Demo: Calling a .NET Logging Package from Java

Version 10.1

jnbridge.com

JNBridge, LLC
jnbridge.com

COPYRIGHT © 2002–2019 JNBridge, LLC. All rights reserved.

JNBridge is a registered trademark and JNBridgePro and the JNBridge logo are trademarks of JNBridge, LLC.

Java is a registered trademark of Oracle and/or its affiliates. Microsoft, Visual Studio, and IntelliSense are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries. Apache is a trademark of The Apache Software Foundation.

All other marks are the property of their respective owners.

August 26, 2019

Introduction

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

In the example, JNBridgePro is used to allow Java code to call log4net, a .NET-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 Java classes with existing .NET classes that already use log4net 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 log4net 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 have a .NET-based driver method makes logging calls to log4net, and we will see how the Java-originated logging messages are displayed on the same console output by the .NET-based logging package.

Generating the proxies

While this example uses the standalone proxy generation tool, you can also use the Eclipse 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 log4net package, and for their supporting classes. Start by launching JNBProxy, the GUI-based proxy generator, then selecting "Create new Java → .NET project" when the "Launch JNBProxy" form is displayed (Figure 1). After doing this, the main form of JNBProxy is displayed (Figure 2).

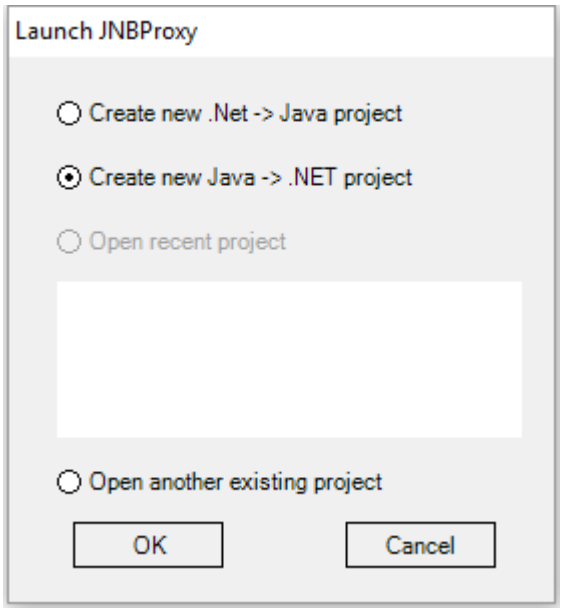

Figure 1. JNBProxy launch form

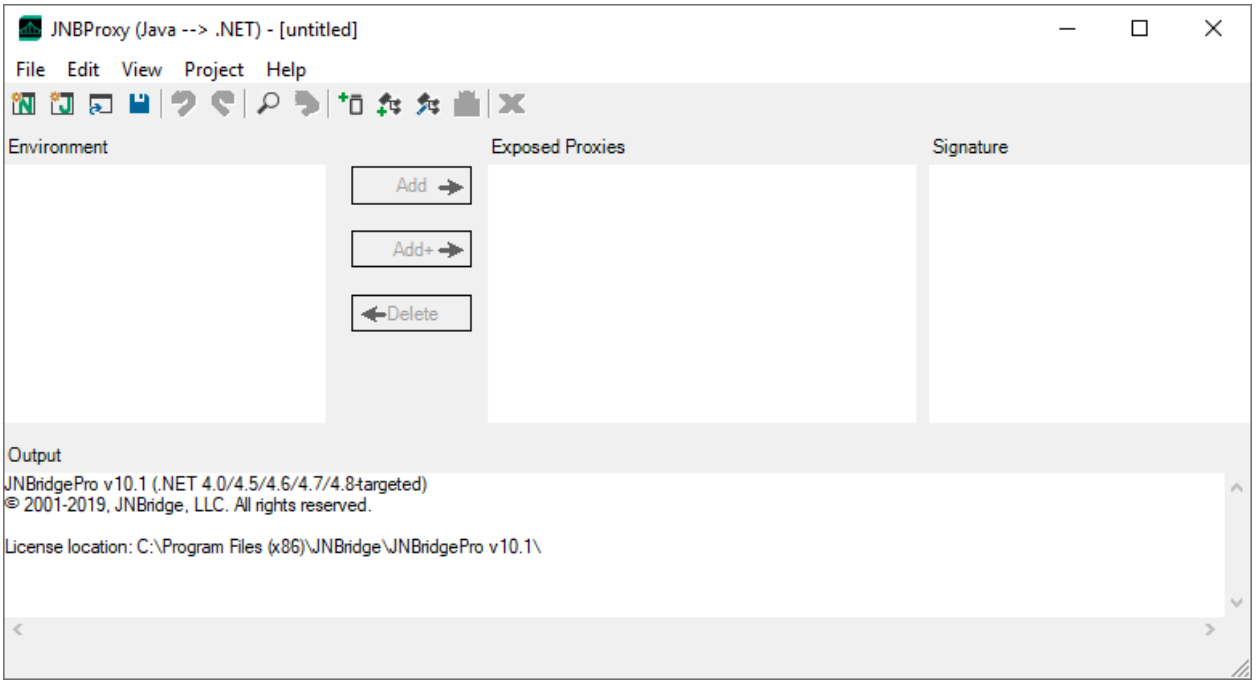

Figure 2. JNBProxy

Next, add the file log4net.dll to the assembly list path to be searched by JNBProxy. (We have supplied a copy of log4net with the demo.) Use the menu command **Project→Edit Assembly List…**. The **Edit Assembly List** 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).

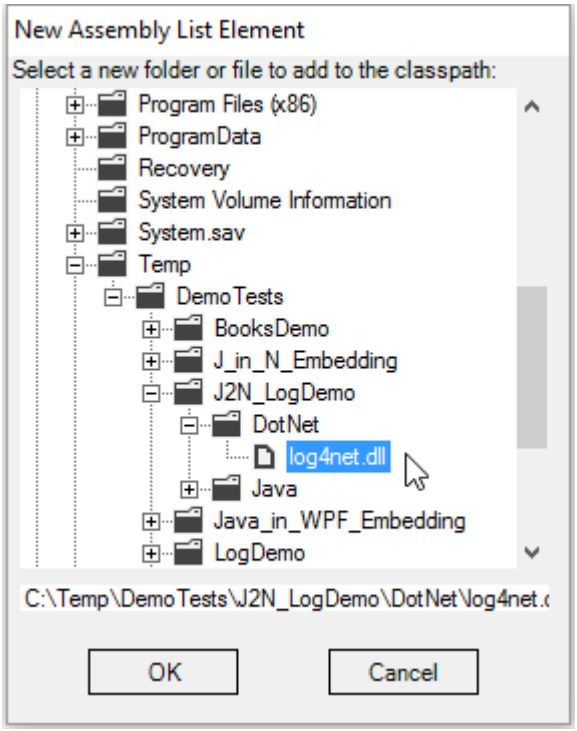

Figure 3. Adding a new classpath element

When this assembly list element is added, the **Edit Assembly List** dialog should contain information similar to that shown in Figure 4.

