Example: Calling .NET Core code from Java code

Version 10.1

SPANNING JAVA & .NET

jnbridge.com

September 6, 2019

Introduction

This document shows how to use JNBridgePro to construct and run a simple Java application that calls code in a .NET Core class library. The purpose of the document is to show how such an application is constructed, configured, and deployed, and to highlight the differences between using JNBridgePro to enable Java to call .NET Core, and using JNBridgePro to enable Java to call .NET Framework (which was the only scenario supported before JNBridgePro 10.0). We will show how to configure and run the example using both TCP/binary and shared memory, and on both Windows and Linux.

The ways in which Java code actually calls proxies is unchanged from previous versions, and will not be the focus of this document. For more details on the use of proxies, please see the *Users' Guide* and the .NET Framework-targeted demos that come with the JNBridgePro installation.

Example code

The Java code in this example will call a simple class in a .NET Core 3.0 class library DotNetClasses.dll. The class library contains a single class:

```
namespace DotNetClasses
{
    public class MyDotNetClass
    {
        public string StringValue { get; }

        public MyDotNetClass(string stringValue)
        {
            StringValue = stringValue;
        }
    }
}
```

The Java application is also simple, containing a single class with a main method that configures JNBridgePro, then instantiates MyDotNetClass and accesses the StringValue property:

```
import com.jnbridge.jnbcore.DotNetSide;

public class MainClass
{
   public static void main(String[] args)
   {
          DotNetSide.init(args[0]);
          System.out.println("About to instantiate MyDotNetClass");
          DotNetClasses.MyDotNetClass c =
                 new DotNetClasses.MyDotNetClass("test string");
          String s = c.Get_StringValue();
          System.out.println("StringValue = " + s);
   }
}
```

Generating the proxies

Java proxy jar files for .NET Core DLLs must be generated using the .NET Core-targeted command-line proxy generation tool. To generate the proxy for DotNetClasses.MyDotNetClass, and supporting classes, use the following command line (assuming the current working directory is the folder containing DotNetClasses.dll):

%JNBRIDGE% is the path to the JNBridgePro 10.1 installation.

```
dotnet "%JNBRIDGE%\DotNet Core\jnbproxy.dll" /pd j2n /al ".\DotNetClasses.dll"
    /d . /n proxies
    /jp "%JNBRIDGE%\jnbcore\jnbcore.jar"
    /bp "%JNBRIDGE%\jnbcore\bcel-5.1-jnbridge.jar"
    /pro b /host localhost /port 8085
    /java "C:\Program Files\Java\jdk-11\bin\java.exe"
    DotNetClasses.MyDotNetClass
```

Note that the command line above should actually be on a single line; we have reformatted it onto separate lines for readability. The command line will cause a proxy file proxies.jar to be created in the current working directory. The proxy jar file will contain proxies for DotNetClasses.MyDotNetClass and for all supporting classes. For more information on options for the command-line version of the proxy generation tool, please see the *Users' Guide*.

Note: In order for the .NET Core-targeted proxy generation tool to work, a copy of the license file must be placed in the folder containing jnbproxy.dll. If the license file is not there, an exception will be thrown and an error will be reported. We hope to remedy this issue in an upcoming release so that the licensing mechanism will look for the license file in the customary locations.

Note: For historical reasons, the command-line proxy generation tool has not had the ability to proxy every class in a DLL. Instead, it can be driven by a text file containing a class list, using the /f option. These class lists, which can be very large, were typically generated by the GUI-based proxy generation tool, but there is no .NET Core-targeted GUI-based proxy tool, so this is not an option. We recognize that this is a problem, and are providing a simple .NET Core-based tool to generate class list files containing the names of all classes in a list of DLLs; this list can then be used to drive the proxy generation tool. Please download the tool from the JNBridge website, or contact support@jnbridge.com. We expect to add the ability to proxy all classes in a DLL in an upcoming release.

Building the application

Build the application in the same way as you would for a Java-to-.NET Framework project. Compile the Java application, making sure to include proxies.jar, jnbcore.jar, and bcel-5.1-jnbridge.jar in the project's build classpath.

