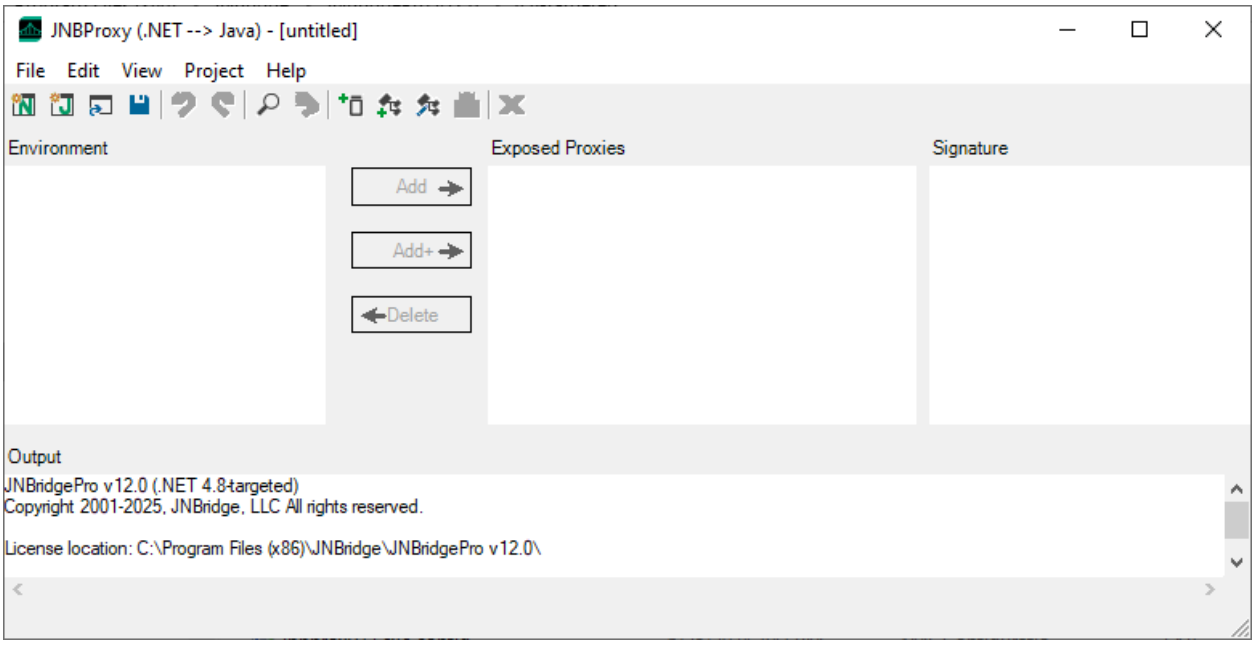

Figure 2. JNBProxy

Next, add the files log4j.jar and log4j-core.jar to the class path to be searched by JNBProxy. (You can download the log4j JAR files from http://jakarta.apache.org/log4j/docs/index.html.) Also add the folder in which the folder loggerDemo (which contains JavaClass.class) is 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 paths of the Jar and class files (Figure 3).

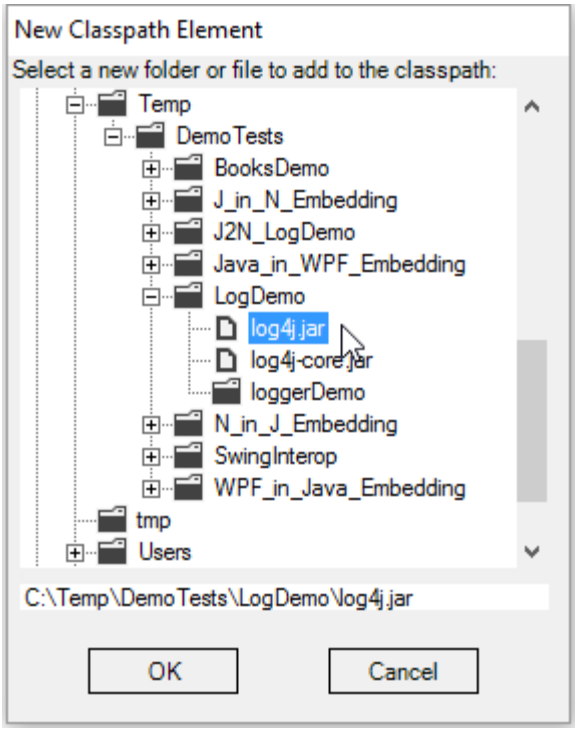

Figure 3. Adding a new classpath element

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.