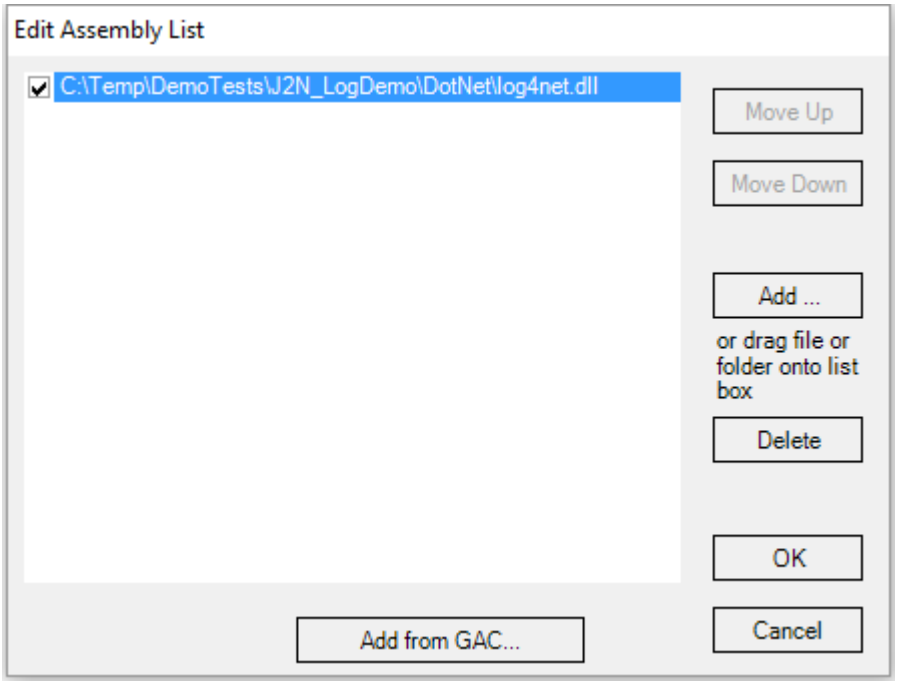

Figure 4. After creating classpath

The next step is to load the classes from log4net.dll. Use the menu command **Project→Add Classes from Assembly File…** for each DLL file. (Figure 5).

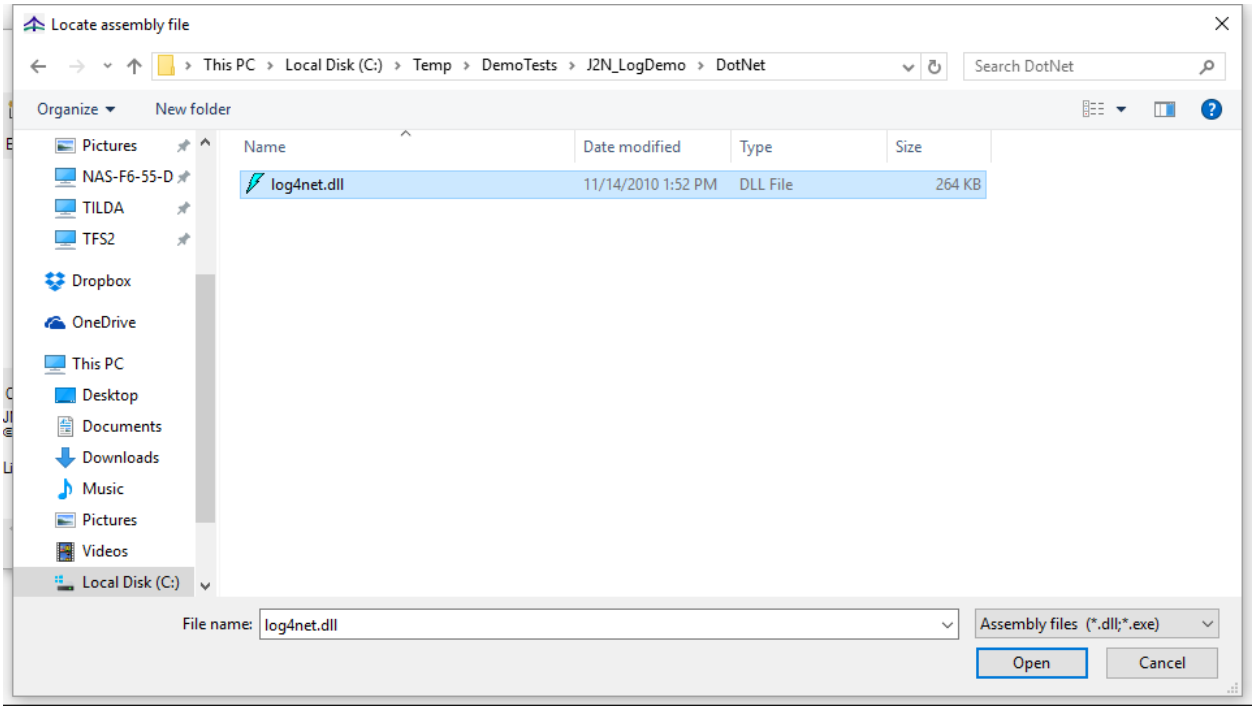

Figure 5. Adding classes from a DLL

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 log4net.dll files will be displayed in the Environment pane on the upper left of the JNBProxy window (Figure 6).

