Example: Calling a Java Logging Package from .NET Core

Version 10.1

jnbridge.com

JNBridge, LLC
jnbridge.com

September 6, 2019

Introduction

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

This example is based on the "log demo" example that is distributed with the JNBridgePro installation. That example is targeted toward .NET Framework running on Windows. When working through the example, please use the files supplied with that installation. This document will show how to work through the entire example, and details that differ when working with .NET Core will be annotated. We will show how to configure, deploy, and run the example on both Windows and on Linux, using both TCP/binary and shared memory communications.

Generating the proxies

> **.NET Core-specific information:**
> There is no proxy generation tool targeted toward .NET Core that generates proxy DLLs. Proxy DLLs for use with .NET Core must be generated with one of the .NET Framework-targeted proxy generation tools (standalone GUI-based or command-line, or the Visual Studio plugin) running on Windows using .NET Framework. We apologize for any inconvenience this might present, and hope to rectify this problem in a future release.

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).

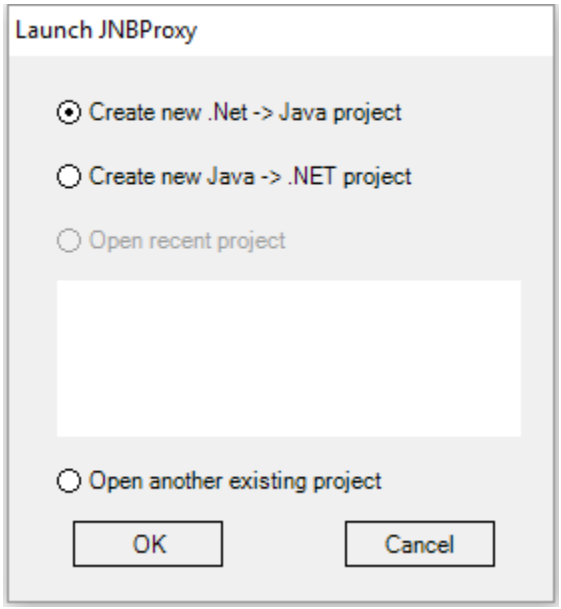


Figure 1. JNBProxy launch form

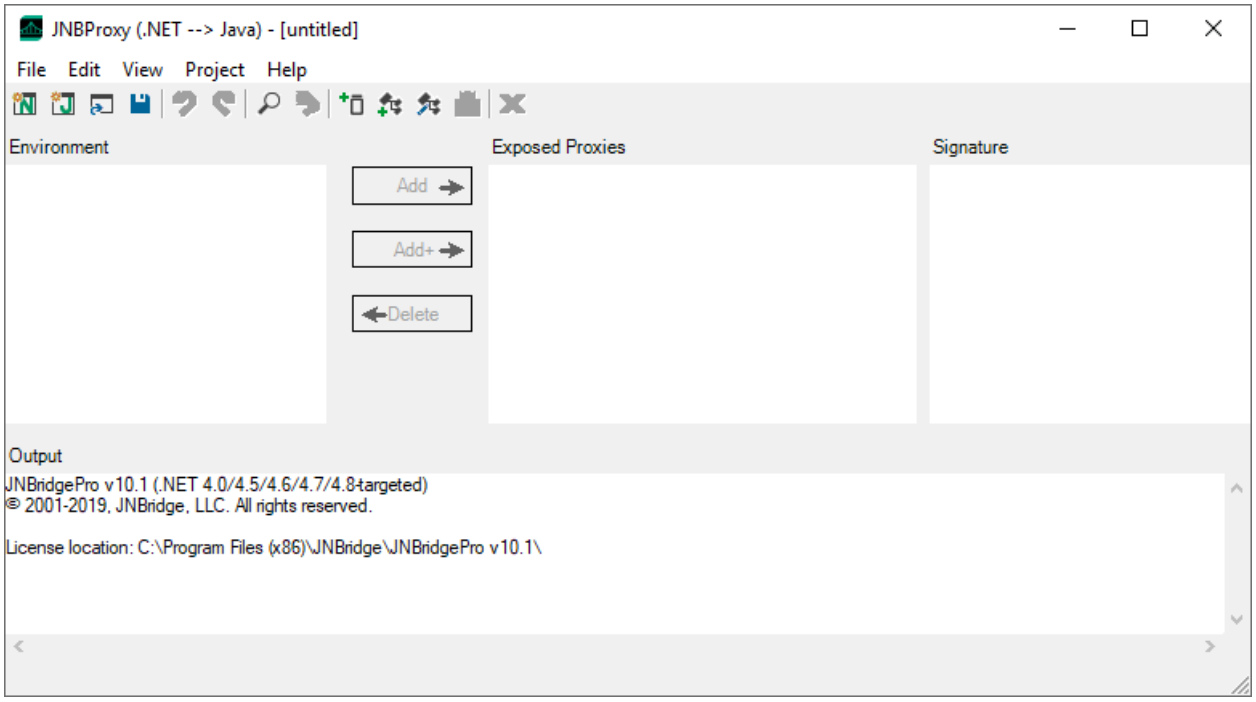


Figure 2. JNBProxy

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).

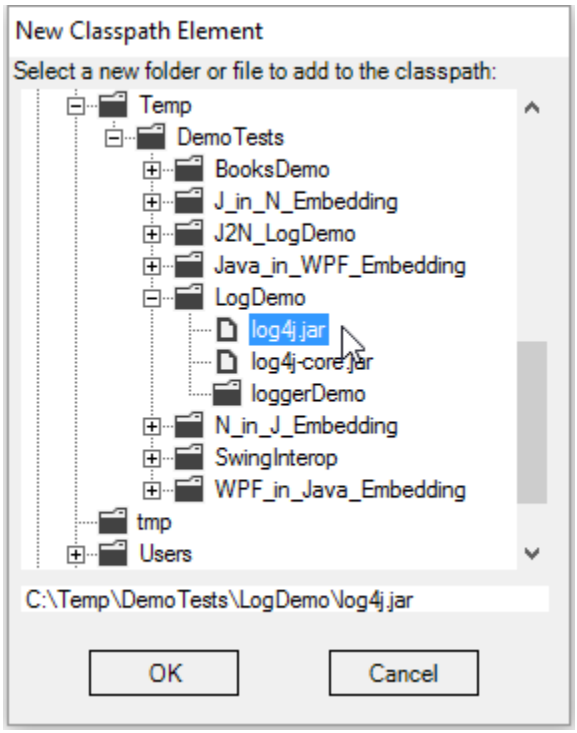


Figure 3. Adding a new classpath element

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.

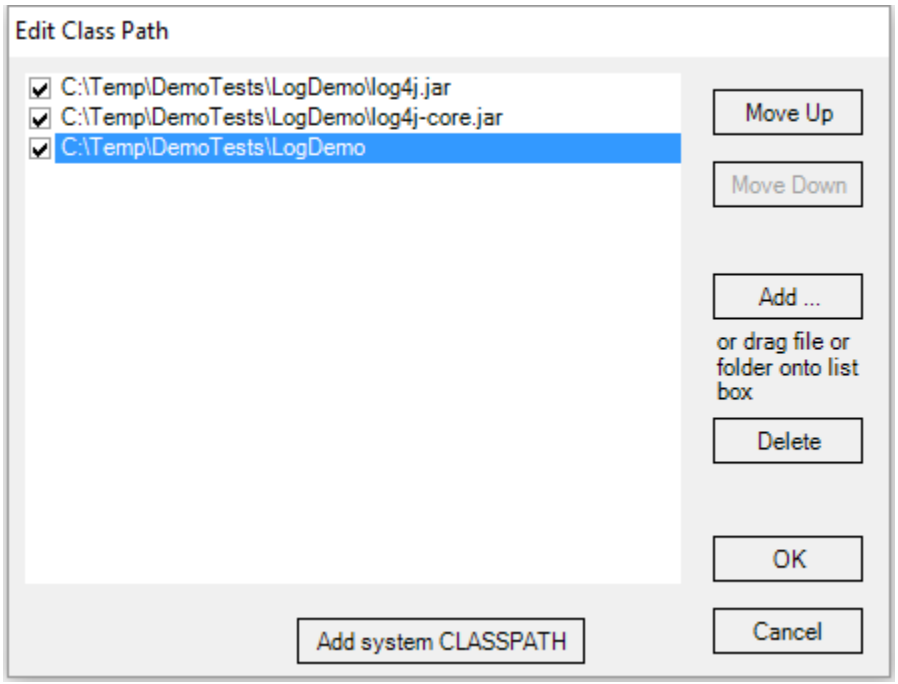


Figure 4. After creating classpath

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).