Deploying and running the application (with Windows)

Using TCP/binary communications: Create a folder named 'DotNetSide.' Copy into that folder the following files from the DotNet Core folder in the JNBridgePro installation:

- JNBDotNetSide.dll
- JNBDotNetSide.deps.json

- JNBDotNetSide.runtimeconfig.json
- jnbauth_x64.dll or jnbauth_x86.dll, or both, depending on whether this will run as a 64-bit or 32-bit process
- JNBShare.dll
- 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

Copy the `target` DLL DotNetClasses.dll into the DotNetSide folder.

Add a test file jnbridgeConfig.json into the folder, and put the following contents into it and save it away:

```
{
  "javaToDotNetConfig": {
    "scheme": "jtcp",
    "port": "8086"
    "useSSL": "false",
    "useClassWhiteList": "false",
  },
  "assemblyList": [ ".\\DotNetClasses.dll" ],
  "licenseLocation": {
    "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v10.1"
  }
}
```

Note that this example does not use security features such as SSL or class whitelisting. To use these security features, see the Users' Guide for more details.

In a command-line window, navigate to the DotNetSide folder and enter the following command:

```
dotnet JNBDotNetSide.dll
```

This starts the .NET side:

```
JNBDotNetSide, Copyright 2004-2019, JNBridge, LLC
Starting .NET Core-side server
Hit <return> to exit
```

To start the Java side, first create a new folder and name it 'JavaSide.' Inside this folder, place copies of the compiled MainClass.class binary, proxies.jar, jnbcore.jar, bcel-5.1-jnbridge.jar, and jnbcore_tcp.properties. The jnbcore.jar and bcel-5.1-jnbridge.jar can be found in the jnbcore subfolder in the JNBridgePro installation. jnbcore_tcp.properties should be a new file containing the following lines:

```
dotNetSide.serverType=tcp
dotNetSide.host=localhost
dotNetSide.port=8086
dotNetSide.useSSL=false
javaSide.useSSL=false
```

Note that this example does not use security features such as SSL or class whitelisting. To use these security features, see the Users' Guide for more details.

Then open a new command-line window, navigate to the JavaSide folder, and enter the following command:

```
java -cp ".;proxies.jar;jnbcore.jar;bcel-5.1-jnbridge.jar" MainClass
jnbcore_tcp.properties
```

This will start up the Java side, and will result in the following output:

```
About to instantiate MyDotNetClass
StringValue = test string
```

Using shared memory communications: Create a folder named 'appBase,' and place inside it the following files from the DotNet Core subfolder in the JNBridgePro 10.1 installation:

- jnbauth_x64.dll or jnbauth_x86.dll, or both, depending on whether this will run as a 64-bit or 32-bit process
- JNBJavaEntry_x64.dll or JNBJavaEntry_x86.dll, or both, depending on whether this will run as a 64-bit or 32-bit process
- JNBSharedMem_x64.dll or JNBSharedMem_x86.dll, or both, depending on whether this will run as a 64-bit or 32-bit process
- JNBJavaEntry2.dll
- JNBShare.dll
- 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

In ‘appBase,’ create a new text file jnbridgeConfig.json, add the following content, and save it away:

```
{
  "licenseLocation": {
    "directory": "C:\\Program Files (x86)\\JNBridge\\JNBridgePro v10.1"
  }
}
```

If you haven’t done so already when you tested TCP/binary communications, create a folder ‘JavaSide,’ and copy into it the compiled MainClass.class binary file, proxies.jar, jnbcore.jar, and bcel-5.1-jar, and DotNetClasses.dll.

In the JavaSide folder, create a new text file jnbcore_sharedmem.properties, add the following content, and save it away:

```
dotNetSide.serverType=sharedmem
dotNetSide.assemblyList.1=./DotNetClasses.dll
dotNetSide.javaEntry=path to appBase folder
dotNetSide.appBase=path to appBase folder
dotNetSide.coreClrPath=C:/Program Files/dotnet/shared/Microsoft.NETCore.App/3.0.0-
preview9-19423-09
```

Note that all paths should use forward-slashes (‘/’) rather than backslashes (‘\’). Also note that the coreClrPath may be different depending on the version of .NET Core being used, and whether a 32-bit or 64-bit runtime is being used.