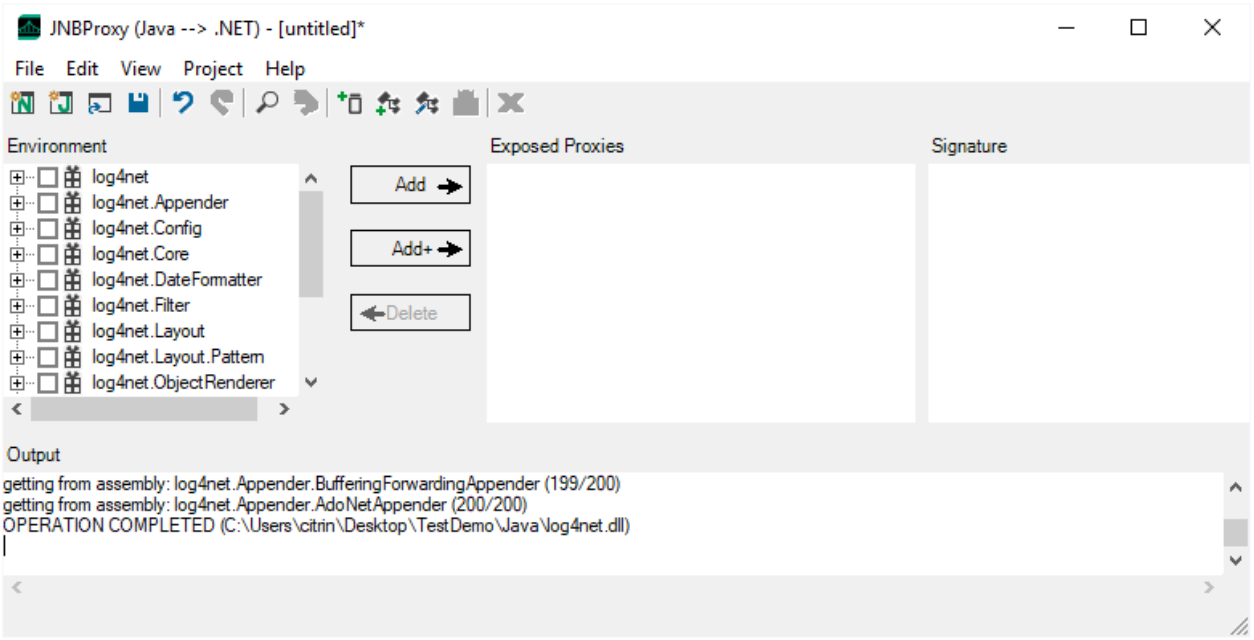

Figure 6. After adding classes

We wish to generate proxies for all these classes as well as their supporting 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, as well as other classes that might be used in connection with these classes (for example, parameter classes, return values, superclasses, and implemented interfaces) to the list of proxies to be exposed. These will be shown in the Exposed Proxies pane (Figure 7).