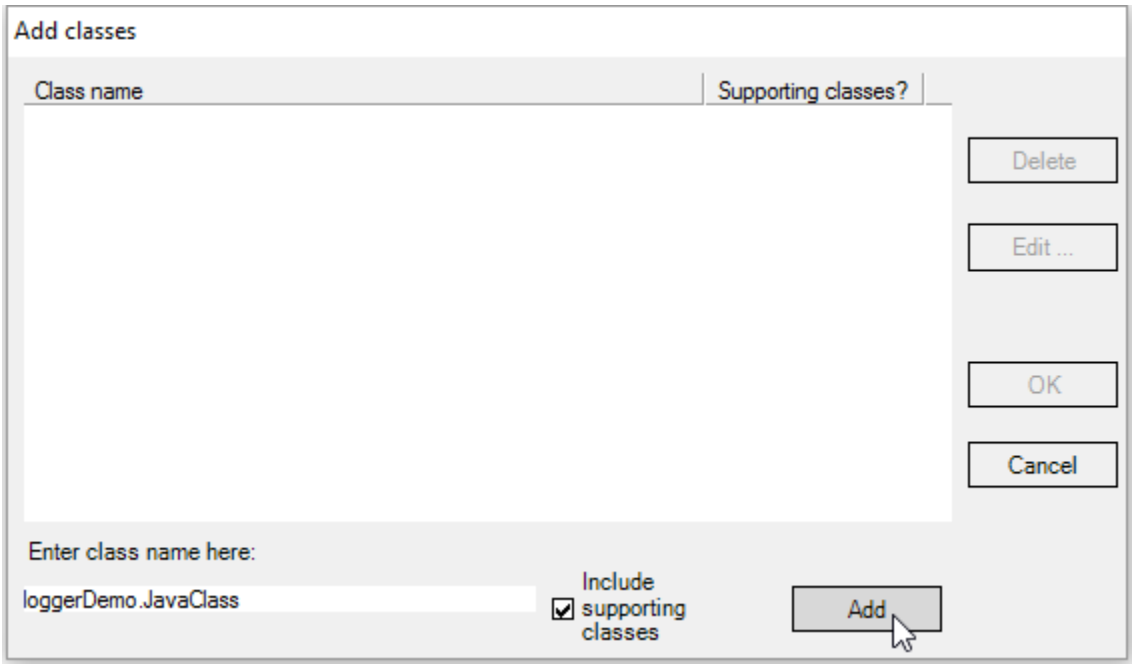


Figure 5. Adding a class from the classpath

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.