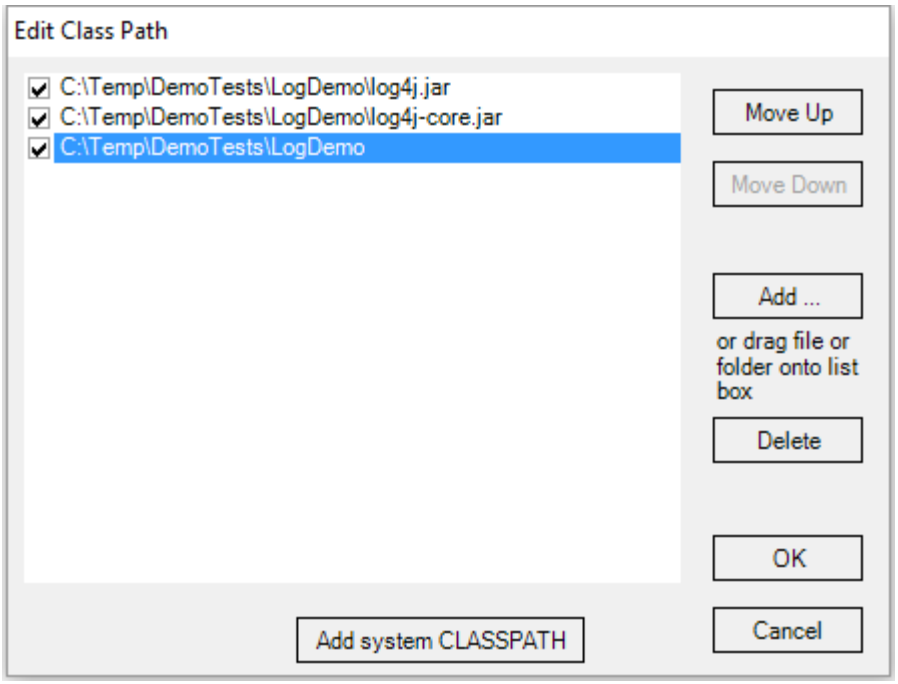

Figure 4. After creating classpath

The next step is to load the classes from each of the Jar files, and to add JavaClass. For the Jar files, use the menu command **Project→Add Classes from JAR File…** for each Jar file. For a single class such as JavaClass, use the menu command **Project→Add Classes from Classpath…** and enter the fully qualified class name loggerDemo.JavaClass (Figure 5).