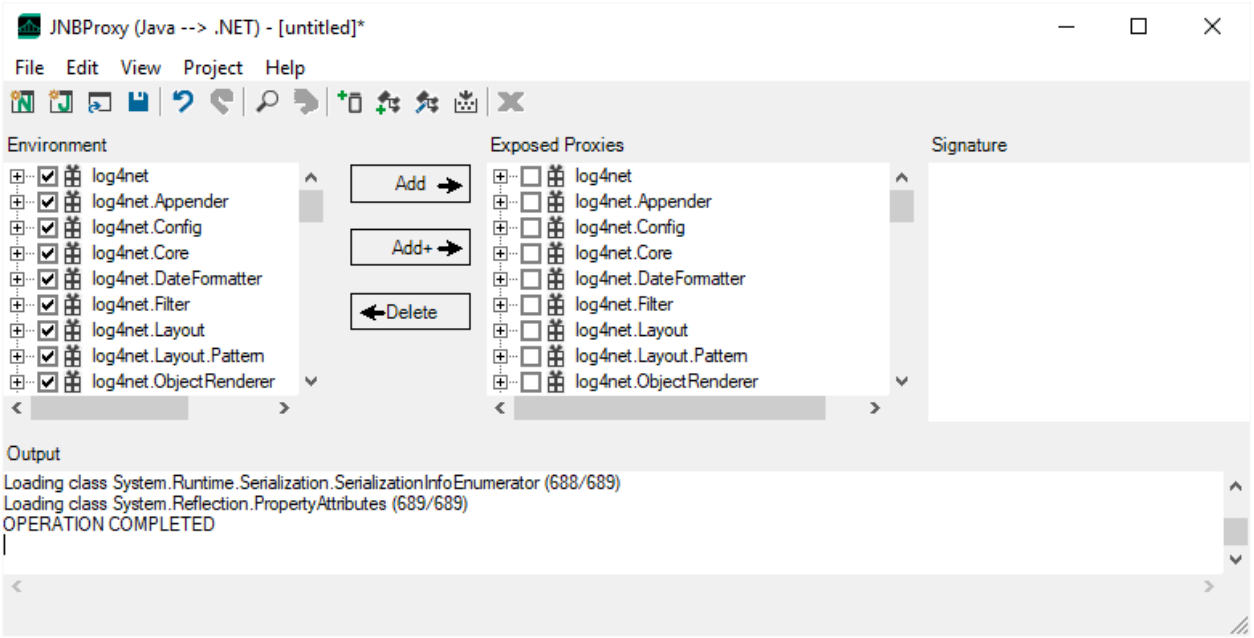

Figure 7. After adding classes to Exposed Proxies pane

We are now ready to generate the proxies. Select the **Project→Build...** menu command, and choose a name and location for the JAR (.jar) 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 proxies.jar.

Using the proxies

Now that the proxies have been generated, we can use them to access .NET classes from Java. We have supplied the Java file MainClass.java, which contains the Java program that drives the example. We have also provided the compiled MainClass.class file, as well as scripts (in .bat files) for compiling the code and for running it under various configurations.

Note that strings passed to the Error(), Warn() and Debug() methods need to be wrapped in a System.DotNetString() object. This is because Error(), Warn() and Debug() all expect a parameter of class System.Object, and the Java string is not a subclass of System.Object, while System.DotNetString is. See the user's manual for more details.

The proxies for the Java objects in log4net are used exactly as the original objects would be used in Java. Note the following items of interest:

- Proxies for the .NET classes have namespaces identical to the package names of the original Java classes. Thus, we simply import the namespace log4net, and afterwards can use the names of the log4net .NET classes. Proxies in the System namespace are a little more problematic, since there is also a Java class java.lang.System. Importing the actual class name (for example, System.IDisposable), allows us to just reference the proxy class by the name IDisposable in the Java code.
- Proxies for the .NET classes ILog and Configurator are used in exactly the same way as the original .NET classes would have been used. Since Java does not have the equivalent of the

.NET property, you access a property through a Get_ or Set_ method. For example, in the Java code we call log.Get_IsDebugEnabled() in order to access the underlying property log.IsDebugEnabled.

- The Java code's calls to the logger object log will cause messages to be written to the logging output in the console.
- Before the first proxy is called, the Java side must be configured by calling com.jnbridge.jnbcore.DotNetSide.init(). You can supply an argument giving the path to a properties file containing the necessary configuration information, or you can create a Java Properties object, add property information, and pass the Properties object as an argument to init(). The configuration properties specify the mechanism used to communicate between the Java and .NET sides, as well as certain parameters required by each communication mechanism. We have provided two properties files, one for shared-memory communication and one for tcp/binary communication. We recommend that you examine these properties files to see how configuration is done. Details of the configuration information in the properties files can be found in the *Users' Guide*.

After entering the code, build the project to obtain the executable, using buildJava.bat.

Running the program

The following instructions show how to run the demo using TCP/binary communications without the security features (class whitelisting and SSL) enabled. Subsequent sections show how to run the demo using those features.