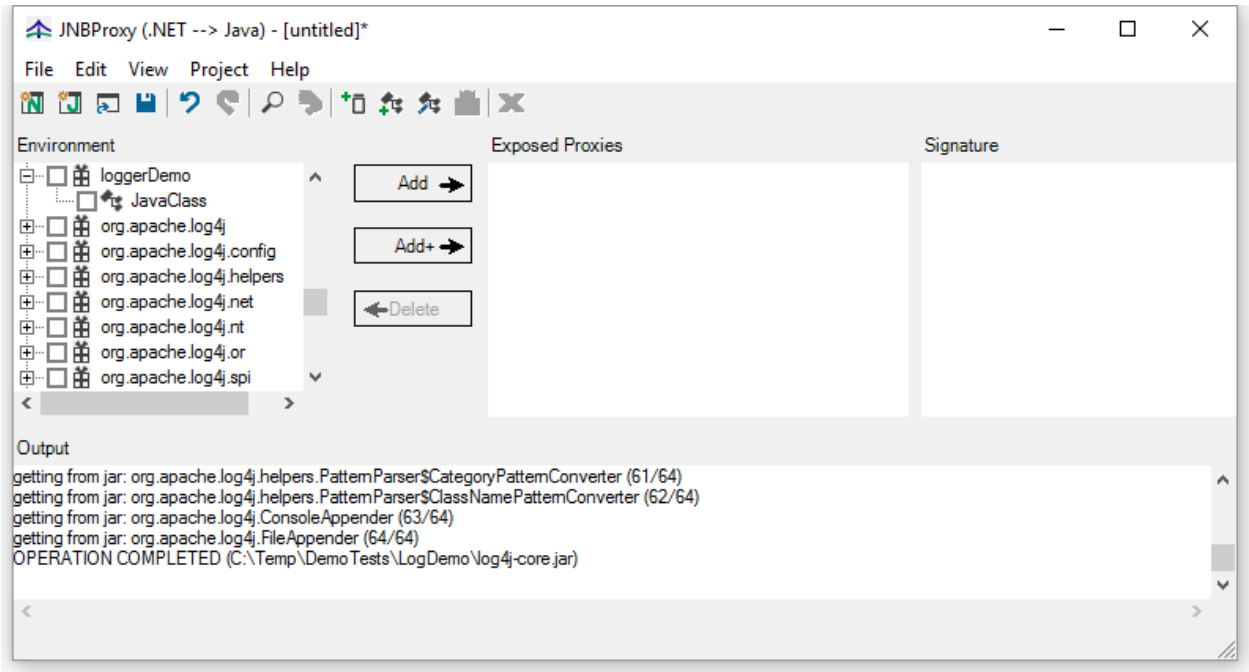


Figure 6. After adding classes

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).

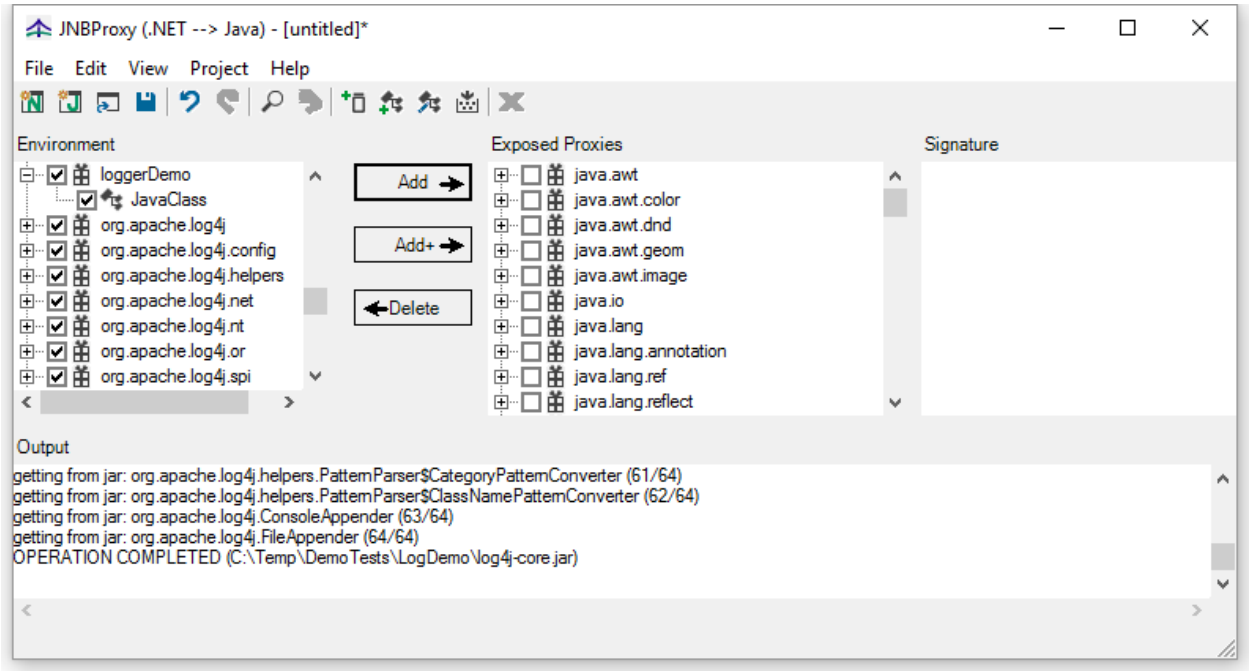


Figure 7. After adding classes to Exposed Proxies pane

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 (with Windows)