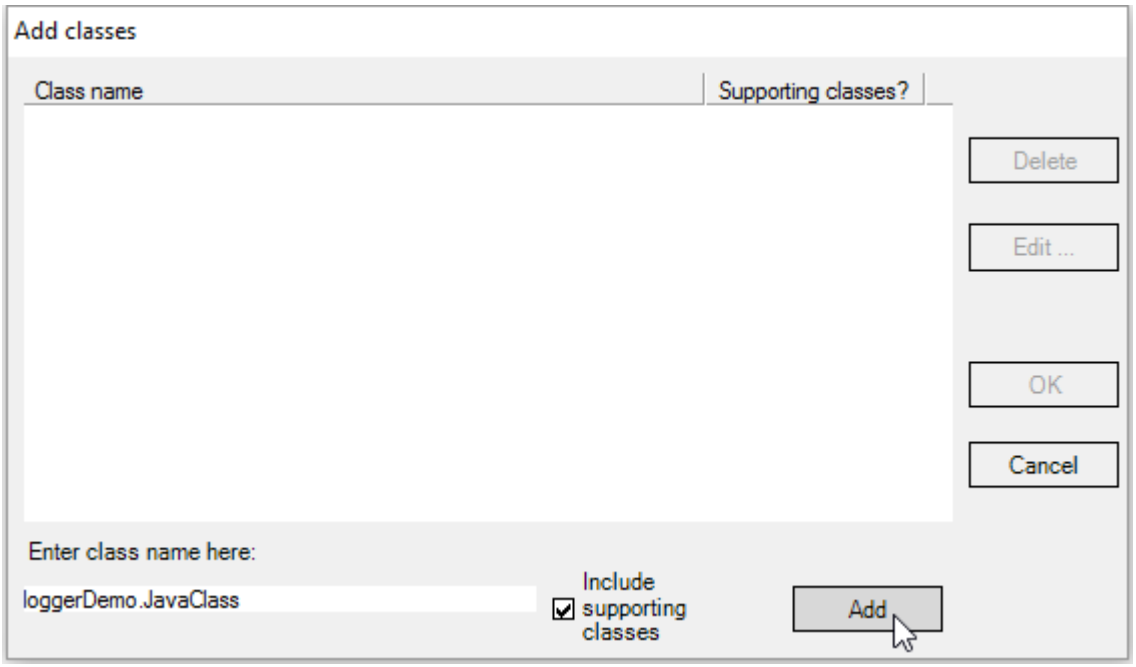

Figure 5. Adding a class from the classpath

Loading the classes may take a few minutes. Progress will be shown in the output pane in the bottom of the window, and in the progress bar. When completed, the classes in the log4j Jar files and loggerDemo.JavaClass will be displayed in the Environment pane on the upper left of the JNBProxy window (Figure 6). Note that JNBProxy will warn us that we are missing a number of classes relating to JMS (Java Messaging Service), XML, and JavaMail. Since we are not going to use these capabilities of log4j, we can safely ignore this warning.

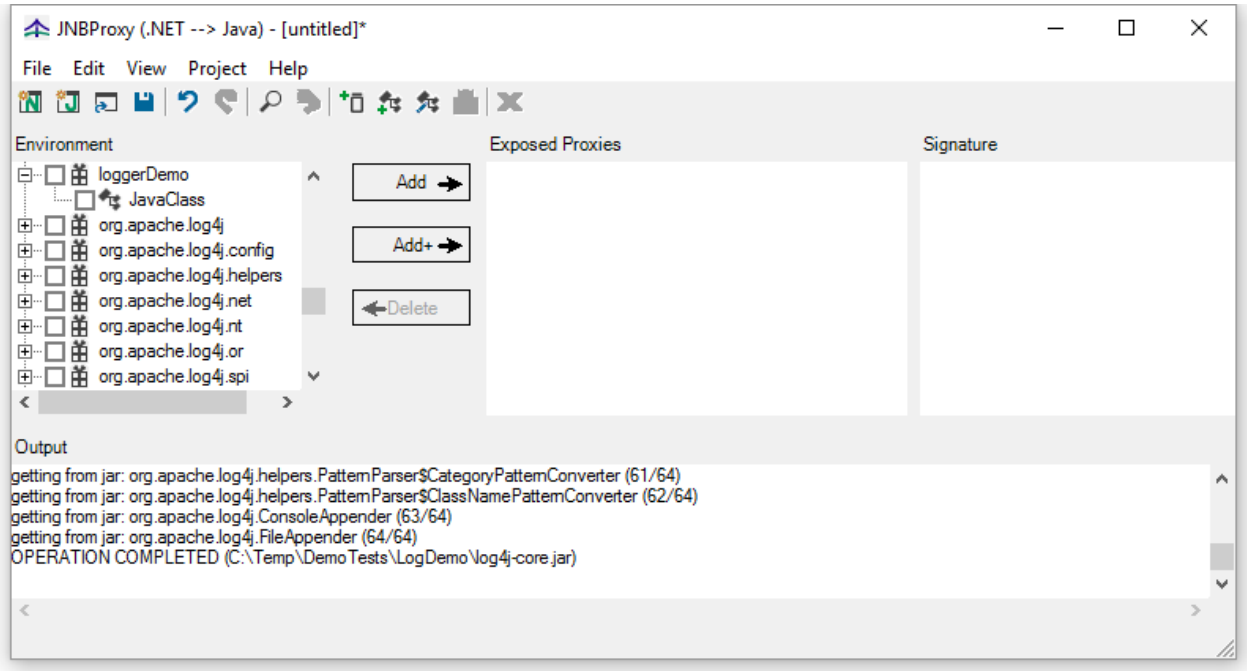

Figure 6. After adding classes

We wish to generate proxies for all 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).