Running the program is simple. Make sure that JNBridgePro is properly configured on the Java side (i.e., there is a properties file with the proper Java-side configuration information – see the file tcp_binary_no_security.properties that we have supplied for an example of Java-side configuration information) and on the .NET side (i.e., that there is a copy of JNBDotNetSide.exe.config containing the appropriate .NET-side configuration information in the same folder as JNBDotNetSide.exe), and that the .NET and Java side configurations agree on the protocol and port to be used. Then, start up the .NET side. Assuming that JNBDotNetSide.exe, JNBDotNetSide.exe.config, JNBShare.dll, JNBSharedMem_x86.dll (or JNBSharedMem_x64.dll, or both) and jnbauth_x86.dll (or jnbauth_x64.dll or both) are in the same folder (or JNBShare.dll and JNBSharedMem.dll are installed in the GAC), simply launch JNBDotNetSide.exe. Also note that it is possibly to write your own code to start up your own .NET side using the DotNetSide.startDotNetSide() and DotNetSide.stopDotNetSide() APIs – please see the *Users' Guide* for more information.

In a separate console window, start up the Java using the script runJava_tcp_binary_no_security.bat. Please examine the contents of the .bat file to see how the classpath is constructed and how the properties information is supplied on the command line. Also note that you may need to edit the .bat file to reflect the installation location of JNBridgePro on your machine, and the location of java.exe.

The .NET console window will display logging messages resulting from the calls from Java code to the log4net API (Figure 8).