> **.NET Core-specific information:**
> Elements of the example that differ when using .NET Core are marked as this paragraph is.
>
> Now that the proxies have been generated, we can use them to access Java classes from .NET Core. Launch Visual Studio 2019 (making sure that you have .NET Core 3.0 installed), and create a new .NET Core C# console application project. Add references to the assemblies logging.dll (the one just generated) and the .NET Core-targeted jnbshare.dll (distributed with JNBridgePro). Add to your project the file jnbauth_x86.dll or jnbauth_x64.dll or both, and JNBSharedMem_x86.dll or JNBSharedMem_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 and JNBSharedMem dlls, make sure that their properties settings include “Copy always.”
>
> In addition, add the following extension DLLs, included in the JNBridgePro installation, to the project:
>
> - Microsoft.Extensions.Configuration.dll
> - Microsoft.Extensions.Configuration.Abstractions.dll

- Microsoft.Extensions.Configuration.Binder.dll
- Microsoft.Extensions.Configuration.FileExtensions.dll
- Microsoft.Extensions.Configuration.Json.dll
- Microsoft.Extensions.FileProviders.Abstractions.dll
- Microsoft.Extensions.FileProviders.Physical.dll
- Microsoft.Extensions.FileSystemGlobbing.dll
- Microsoft.Extensions.Primitives.dll
- Newtonsoft.Json.dll
- System.Runtime.CompilerServices.Unsafe.dll

As with the jnbauth and JNBSharedMem DLLs, make sure that the properties settings for the files are set to “Copy always.”

Add an empty file named jnbridgeConfig.json to the project. It is JSON configuration file that configures the .NET side of JNBridgePro. Note that this differs from JNBridgePro for .NET Framework, as it uses a JNBridgePro-specific JSON file for configuration rather than an application-wide XML-based configuration file, as is used in .NET Framework projects. As with the jnbauth and JNBSharedMem DLLs, make sure that the properties setting for jnbridgeConfig.json is set to “Copy always.”

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.

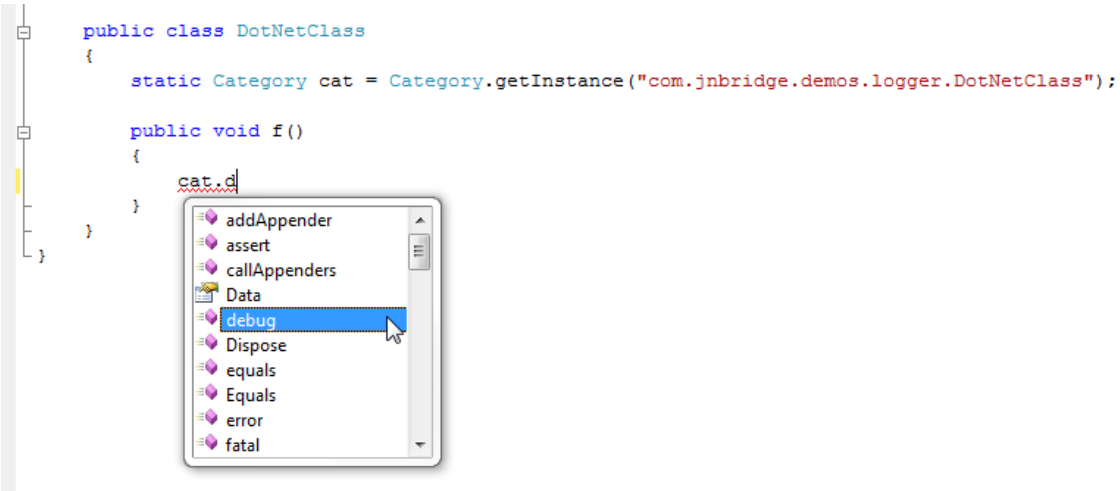


Figure 8. IntelliSense method completion for Java calls

Finally, open the file jnbridgeConfig.json and add the following text:

```
{
   "dotNetToJavaConfig": {
      "scheme": "jtcp",
      "host": "localhost",
      "port": 8085,
      "useSSL": "false"
   },
  "licenseLocation": {
      "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v10.1"
   }
}
```

Note the following elements:

- The “dotNetToJavaConfig” section specifies that TCP/binary communications should be used, and that the .NET Core side will be communicating with a Java side on the same machine (“localhost”) which will be listening on port 8085.
- The JNBridgePro license file that will be used by this application is installed in C:\Program Files (x86)\JNBridge\JNBridgePro v10.1 (the JNBridgePro installation folder). Note that this location might be different on your machine. Also note that the path contains backslashes (‘\’) that must be escaped as double-backslashes (‘\\’).
- This example does not use available security features such as SSL and class whitelisting. For information on using the security features, see the *Users’ Guide*.

