



## **Demo: Embedding Windows Presentation Foundation elements inside a Java GUI application**

**Version 6.0**

JNBridge, LLC  
[www.jnbridge.com](http://www.jnbridge.com)

COPYRIGHT © 2002–2011 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.

May 6, 2011

## Introduction

This document shows how a .NET Windows Presentation Foundation (WPF) control can be embedded inside a Java GUI application (either an AWT, Swing, or SWT application). If you are unfamiliar with JNBridgePro, we recommend that you work through one of the other demos first. We recommend working through the “Java-to-.NET demo,” which will work through the entire process of generating proxies and setting up, configuring, and running an interop project. This current document assumes such knowledge, and is mainly a guided tour of the code and configuration information necessary to embed .NET GUI elements inside GUI-based Java applications.

## The .NET GUI component

In this example, we have provided a Windows Presentation Foundation control, `WPFControlDemo.UserControl1`. This control is adapted from the example in the first chapter of the book *Essential Windows Presentation Foundation*, by Chris Anderson (Addison-Wesley).

Any WPF component to be embedded inside a Java GUI application must be derived from `System.Windows.Controls.Control`. The `UserControl1` class above is derived from `System.Windows.Controls.UserControl`, which is a subclass of `System.Windows.Controls.Control`. `UserControl1` contains a `TextBox`, a `Button`, and an `Image`, and also include an animated cube “painted” with copies of the `TextBox`, `Button`, and `Image`. `UserControl1` also contains a public method makes it possible to register an event handler for the button. `UserControl1` should look like this:



## Generating the proxies

We have provided a proxy jar file `proxies.jar`, which contains the proxies for `WPFControlDemo.UserControl1`, plus all supporting classes. However, it is straightforward to generate the proxies oneself.

If you generate the proxy jar file, you should make sure that `WPFControlDemo.dll` (the dll with the WPF control), and `System.Windows.Forms` and `PresentationCore` (from the GAC) are in the proxy generator's assembly list. Make sure you have the correct version of `System.Windows.Forms`. Also, you must explicitly add `System.Windows.Forms.Application` and `System.Windows.Media.Composition.DUCE` (and supporting classes) to the proxy generation project and proxy them. You will be directly calling the proxy of `System.Windows.Forms.Application` (which resides in the assembly `System.Windows.Forms`), and other proxies reference `System.Windows.Media.Composition.DUCE` (which resides in `PresentationCore`) and some development environments may report an error if it isn't present.

## Embedding the Java component inside the Windows Form

We have prepared a Java AWT application to contain the embedded .NET component.

Inside the application's main method, we have added the following lines

```
// create the WPF control
UserControl1 c = new UserControl1();
// wrap it so it can be embedded
DotNetControl dnc = new DotNetControl(c);
// size it
dnc.setSize(300, 390);
// embed it
f.add(dnc, dncConstraints);
```

The code first instantiates the proxy for the .NET component (`UserControl1`), then embeds it inside a special wrapper, `com.jnbridge.embedding.DotNetControl`, which inherits from `java.awt.Canvas`, and which allows the .NET component to be used wherever a Java component is expected. The `DotNetControl` is then resized to be the same size as the embedded .NET control, and the control is added to the application's `Frame` object, along with previously defined layout constraints.

We have also created a callback class that implements `WPFControlDemo.clickDelegate`, the interface representing the button's event handler, and which will be executed whenever the WPF component's button is clicked:

```
static TextField echo;

public static class ClickEventHandler implements clickDelegate
{
    public void Invoke(String message)
    {
        echo.setText(message);
    }
}
```

The callback code takes the text from the WPF TextBox (passed as a parameter), and writes it to the Java TextField echo. The callback is instantiated and registered with the WPF component as an clickDelegate by the following line in the main method:

```
c.registerClickDelegate(new ClickEventHandler());
```

Before any proxy call, JNBridgePro is started and configured through a call to `com.jnbridge.jnbc.DotNetSide.init()`:

```
DotNetSide.init(props);
```

where `props` is a Java Properties object containing the configuration information. The next section explains the configuration. It is also possible to put this information into a `.properties` file and supply the path to the `.properties` file as an argument to `DotNetSide.init()`.