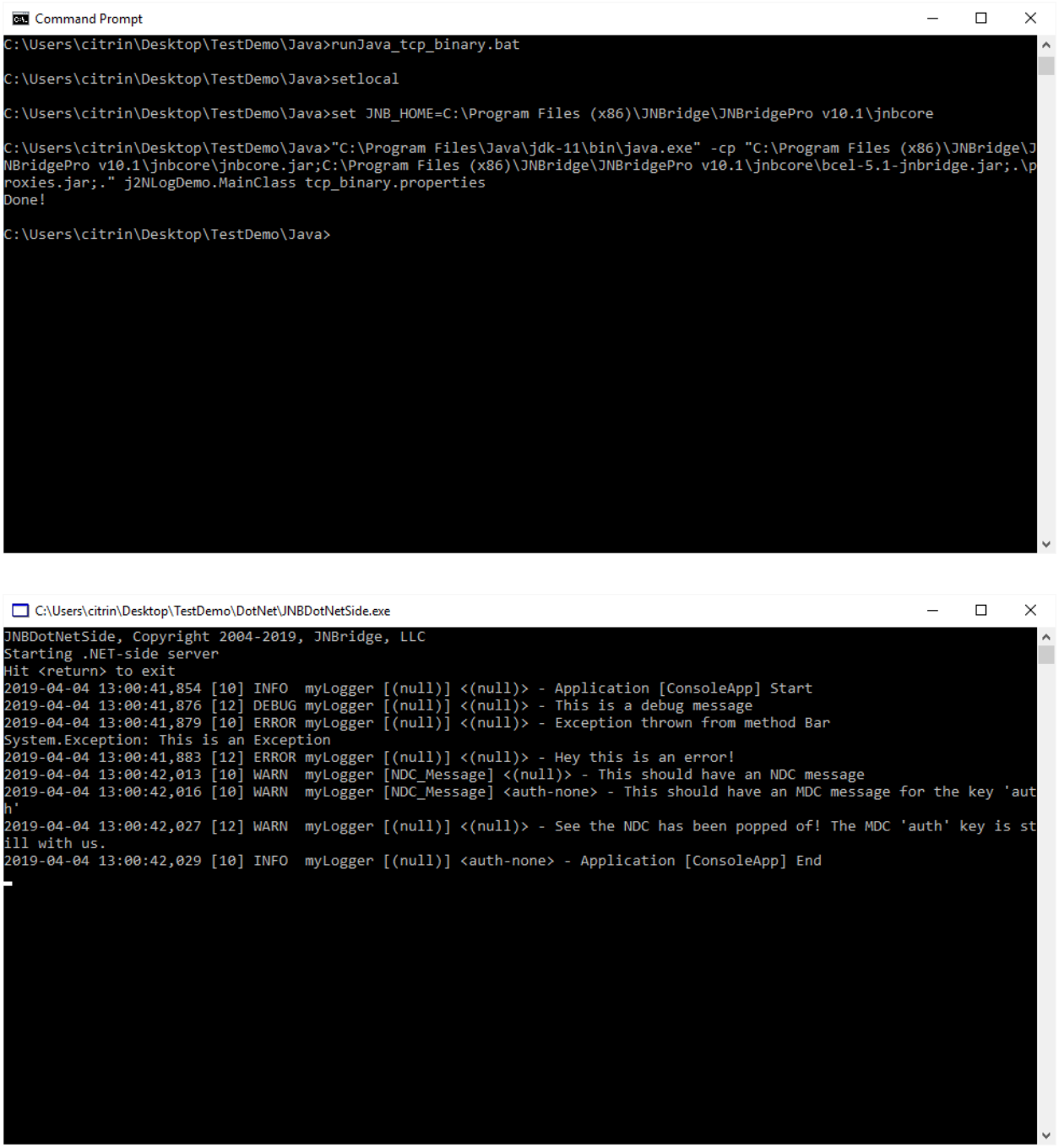

Figure 8. (a) Running the Java side. (b) Running the .NET side.

Class whitelisting: When using TCP/binary communications, the .NET side can be configured to only allow requests from the Java side that reference specific .NET classes. This prevents the possibility of malicious clients accessing sensitive APIs (which need not even have been proxied). The file JNBDotNetSide.exe.config contains a variant of <javaToDotNetConfig> that uses class whitelisting. It contains the element useClassWhiteList=”true”. This is the default value and may be omitted. To turn off class whitelisting, the element must be explicitly set to false.

<javaToDotNetConfig> also contains an element classWhiteListFile=".\classWhiteList.txt". The element above is the path to a text file each of whose lines is a class that can be accessed from the Java side. If the Java 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 Java side:

log4net.ILog
log4net.LogManager
log4net.Config.XmlConfigurator
System.IO.FileInfo
System.IDisposable
log4net.NDC
log4net.MDC

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

To use class whitelisting (and SSL) in the demo, uncomment the variant of <javaToDotNetConfig> in JNBDotNetSide.exe.config that uses the security features, comment out the variant that does not use the security features.

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

Secure communications 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. Use the supplied Java-side properties file tcp_binary_with_security.properties. See the *Users' Guide* for a discussion of the meanings of the various additional properties. You may need to edit the paths in the javaSide.trustStore and javaSide.keyStore properties.

On the .NET side, in the JNBDotNetSide.exe.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
- *serverCertificateLocation* – this is the path to the .NET side's server certificate, and is used to authenticate itself to the server side, and also for encryption. This is a .p12 or .pfx file, and as such should contain both the server's public and private keys.

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 (mytestclient).

- *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 (dotnetside).
- *javaSide.trustStorePassword* – this is the password of the truststore file.