After entering the code, build the project to obtain the executable, which, since this is .NET Core, will be a DLL file.

Running the program (with Windows)

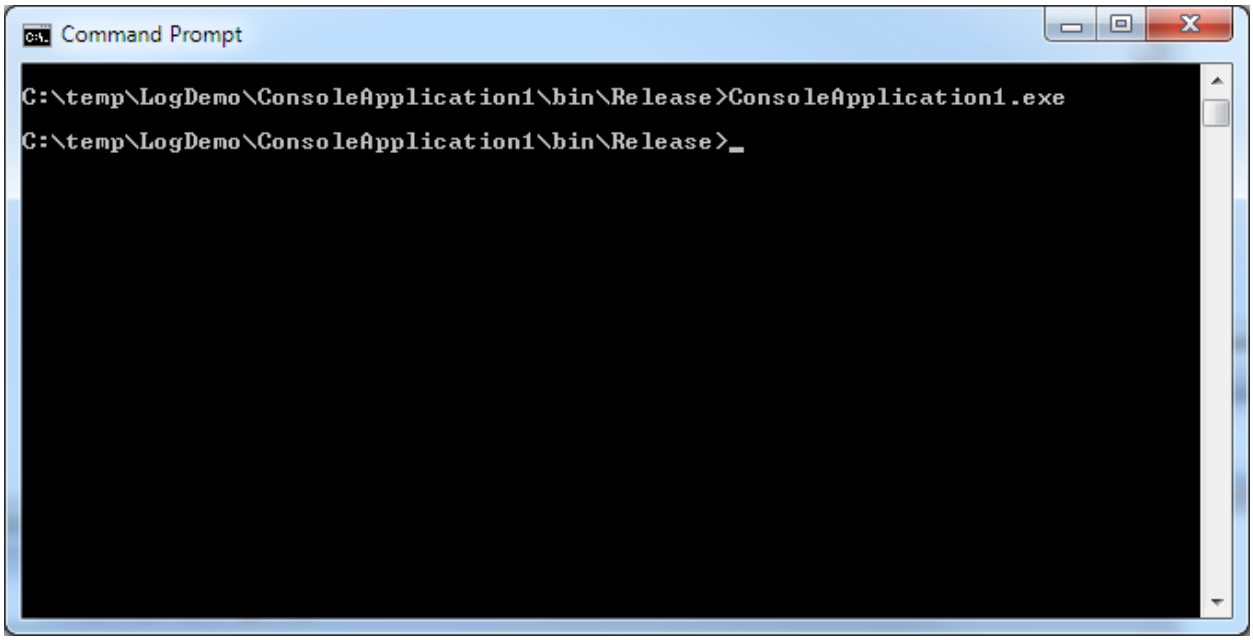
Running the program is simple. Make sure that JNBridgePro is properly configured on the .NET side (i.e., jnbridgeConfig.json has been added to the project – upon building the solution, jnbridgeConfig.json will be copied to your project's build folder) and on the Java side (i.e., that there is a copy 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. Also, make sure that the properties `javaSide.useSSL=false` and `javaSide.useClassWhiteList=false` are included in the jnbcore.properties file. (Security features like SSL and class whitelisting are not being used in this example. To use them, see the *Users'* Guide.) 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.properties
```

In a separate console window, start up the .NET program by running the following command-line (assuming the application was built as logDemo.dll):

dotnet logDemo.dll

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.



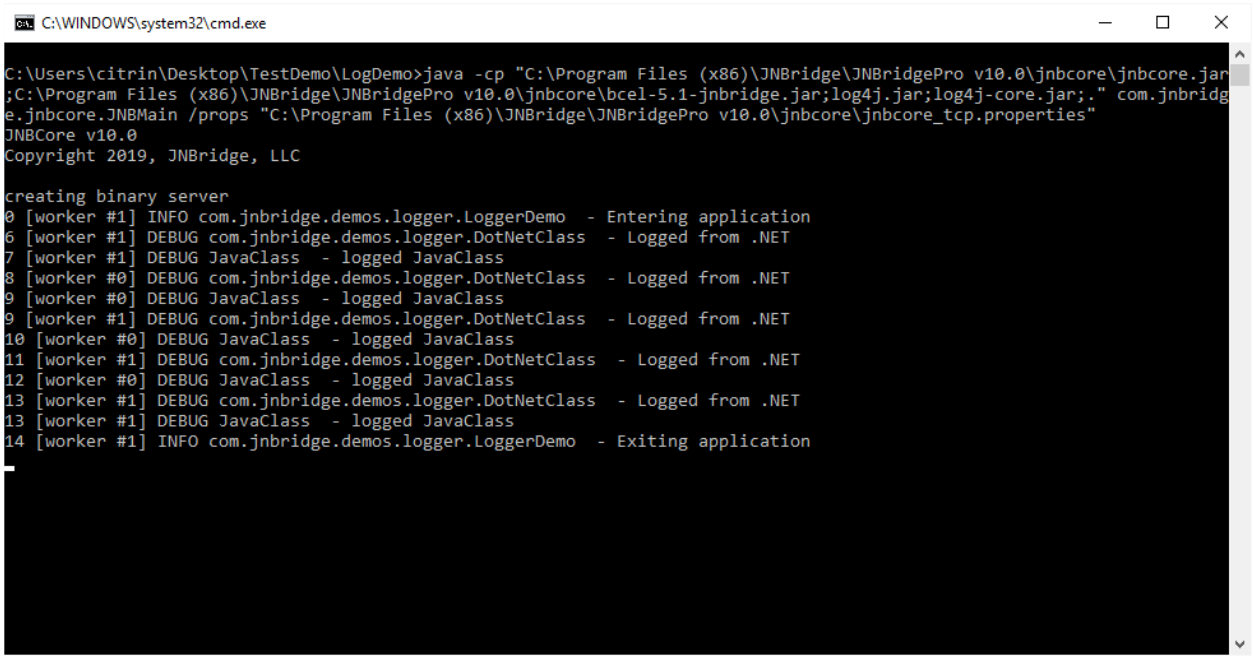


Figure 9. (a) Running the .NET-side. (b) Running the Java side.

> ***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 (HTTP/SOAP is not supported when using .NET Core), 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 jnbridgeConfig.json configuration file and alter it as follows:
>
> ```
> {
>    "dotNetToJavaConfig": {
>       "scheme": "sharedmem",
>       "jvm32": "C:\\Program Files
> (x86)\\Java\\jre1.8.0_111\\bin\\client\\jvm.dll",
>       "jvm64": "C:\\Program Files\\Java\\jre-10.0.1\\bin\\server\\jvm.dll",
>       "jnbcore": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro
> v10.1\\jnbcore\\jnbcore.jar",
>       "bcel": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro
> v10.1\\jnbcore\\bcel-5.1-jnbridge.jar",
>       "classpath": "D:\\DotNet Core examples\\LogDemo\\log4j.jar;D:\\DotNet
> Core examples\\LogDemo\\log4j-core.jar;D:\\DotNet Core examples\\LogDemo"
>    },
>    "licenseLocation": {