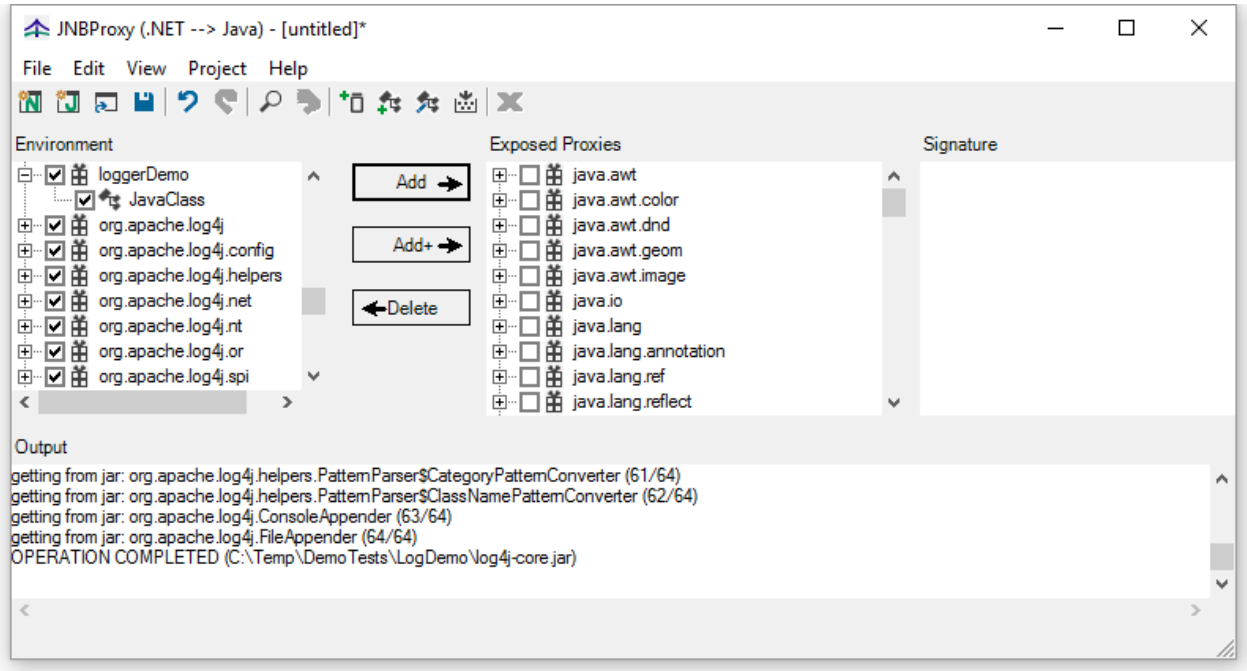

Figure 7. After adding classes to Exposed Proxies pane

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 logging.dll.

Using the proxies

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# console application project. Add references to the assemblies logging.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). When adding the jnbauth dll, make sure that its properties settings include "Copy always." 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. Next, add a new class and enter the following C# code:

```
using System;
using org.apache.log4j;
using java.lang;
using loggerDemo;

namespace com.jnbridge.demos.logger
{
```


```
    class LoggerDemo
    {
            static Category cat
                    = Category.getInstance("com.jnbridge.demos.logger.LoggerDemo");

            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            [STAThread]
            static void Main(string[] args)
            {
                    BasicConfigurator.configure();

                    cat.info(new JavaString("Entering application"));
                    DotNetClass dotNetClass = new DotNetClass();
                    JavaClass javaClass = new JavaClass();
                    for (int i = 0; i < 5; i++)
                    {
                            dotNetClass.f();
                            javaClass.doIt();
                    }
                    cat.info(new JavaString("Exiting application"));
            }
    }

    public class DotNetClass
    {
            static Category cat =
                    Category.getInstance("com.jnbridge.demos.logger.DotNetClass");

            public void f()
            {
                    cat.debug(new JavaString("Logged from .NET"));
            }
    }
}
```

Note that strings passed to the info() and debug() methods need to be wrapped in a java.lang.JavaString() object. This is because info() and debug() both expect a parameter of class java.lang.Object, and the .NET string is not a subclass of java.lang.Object, while java.lang.JavaString is. See the user's manual for more details.

The proxies for the Java objects in log4j 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 namespaces org.apache.log4j, java.lang, and loggerDemo, and afterwards can use the names of the Java classes.
- Proxies for the Java classes Category, BasicConfigurator, and JavaClass are used in exactly the same way as the original Java classes would have been used.
- The .NET class DotNetClass's calls to the logger object cat will cause messages to be written to the same output as the messages logged by JavaClass.
- When typing in the calls to the Java objects, Visual Studio's IntelliSense facility will offer to complete the names of method calls in the same way that it would for calls to .NET objects (Figure 8), and will provide information on number and types of parameters.

Figure 8. IntelliSense method completion for Java calls

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

Running the program

Running the program is simple. 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 jnbcore.jar, log4j.jar, log4j-core.jar, jnbcore_tcp.properties and loggerDemo\JavaClass.class are in the same folder, we can start up the Java-side in a console window as follows:

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

The above command line will start up a Java side without the SSL and class whitelisting security features. To use these security features with TCP/binary communications, see the sections "*Secure communications using SSL"* and "*Class whitelisting*," below.