Since the Java-side server certificate (in this case, myTestClient.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.

In addition, on the .NET-side machine, install the version of the server certificate that contains the public/private key pair (dotnetside.p12) in the Windows certificate store using the instructions above. (You will need to supply the password *changeit* in this case.)

Once you have done all of this, start the .NET side (JNBDotNetSide.exe), then run runJava_tcp_binary_with_security.bat to start the Java side.

Using shared-memory communication. It is possible to run the .NET side in the same process as the Java 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 .NET 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 run runJava_sharedmem.bat. This script uses sharedmem.properties to configure the Java and .NET sides. We recommend that you open this file and examine the information that is supplied inside. Please note that you may need to edit the file to change the javaEntry and appBase properties, if the location of the JNBridgePro installation is different on your machine.

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 Java program can call a .NET-based API.

Creating this program was accomplished in three stages:

- In the first stage, proxies were generated allowing access by Java classes to the .NET classes. The proxies were generated using JNBProxy, a visual tool that allows developers a wide variety of strategies for determining which .NET classes are to be exposed to access by Java.
- In the second stage, the Java .jar file containing the proxies was incorporated into a Java project's build classpath, and Java code was written that accessed the proxies. Java classes can access .NET classes transparently, as if the .NET classes had themselves been written in Java. Nothing special or additional needs to be done to manage Java-.NET communications or object lifecycles.
- In the third stage, the integrated Java and .NET code is run. All that is required is to start a .NET-side containing the .NET code to be accessed. Once the .NET-side is started, the user simply runs the Java program that will access the Java objects. One can also use shared-memory communications to run a .NET side automatically embedded inside the Java process, in which case it is not necessary to explicitly start the .NET side.

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