Now, open a new command-line window, navigate to the JavaSide folder, and run the following command to run the application:

```
java -cp ".;proxies.jar;jnbcore.jar;bcel-5.1-jnbridge.jar" MainClass
jnbcore_sharedmem.properties
```

Deploying and running the application (with Linux)

Deploying and running Java-to-.NET Core projects on Linux is similar to doing so on Windows. Below, we discuss the differences.

Using TCP/binary communications: On your Linux machine, create the ‘DotNetSide’ and ‘JavaSide’ folders as described in the Windows section, and copy files into those folders as described in the Windows section, with the following differences in the DotNetSide folder:

- Use jnbauth_x64.so instead of jnbauth_x64.dll or jnbauth_x86.dll.
- Copy the license file (which will be different on the Linux machine from the one on the Windows machine) into the DotNetSide folder

- Alter jnbridgeConfig.json so that the paths to DotNetClasses.dll and the license file are correct for your machine. For example:

```
{
  "javaToDotNetConfig": {
    "scheme": "jtcp",
    "port": "8086"
    "useSSL": "false",
    "useClassWhiteList": "false",
  },
  "assemblyList": [ "./DotNetClasses.dll" ],
  "licenseLocation": {
    "directory": "/home/myAccount/J2NExample/DotNetSide”
  }
}
```

Open a terminal window, navigate to the DotNetSide folder, and run the following command to start the .NET side (assuming that the dotnet command is in your path):

```
dotnet JNBDotNetSide.dll
```

Open a terminal window, navigate to the JavaSide folder, and run the following command:

```
java -cp ".:proxies.jar:jnbcore.jar:bcel-5.1-jnbridge.jar" MainClass
jnbcore_tcp.properties
```

Note that the classpath is colon-separated on Linux, rather than semicolon-separated, as it is on Windows.

Using shared memory communications: On your Linux machine, create the ‘appBase’ and ‘JavaSide’ folders as described in the Windows section, and copy files into those folders, with the following differences in the appBase folder:

- Use jnbauth_x64.so instead of jnbauth_x64.dll or jnbauth_x86.dll
- Use libJNBJavaEntry_x64.so instead of JNBJavaEntry_x64.dll or JNBJavaEntry_x86.dll
- Use libJNBSharedMem_x64.so instead of JNBSharedMem_x64.dll or JNBSharedMem_x86.dll
- Copy the license file (which will be different on the Linux machine from the one on the Windows machine) into the appBase folder
- Alter jnbridgeConfig.json so that the paths to DotNetClasses.dll and the license file are correct for your machine. For example:

```
{
  "licenseLocation": {
    "directory": "/home/myAccount/J2NExample/appBase”
  }
}
```

In the JavaSide folder, alter the file jnbcore_sharedmem.properties so that the paths are correct for your machine. For example:

```
dotNetSide.serverType=sharedmem
dotNetSide.assemblyList.1=./DotNetClasses.dll
dotNetSide.javaEntry=path to appBase folder
dotNetSide.appBase=path to appBase folder
dotNetSide.coreClrPath=/home/myAccount/shared/Microsoft.NETCore.App/3.0
.0-preview9-19423-09
```

Note that the value of the dotNetSide.coreClrPath property may be different on your machine.

To run the application, open a terminal window, navigate to the JavaSide folder, and run the following command:

```
java -cp ".:proxies.jar:jnbcore.jar:bcel-5.1-jnbridge.jar" MainClass
jnbcore_sharedmem.properties
```

Note that the classpath is colon-separated on Linux, rather than semicolon-separated, as it is on Windows.

Summary

The above example shows how JNBridgePro can be used to create applications with Java code that calls .NET Core 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.