Finally, we must drive the .NET-side event loop. We do so by calling `System.Windows.Forms.Application.DoEvents()` inside a loop:

```
while(!mustStop)
{
    Application.DoEvents();
}
```

We terminate the loop by setting the `mustStop` flag inside a `windowClosing()` event handler that we install in the application's Frame:

```
f.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent evt) {
        Window w = evt.getWindow();
        w.setVisible(false);
        w.dispose();
        mustStop = true;
    }
});
```

Note that the `mustStop` flag must be set as the last action in `windowClosing()`. If it is set before `w.dispose()`, the application's process may not terminate.

## Configuring and running the application

The project is constructed in the same way as other .NET-to-Java interop projects. The communications mechanism must be shared memory:

```
Properties props = new Properties();
props.put("dotNetSide.serverType", "sharedmem");
props.put("dotNetSide.assemblyList.1",
    "../DotNet/bin/Debug/WPFControlDemo.dll");
props.put("dotNetSide.javaEntry",
    "C:/Program Files/JNBridge/JNBridgePro v6.0/2.0-
targeted/JNBJavaEntry.dll");
props.put("dotNetSide.appBase",
    "C:/Program Files/JNBridge/JNBridgePro v6.0/2.0-targeted");
props.put("dotNetSide.apartmentThreadingModel", "STA");
```

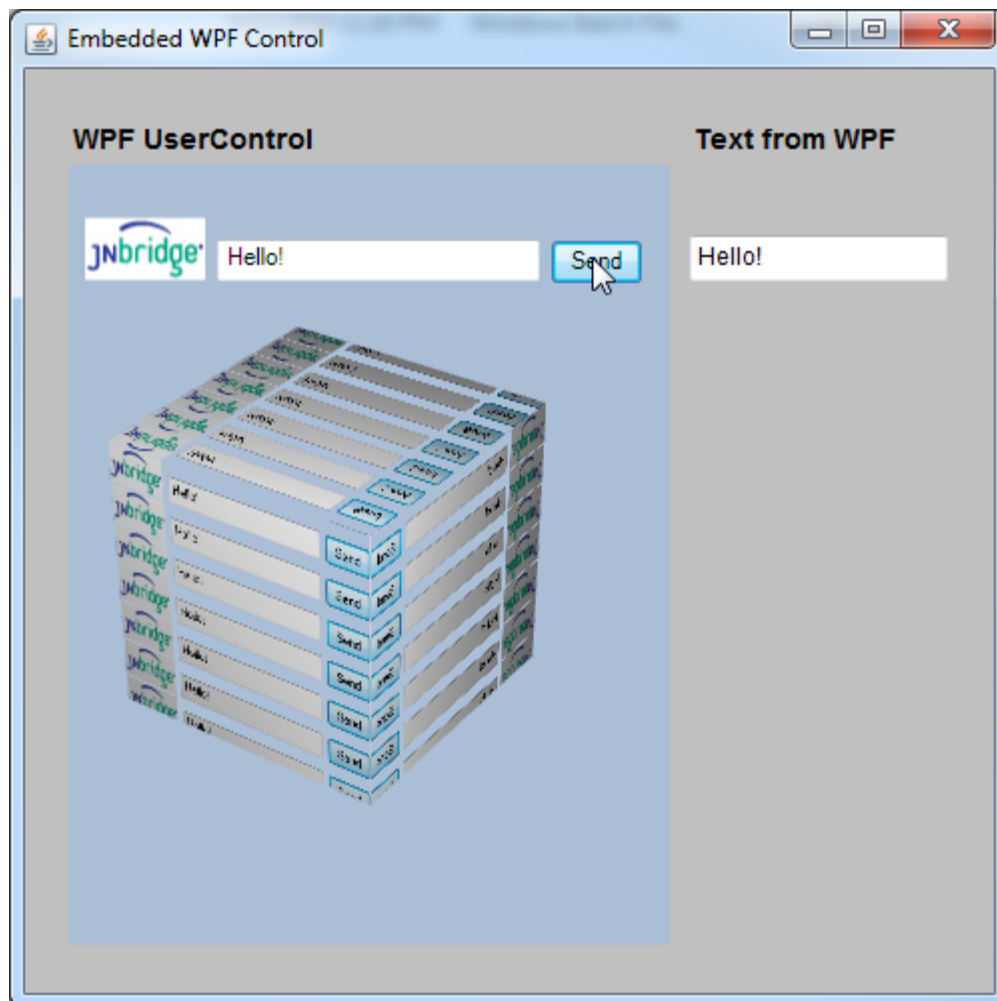
The properties above specify that shared memory is to be used. They also specify the location of the DLL containing the WPF control. (Since `WPFControlDemo.dll` references `System.Windows.Forms.dll`, it will be loaded automatically, although there is no harm in also adding