>       "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v10.1"
>    }
> }
> ```
>
> Note that the "dotNetToJavaConfig" element now specifies that shared memory should be used. The element specifies the 32-bit or 64-bit jvm.dll that should be used, depending on whether the application is running as 32-bit or 64-bit. (If you know that you will never run the application

as 32-bit, you can leave out the “jvm32” element; similarly, if it will never run as 64-bit, you can leave out the “jvm64” element.) The “jnbcore” and “bcel” elements point to the locations of jnbcore.jar and bcel-5.1-jnbridge.jar on your machine. The “classpath” element specifies the Java-side classpath as a semicolon-separated list of paths. As before, the JSON file also specifies the path to the folder containing the license file. Also as before, all paths must use double-backslashes (‘\\’).

Once you have made the changes, build the project and start it (assuming the application was built as logDemo.dll):

dotnet logDemo.dll

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.

Using the proxies (with Linux)

Running the application with .NET Core on Linux is similar to running it on Windows. In this section, we cover the differences. Where not mentioned, please follow the instructions previously given using Windows.

Build the application on Windows using Visual Studio (alternatively, you can build and deploy the application on Windows using Visual Studio Code). Instead of (or in addition to) the jnbauth and JNBSharedMem dlls, include their Linux equivalents: jnbauth_x64.so and libJNBSharedMem_x64.so. (JNBridgePro for .NET Core only supports 64-bit .NET Core on Linux.) Note that, if you want your application to run on both Windows and Linux, include both the DLLs and the equivalent .so files: the appropriate ones will be automatically loaded, depending on the operating system on which the application is running.

When using TCP/binary, the .NET Core-side configuration file jnbridgeConfig.json is similar to the one used on Windows – the only difference is likely to be in the license location:

```
{
   "dotNetToJavaConfig": {
      "scheme": "jtcp",
      "host": "localhost",
      "port": 8085,
      "useSSL": "false",
   },
  "licenseLocation": {
      "directory": "/home/myAccount/logDemo"
   }
}
```

Note that the directory path uses single forward slashes (‘/’) and that the path is case sensitive.

Running the program (with Linux)

Running the application on Linux is similar to running it on Windows. Deploy the application, including all DLLs and .so files, as described above, along with the JSON configuration file,

jnbcore.jar, bcel-5.1-jnbridge.jar, the Java-side properties file containing the configuration, and your Java binaries.

Start up the Java side as usual:

```
java –cp ".:log4j.jar:log4j-core.jar:jnbcore.jar:bcel-5.1-jnbridge.jar"
com.jnbridge.jnbcore.JNBMain /props jnbcore.properties
```

Note that the classpath is colon-separated rather than semicolon-separated, as in Windows. Also note that all paths in the classpath (and to the properties file) must be correct absolute or relative paths.

Then, run the logDemo DLL:

dotnet logDemo.dll

(assuming the dotnet command is in your path).

Using shared-memory communication. It is possible on Linux 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 (HTTP/SOAP is not supported when using .NET Core), 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 jnbridgeConfig.json configuration file and alter it as follows:

```
{
   "dotNetToJavaConfig": {
      "scheme": "sharedmem",
      "jvm64": "/usr/lib/jvm/java-11-openjdk-
amd64/jre/lib/amd64/server/libjvm.so",
      "jnbcore": "./jnbcore.jar",
      "bcel": "./bcel-5.1-jnbridge.jar",
      "classpath": "./LogDemo/log4j.jar:./LogDemo/log4j-core.jar:./LogDemo"
   },
   "licenseLocation": {
      "directory": "/home/myAccount/logDemo"
   }
}
```

Note that the "dotNetToJavaConfig" element now specifies that shared memory should be used. The element specifies the 64-bit libjvm.so (the equivalent to jvm.dll on Windows) that should be used. (32-bit processes are not supported on Linux using JNBridgePro; only 64-bit is supported on Linux.) The "jnbcore" and "bcel" elements point to the locations of jnbcore.jar and bcel-5.1-jnbridge.jar on your machine. The "classpath" element specifies the Java-side classpath as a semicolon-separated list of paths. As before, the JSON file also specifies the path to the folder containing the license file. Also as before, all paths must use single forward slashes ('/'), are case sensitive, and must be correct absolute or relative paths.

Once you have made the changes, build the project and start it (assuming the application was built as logDemo.dll, and that the dotnet command is in your path):

dotnet logDemo.dll

> 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.

Summary

The above example shows how JNBridgePro can be used to create applications with .NET Core code that calls Java code. The applications can be run on Windows or Linux (MacOS will be coming in a future release) and can use either TCP/binary or shared memory communications. On Windows, the applications can run as 32-bit or 64-bit processes; on Linux, they can only run as 64-bit processes. .NET Core 3.0 (or later) is required on the .NET Core side.

This example is based on the “log demo” example that comes with the JNBridgePro installation, and which targets .NET Framework. Where using JNBridgePro with .NET Core differs from using it with .NET Framework, the differences are annotated in this document. Please use the example files for the “log demo” that come with the JNBridgePro installation.