In a separate console window, start up the .NET program. The Java console window will display logging messages originating on both the .NET and the Java side (Figure 9). Note that Figure 8(b) contains logging output that originated on both the Java and .NET sides.

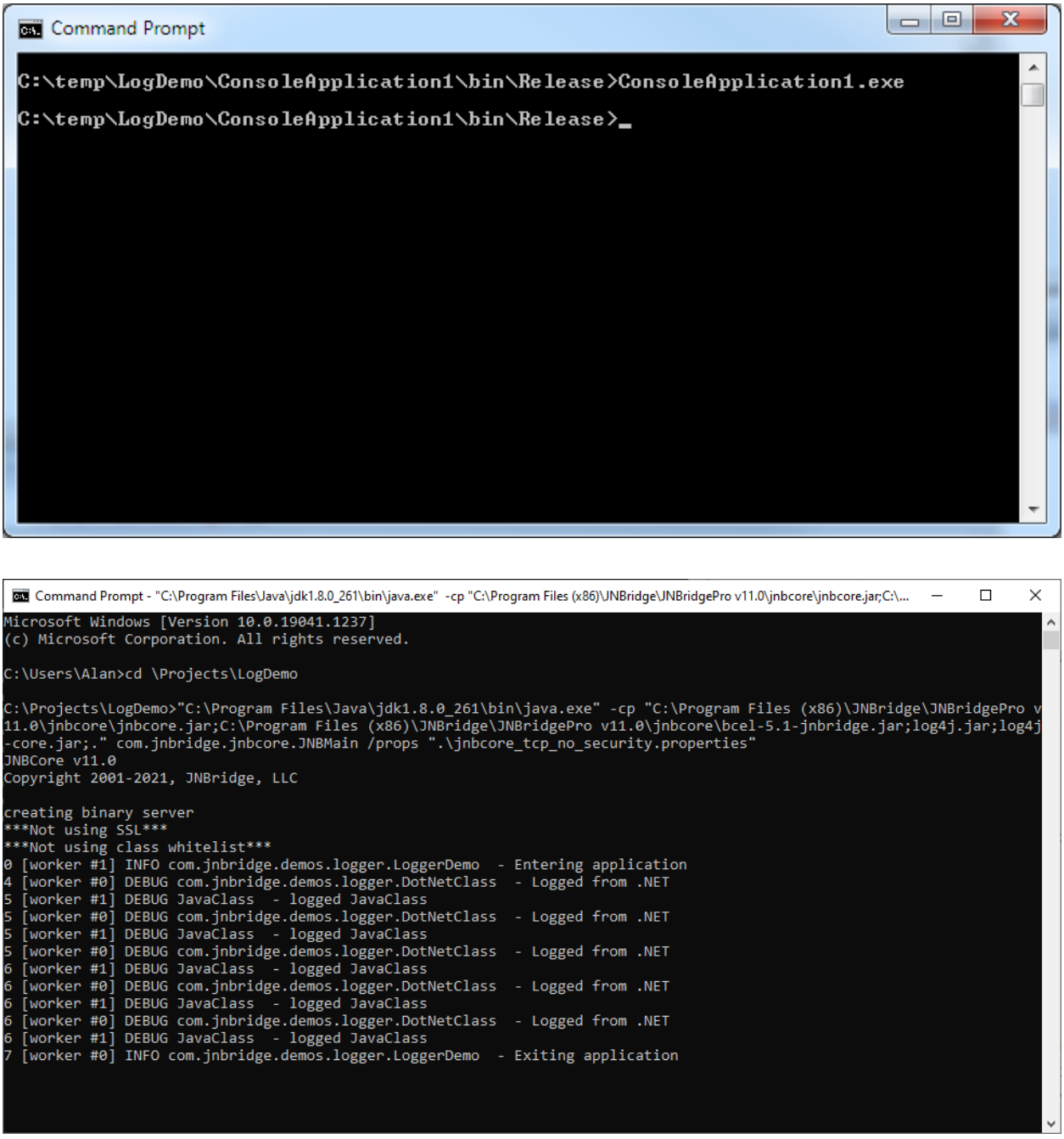

Figure 9. (a) Running the .NET-side. (b) Running the Java side.

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 mechanism, 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 jar files log4j.jar, log4j-core.jar, and the path to the folder containing the loggerDemo folder (which in turn contains JavaClass.class). 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.

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 log information from both Java and .NET using a common logging infrastructure. Such a strategy greatly simplifies development and debugging of code running on both Java and .NET platforms.

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.

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.