it to the assembly list in another property. Following that, we specify the location of JNBJavaEntry.dll, and the folder in which jnbshare.dll, jnbsharedmem.dll, jnbjavaentry2.dll, jnbwpfembedding.dll, and rlm913\_x86.dll (or rlm913\_x64.dll) reside. Finally, we must specify that the .NET side will use the single-threaded apartment model. (If we leave this out, an exception will be thrown.)

It is also possible to leave out the dotNetSide.appBase property, if you must make sure that jnbshare.dll, jnbsharedmem.dll, and jnbjavaentry2.dll are all installed in the Global Assembly Cache (GAC), but we recommend using the dotNetSide.appBase property.

We supply a .bat file runJava.bat to encapsulate the commands needed to start the application

When the Java application is run, the .WPF component appears embedded in the Java application, and when text is entered in the WPF component's text field and the "Send" button is clicked, the text will appear in the Java application's text box, illustrating how the WPF and Java GUI elements communicate.



## Embedding .NET controls in SWT applications

It is also possible to embed a .NET control in an SWT application. The process is identical to that of embedding in AWT/Swing applications, as described above, except for the following differences:

- Instead of wrapping the embedded .NET control inside `com.jnbridge.embedding.DotNetControl`, you must embed it inside `com.jnbridge.embedding.DotNetSWTControl`, which inherits from `org.eclipse.swt.widgets.Composite`. The constructor for `DotNetSWTControl` takes as arguments both the embedded .NET control and the parent SWT composite object that will contain it:

```
DotNetSWTControl myControl = new DotNetSWTControl(dotNetControl, parent);  
myControl.setBounds(0, 50, 655, 303); // and set its size and location
```

Note that the `DotNetSWTControl` should be positioned and sized after being created.

- The Win32 version of `swt.jar` must be in the run-time classpath.
- The path of the folder containing the Win32 version of the SWT dlls (typically the `plugins\org.eclipse.swt.win32_3.1.0\os\win32\x86` folder in the Eclipse installation folder) must be in the Java library path. For example, for Eclipse 3.1, add `-Djava.library.path="C:\Program Files\Eclipse 3.1\eclipse\plugins\org.eclipse.swt.win32_3.1.0\os\win32\x86"` to the command-line immediately after the `java` command. This path will differ depending on your installation and the version of Eclipse you are using.

## Summary

The above example shows how simple it is to embed a Windows Presentation Foundation (WPF) component inside a Java GUI application. This embedding can be accomplished in three steps:

- Proxy the WPF component and the supporting classes
- Write code to wrap the Java component's proxy in the special `DotNetControl` wrapper class, size it, and add that `DotNetControl` object to the containing Java Frame.
- Create Java classes to implement any event handlers, and register them with the .NET control's proxy,

We also describe how to embed .NET controls inside SWT applications. The main difference between embedding in SWT applications and embedding in AWT/SWT applications is that you must use the `DotNetSWTControl` wrapper class when embedding in SWT applications.